Understanding the Re-Engineering of Variant-Rich Systems: An Empirical Work on Economics, Knowledge, Traceability, and Practices

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von M.Sc. Jacob Krüger

geb. am 12.03.1991 in Gardelegen

Gutachterinnen/Gutachter

Prof. Dr. rer. nat. habil. Gunter Saake
Prof. Dr.-Ing. Thomas Leich
Prof. dr. ir. Jan Bosch
Univ.-Prof. Mag. Dr. rer. soc. oec. Rick Rabiser

Magdeburg, den 10.09.2021

Otto-von-Guericke University Magdeburg

Faculty of Computer Science

Dissertation

Understanding the Re-Engineering of Variant-Rich Systems

An Empirical Work on Economics, Knowledge, Traceability, and Practices

Author:

Jacob Krüger

10.09.2021

Reviewers:

Prof. Dr. rer. nat. habil. Gunter Saake
Otto-von-Guericke University Magdeburg, Germany

Prof. Dr.-Ing. Thomas Leich
Harz University of Applied Sciences Wernigerode, Germany

Prof. dr. ir. Jan Bosch
Chalmers University of Technology Gothenburg, Sweden

Univ.-Prof. Mag. Dr. rer. soc. oec. Rick Rabiser
Johannes Kepler University Linz, Austria

Krüger, Jacob:
Understanding the Re-Engineering of Variant-Rich Systems
An Empirical Work on Economics, Knowledge, Traceability, and Practices
Dissertation, Otto-von-Guericke University Magdeburg, 2021.

Abstract

Context: Most modern software systems exist in different variants to address a variety of requirements, such as customer requests, regulations, or hardware restrictions. To benefit from the high commonality between variants, such *variant-rich systems* (e.g., Linux kernel, automotive software, webservers) usually reuse existing artifacts (which implement so-called *features*). In fact, in many organizations, a variant-rich system establishes itself over time through developers using clone & own (i.e., copying and adapting a variant) as a reuse strategy. Typically, the maintenance burden of having numerous separated variants forces an organization to *re-engineer* its cloned variants into a reusable platform by adopting concepts of software product-line engineering. Despite the practical prevalence of this re-engineering scenario, most research on decision support has focused on the proactive adoption (i.e., starting from scratch) of platform engineering.

Objective: In this dissertation, we empirically study four closely related properties in the context of variant-rich systems, namely economics, knowledge, traceability, and practices. Note that, while we focus on the re-engineering of cloned variants into a platform, many of our findings are relevant for engineering any (variant-rich) software system. More precisely, we aim to contribute an empirics-based body-of-knowledge that can guide organizations in planning and monitoring their (re-)engineering projects. In parallel, our studies advance on educated guesses, which are widely used to reason on variant-rich systems. To this end, we aim to provide economical data that allows to compare and understand the differences between clone & own and platform engineering. Since our findings highlight the economical impact and close relation of knowledge and feature traceability, we further aim to provide a detailed understanding of these two properties in the context of re-engineering projects. Finally, we aim to synthesize all of our findings and connect them to contemporary software-engineering practices to derive processes and recommendations for planning, initiating, steering, and monitoring platform engineering.

Method: To address our objectives, we relied on a number of empirical research methods to collect data from various sources. In most cases, we built on eliciting qualitative data from the literature, which we identified through systematic literature reviews. To enrich that data, we conducted interview and online surveys, measurement and multi-case studies, as well as experiments; which we selected and employed based on their feasibility to address a certain objective. By synthesizing from different sources, we aimed to improve the validity of our data to provide reliable insights for researchers and practitioners.

Results: On an abstract level, we can summarize four key contributions. First, we contribute a rich dataset on the economics of (re-)engineering variant-rich systems, from which we derive the core insight that moving towards platform engineering (e.g., more systematic clone management) is economically promising. Second, we contribute an understanding of developers' memory and how to support their knowledge needs, leading to the core insight that expensive recovery activities can be mitigated by enforcing suitable documentation techniques (e.g., feature traceability). Third, we contribute insights on how different feature traces impact developers' program comprehension, based on which our core insight is that feature traceability should ideally be independent of configurability. Finally, we contribute a process model and recommendations on how to (re-)engineer variant-rich systems, with our core insight being that carefully planning and periodically assessing a variant-rich system helps to exploit its full potential (e.g., in terms of cost savings).

Conclusion: Overall, we provide detailed insights into four important properties that help organizations as well as researchers understand and guide (re-)engineering projects for variant-rich systems. We discuss these insights and their connections to each other as well as to contemporary software-engineering practices, enabling others to adopt them to different scenarios. So, our contributions involve the synthesis and considerable extension of the existing body-of-knowledge on (re-)engineering variant-rich systems.

Keywords

Software reuse, Software economics, Re-engineering, Platform engineering, Clone & own, Software product line, Knowledge, Traceability

ACM CCS Concepts

• Software and its engineering → Software development process management; Reusability; Software evolution

Zusammenfassung

Kontext: Moderne Softwaresysteme werden in einer Vielzahl an Varianten angeboten um verschiedenste Anforderungen von Kunden, Regulierungen oder der Hardware zu bedienen. Da sich die einzelnen Varianten solcher variantenreichen Systeme (z.b. der Linux Kernel, Fahrzeugsoftware oder Webserver) sehr ähneln, können Entwickler die Wiederverwendung existierender Artefakte, die ein bestimmtes Feature implementieren, während der Entwicklung ausnutzen. Dabei werden in den meisten Fällen Varianten zuerst nur kopiert und an neue Anforderungen angepasst. Da solche kopierten Varianten voneinander losgelöst sind, wird deren Wartung in den meisten Fällen immer kostenintensiver, was oftmals dazu führt, dass die Entwickler eine wiederverwendbare Softwareplattform aus diesen extrahieren. Obwohl dieser extraktive Ansatz in der Praxis am verbreitetsten ist, fokussiert sich die Forschung meistens darauf, Entscheidungshilfen für die proaktive Neuentwicklung einer Plattform zu entwickeln.

Ziele: In dieser Dissertation werden empirische Methoden genutzt, um vier zusammenhängende Eigenschaften mit Bezug zu variantenreichen Systemen zu untersuchen: Kosten, Wissen, Nachvollziehbarkeit und Verfahren. Anzumerken ist, dass trotz unseres Fokus auf den extraktiven Ansatz, viele unserer Erkenntnisse für jegliche (variantenreiche) Softwaresysteme relevant sind. Wir sammeln empirische Daten um den wissenschaftlichen Stand in Bezug auf variantenreiche Systeme zu erweitern und Unternehmen bei der Planung und Beobachtung von Softwareprojekten zu unterstützen. Dazu vergleichen wir zuerst die Kosten um Varianten basierend auf Kopien oder einer Plattform zu entwickeln. Bei dieser Untersuchung haben wir festgestellt, dass insbesondere das Wissen der Entwickler und dessen Nachvollziehbarkeit die Entwicklung variantenreicher Systeme beeinflussen. Nachdem wir diese beiden Eigenschaften detaillierter untersucht haben, kombinieren wir alle unsere Ergebnisse um Methodiken und Empfehlungen für die Planung, Initialisierung und Beobachtung von Softwareplattformen zu definieren.

Methodik: Unsere Forschung basiert auf verschiedenen empirischen Methoden um Daten aus unterschiedlichen Quellen zu erheben. Durch dieses Vorgehen konnten wir die Validität unserer Ergebnisse verbessern und liefern damit eine Alternative zu den begründeten Vermutungen auf denen sich Entscheidungen bezüglich variantenreicher Systeme bisher meist stützen. Wir haben meist systematische Literaturreviews genutzt um qualitative Daten aus existierenden Arbeiten zu erheben. Um diese Daten zu komplementieren, haben wir dem jeweiligen Ziel angepasste Methoden genutzt, beispielsweise Interviews, Umfragen, Fallstudien oder Experimente.

Ergebnisse: Stark zusammengefasst können wir vier Kernergebnisse definieren. Erstens liefern wir Daten zu den Kosten der Entwicklung variantenreicher Systeme, die zeigen, dass Unternehmen darauf hinarbeiten sollten, ihre Entwicklung in Richtung einer Plattform zu systematisieren. Zweitens bieten wir Einsichten dazu, welches Wissen Entwickler benötigen und aus welchen Quellen sie dieses gewinnen können, wobei unsere zentrale Einsicht die Notwendigkeit geeigneter Dokumentationen ist. Drittens haben wir die Nachvollziehbarkeit von Features als eine Art der Dokumentation untersucht und festgestellt, dass diese idealerweise nicht auf Techniken für die Konfiguration von Software aufbaut. Zuletzt haben wir Methodiken und Empfehlungen aus unseren Ergebnissen synthetisiert und die Einsicht gewonnen, dass die Planung und regelmäßige Evaluierung variantenreicher Systeme hilft, deren Nutzen zu maximieren.

Fazit: Insgesamt beschreiben wir in dieser Dissertation detaillierte Einsichten zu vier essentiellen Eigenschaften für die Entwicklung und Extraktion von variantenreichen Systemen. Damit diese Eigenschaften an verschiedene Szenarien angepasst werden können, diskutieren wir ihre Zusammenhänge miteinander und mit aktuellen Methoden der Softwareentwicklung. Daraus resultiert eine Synthese und erhebliche Erweiterung des existierenden Wissensstandes bezüglich der Entwicklung variantenreicher Systeme.

List of Publications

In the following, I first list my reviewed publications that contribute at least partly to this dissertation. Note that I provide mappings between these publications and this dissertation in the introduction, the beginning of each chapter, and in the actual text. Furthermore, I classified these publications according to the CRediT author statement (in parentheses).[1] Afterwards, I list my publications that developed in parallel to my core research.

Publications Contributing to this Dissertation

1 **Jacob Krüger**, Gül Çalıklı, Thorsten Berger, and Thomas Leich. How Explicit Feature Traces Did Not Impact Developers' Memory. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021. (Conceptualization; Methodology; Software; Validation; Investigation; Data Curation; Writing - Original Draft; Writing - Review & Editing; Visualization).

2 Robert Lindohf, **Jacob Krüger**, Erik Herzog, and Thorsten Berger. Software Product-Line Evaluation in the Large. *Empirical Software Engineering*, 26(30):1–41, 2021. (Conceptualization; Methodology; Investigation; Writing - Original Draft; Writing - Review & Editing; Supervision).

3 Wolfram Fenske, **Jacob Krüger**, Maria Kanyshkova, and Sandro Schulze. #ifdef Directives and Program Comprehension: The Dilemma between Correctness and Preference. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 255–266. IEEE, 2020. (Investigation; Writing - Original Draft; Writing - Review & Editing).

4 **Jacob Krüger** and Thorsten Berger. Activities and Costs of Re-Engineering Cloned Variants Into an Integrated Platform. In *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 21:1–10. ACM, 2020a. (Conceptualization; Methodology; Investigation; Data Curation; Writing - Original Draft; Writing - Review & Editing; Visualization).

5 **Jacob Krüger** and Thorsten Berger. An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 432–444. ACM, 2020b. (Conceptualization; Methodology; Validation; Investigation; Data Curation; Writing - Original Draft; Writing - Review & Editing; Visualization).

6 **Jacob Krüger** and Regina Hebig. What Developers (Care to) Recall: An Interview Survey on Smaller Systems. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 46–57. IEEE, 2020. (Conceptualization; Methodology; Investigation; Data Curation; Writing - Original Draft; Writing - Review & Editing; Visualization).

7 **Jacob Krüger**, Sebastian Krieter, Gunter Saake, and Thomas Leich. EXtracting Product Lines from vAriaNTs (EXPLANT). In *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 13:1–2. ACM, 2020a. (Conceptualization; Writing - Original Draft; Writing - Review & Editing).

8 **Jacob Krüger**, Wardah Mahmood, and Thorsten Berger. Promote-pl: A Round-Trip Engineering Process Model for Adopting and Evolving Product Lines. In *International Systems and Software Product Line Conference (SPLC)*, pages 2:1–12. ACM, 2020b. (Conceptualization; Methodology; Investigation; Writing - Original Draft; Writing - Review & Editing; Visualization).

[1] https://www.elsevier.com/authors/policies-and-guidelines/credit-author-statement

9 **Jacob Krüger**, Sebastian Nielebock, and Robert Heumüller. How Can I Contribute? A Qualitative Analysis of Community Websites of 25 Unix-Like Distributions. In *International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 324–329. ACM, 2020c. (Conceptualization; Methodology; Investigation; Writing - Original Draft; Writing - Review & Editing).

10 Kai Ludwig, **Jacob Krüger**, and Thomas Leich. FeatureCoPP: Unfolding Preprocessor Variability. In *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 24:1–9. ACM, 2020. (Conceptualization; Writing - Original Draft; Writing - Review & Editing).

11 Jonas Åkesson, Sebastian Nilsson, **Jacob Krüger**, and Thorsten Berger. Migrating the Android Apo-Games into an Annotation-Based Software Product Line. In *International Systems and Software Product Line Conference (SPLC)*, pages 103–107. ACM, 2019. (Conceptualization; Methodology; Data Curation; Writing - Original Draft; Writing - Review & Editing; Supervision).

12 Jamel Debbiche, Oskar Lignell, **Jacob Krüger**, and Thorsten Berger. Migrating Java-Based Apo-Games into a Composition-Based Software Product Line. In *International Systems and Software Product Line Conference (SPLC)*, pages 98–102. ACM, 2019. (Conceptualization; Methodology; Data Curation; Writing - Original Draft; Writing - Review & Editing; Supervision).

13 **Jacob Krüger**. Tackling Knowledge Needs during Software Evolution. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1244–1246. ACM, 2019a. (Conceptualization; Writing - Original Draft; Writing - Review & Editing; Visualization).

14 **Jacob Krüger**. Are You Talking about Software Product Lines? An Analysis of Developer Communities. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 11:1–9. ACM, 2019b. (Conceptualization; Methodology; Validation; Investigation; Data Curation; Writing - Original Draft; Writing - Review & Editing; Visualization).

15 **Jacob Krüger**, Thorsten Berger, and Thomas Leich. Features and How to Find Them - A Survey of Manual Feature Location. In *Software Engineering for Variability Intensive Systems*, pages 153–172. CRC Press, 2019a. (Conceptualization; Methodology; Investigation; Data Curation; Writing - Original Draft; Writing - Review & Editing).

16 **Jacob Krüger**, Gül Çalıklı, Thorsten Berger, Thomas Leich, and Gunter Saake. Effects of Explicit Feature Traceability on Program Comprehension. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 338–349. ACM, 2019b. (Conceptualization; Methodology; Software; Validation; Formal analysis; Investigation; Data Curation; Writing - Original Draft; Writing - Review & Editing; Visualization).

17 **Jacob Krüger**, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. Where is My Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software*, 152:239–253, 2019c. (Conceptualization; Methodology; Investigation; Data Curation; Writing - Original Draft; Writing - Review & Editing).

18 Kai Ludwig, **Jacob Krüger**, and Thomas Leich. Covert and Phantom Features in Annotations: Do They Impact Variability Analysis? In *International Systems and Software Product Line Conference (SPLC)*, pages 218–230. ACM, 2019. (Conceptualization; Methodology; Validation; Writing - Original Draft; Writing - Review & Editing).

19 Damir Nešić, **Jacob Krüger**, Ştefan Stănciulescu, and Thorsten Berger. Principles of Feature Modeling. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 62–73. ACM, 2019. (Conceptualization; Methodology; Investigation; Data Curation; Writing - Original Draft; Writing - Review & Editing).

20 Sebastian Nielebock, Dariusz Krolikowski, **Jacob Krüger**, Thomas Leich, and Frank Ortmeier. Commenting Source Code: Is It Worth It for Small Programming Tasks? *Empirical Software Engineering*, 24(3):1418–1457, 2019. (Formal analysis; Writing - Original Draft; Writing - Review & Editing).

21 Daniel Strüber, Mukelabai Mukelabai, **Jacob Krüger**, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *International Systems and Software Product Line Conference (SPLC)*, pages 177–188. ACM, 2019. (Conceptualization; Methodology; Investigation; Data Curation; Writing - Original Draft; Writing - Review & Editing).

22 Sebastian Krieter, **Jacob Krüger**, and Thomas Leich. Don't Worry About it: Managing Variability On-The-Fly. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 19–26. ACM, 2018. (Conceptualization; Methodology; Software; Validation; Formal analysis; Investigation; Data Curation; Writing - Original Draft; Writing - Review & Editing; Visualization; Supervision).

23 **Jacob Krüger**. When to Extract Features: Towards a Recommender System. In *International Conference on Software Engineering (ICSE)*, pages 518–520. ACM, 2018a. (Conceptualization; Writing - Original Draft; Writing - Review & Editing; Visualization).

24 **Jacob Krüger**. Separation of Concerns: Experiences of the Crowd. In *Symposium on Applied Computing (SAC)*, pages 2076–2077. ACM, 2018b. (Conceptualization; Methodology; Investigation; Data Curation; Writing - Original Draft; Writing - Review & Editing; Visualization).

25 **Jacob Krüger**, Wolfram Fenske, Thomas Thüm, Dirk Aporius, Gunter Saake, and Thomas Leich. Apo-Games - A Case Study for Reverse Engineering Variability from Cloned Java Variants. In *International Systems and Software Product Line Conference (SPLC)*, pages 251–256. ACM, 2018a. (Conceptualization; Methodology; Data Curation; Writing - Original Draft; Writing - Review & Editing; Visualization).

26 **Jacob Krüger**, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. Towards a Better Understanding of Software Features and Their Characteristics. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 105–112. ACM, 2018b. (Formal analysis; Investigation; Data Curation; Writing - Original Draft; Writing - Review & Editing; Visualization).

27 **Jacob Krüger**, Kai Ludwig, Bernhard Zimmermann, and Thomas Leich. Physical Separation of Features: A Survey with CPP Developers. In *Symposium on Applied Computing (SAC)*, pages 2042–2049. ACM, 2018c. (Conceptualization; Methodology; Writing - Original Draft; Writing - Review & Editing; Visualization; Supervision).

28 **Jacob Krüger**, Marcus Pinnecke, Andy Kenner, Christopher Kruczek, Fabian Benduhn, Thomas Leich, and Gunter Saake. Composing Annotations Without Regret? Practical Experiences Using FeatureC. *Software: Practice and Experience*, 48(3):402–427, 2018d. (Conceptualization; Methodology; Writing - Original Draft; Writing - Review & Editing).

29 **Jacob Krüger**, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. Do You Remember This Source Code? In *International Conference on Software Engineering (ICSE)*, pages 764–775. ACM, 2018e. (Conceptualization; Methodology; Formal analysis; Data Curation; Writing - Original Draft; Writing - Review & Editing; Visualization; Supervision).

30 Elias Kuiter, **Jacob Krüger**, Sebastian Krieter, Thomas Leich, and Gunter Saake. Getting Rid of Clone-And-Own: Moving to a Software Product Line for Temperature Monitoring. In *International Systems and Software Product Line Conference (SPLC)*, pages 189–189. ACM, 2018. (Data Curation; Writing - Original Draft; Writing - Review & Editing).

31 Mukelabai Mukelabai, Benjamin Behringer, Moritz Fey, Jochen Palz, **Jacob Krüger**, and Thorsten Berger. Multi-View Editing of Software Product Lines with PEoPL. In *International Conference on Software Engineering (ICSE)*, pages 81–84. ACM, 2018. (Writing - Original Draft; Writing - Review & Editing).

32 **Jacob Krüger**. Lost in Source Code: Physically Separating Features in Legacy Systems. In *International Conference on Software Engineering (ICSE)*, pages 461–462. IEEE, 2017. (Conceptualization; Writing - Original Draft; Writing - Review & Editing; Visualization).

33 **Jacob Krüger**, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. Finding Lost Features in Cloned Systems. In *International Systems and Software Product Line Conference (SPLC)*, pages 65–72. ACM, 2017. (Conceptualization; Methodology; Writing - Original Draft; Writing - Review & Editing; Visualization; Supervision).

34 **Jacob Krüger**, Wolfram Fenske, Jens Meinicke, Thomas Leich, and Gunter Saake. Extracting Software Product Lines: A Cost Estimation Perspective. In *International Systems and Software Product Line Conference (SPLC)*, pages 354–361. ACM, 2016a. (Conceptualization; Methodology; Software; Investigation; Data Curation; Writing - Original Draft; Writing - Review & Editing; Visualization).

35 **Jacob Krüger**, Ivonne Schröter, Andy Kenner, Christopher Kruczek, and Thomas Leich. FeatureCoPP: Compositional Annotations. In *International Workshop on Feature-Oriented Software Development (FOSD)*, pages 74–84. ACM, 2016b. (Conceptualization; Methodology; Investigation; Writing - Original Draft; Writing - Review & Editing; Visualization).

Additional Publications

Editorships

1 Rafael Capilla, Philippe Collet, Paul Gazzillo, **Jacob Krüger**, Roberto E. Lopez-Herrejon, Sarah Nadi, Gilles Perrouin, Iris Reinhartz-Berger, Julia Rubin, and Ina Schaefer, editors. *24th International Systems and Software Product Line Conference (SPLC) - Volume B*. ACM, 2020.

Journals

1 Elias Kuiter, Sebastian Krieter, **Jacob Krüger**, Gunter Saake, and Thomas Leich. VariED: An Editor for Collaborative, Real-Time Feature Modeling. *Empirical Software Engineering*, 26(24):1–47, 2021.

2 Robert Heumüller, Sebastian Nielebock, **Jacob Krüger**, and Frank Ortmeier. Publish or Perish, but do not Forget Sour Software Artifacts. *Empirical Software Engineering*, pages 1–32, 2020.

3 **Jacob Krüger**, Christian Lausberger, Ivonne von Nostitz-Wallwitz, Gunter Saake, and Thomas Leich. Search. Review. Repeat? An Empirical Study of Threats to Replicating SLR Searches. *Empirical Software Engineering*, 25(1):627–677, 2020.

4 Yusra Shakeel, **Jacob Krüger**, Ivonne von Nostitz-Wallwitz, Gunter Saake, and Thomas Leich. Automated Selection and Quality Assessment of Primary Studies. *Journal of Data and Information Quality*, 12(1):4:1–26, 2020.

5 **Jacob Krüger**, Mustafa Al-Hajjaji, Thomas Leich, and Gunter Saake. Mutation Operators for Feature-Oriented Software Product Lines. *Software Testing, Verification and Reliability*, 29(1-2):e1676:1–21, 2018.

Proceedings

1 Rand Alchokr, **Jacob Krüger**, Gunter Saake, and Thomas Leich. A Comparative Analysis of Article Recommendation Platforms. In *Joint Conference on Digital Libraries (JCDL)*. IEEE, 2021a.

2 Rand Alchokr, **Jacob Krüger**, Yusra Shakeel, Gunter Saake, and Thomas Leich. Understanding the Contributions of Junior Researchers at Software-Engineering Conferences. In *Joint Conference on Digital Libraries (JCDL)*. IEEE, 2021b.

3 Wesley K. G. Assunção, Inmaculada Ayala, **Jacob Krüger**, and Sébastien Mosser. VM4ModernTech: International Workshop on Variability Management for Modern Technologies. In *International Systems and Software Product Line Conference (SPLC)*, pages 202–202. ACM, 2021.

4 Wolfram Fenske, **Jacob Krüger**, Maria Kanyshkova, and Sandro Schulze. #ifdef Directives and Program Comprehension: The Dilemma between Correctness and Preference. In *Software Engineering (SE)*, pages 35–36. GI, 2021.

5 Philipp Gnoyke, Sandro Schulze, and **Jacob Krüger**. An Evolutionary Analysis of Software-Architecture Smells. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021.

6 Andy Kenner, Richard May, **Jacob Krüger**, Gunter Saake, and Thomas Leich. Safety, Security, and Configurable Software Systems: A Systematic Mapping Study. In *International Systems and Software Product Line Conference (SPLC)*, pages 148–159. ACM, 2021.

7 **Jacob Krüger** and Thorsten Berger. An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse. In *Software Engineering (SE)*, pages 69–70. GI, 2021.

8 **Jacob Krüger** and Regina Hebig. What Developers (Care to) Recall: An Interview Survey on Smaller Systems. In *Software Engineering (SE)*, pages 71–72. GI, 2021.

9 Elias Kuiter, **Jacob Krüger**, and Gunter Saake. Iterative Development and Changing Requirements: Drivers of Variability in an Industrial System for Veterinary Anesthesia. In *International Systems and Software Product Line Conference (SPLC)*, pages 113–122. ACM, 2021.

10 Sebastian Nielebock, Paul Blockhaus, **Jacob Krüger**, and Frank Ortmeier. AndroidCompass: A Dataset of Android Compatibility Checks in Code Repositories. In *International Coference on Mining Software Repositories (MSR)*, pages 535–539. IEEE, 2021a.

11 Sebastian Nielebock, Paul Blockhaus, **Jacob Krüger**, and Frank Ortmeier. An Experimental Analysis of Graph-Distance Algorithms for Comparing API Usages. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2021b.

12 Yusra Shakeel, Rand Alchokr, **Jacob Krüger**, Gunter Saake, and Thomas Leich. Are Altmetrics Proxies or Complements to Citations for Assessing Impact in Computer Science? In *Joint Conference on Digital Libraries (JCDL)*. IEEE, 2021.

13 Cem Sürücü, Bianying Song, **Jacob Krüger**, Gunter Saake, and Thomas Leich. Using Key Performance Indicators to Compare Software-Development Processes. In *Software Engineering (SE)*, pages 105–106. GI, 2021.

14 Sofia Ananieva, Sandra Greiner, Thomas Kühn, **Jacob Krüger**, Lukas Linsbauer, Sten Grüner, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, Sebastian Krieter, Christoph Seidl, S. Ramesh, Ralf Reussner, and Bernhard Westfechtel. A Conceptual Model for Unifying Variability in Space and Time. In *International Systems and Software Product Line Conference (SPLC)*, pages 15:1–12. ACM, 2020.

15 Wesley K. G. Assunção, **Jacob Krüger**, and Willian D. F. Mendonça. Variability Management meets Microservices: Six Challenges of Re-Engineering Microservice-Based Webshops. In *International Systems and Software Product Line Conference (SPLC)*, pages 22:1–6. ACM, 2020.

16 Andy Kenner, Stephan Dassow, Christian Lausberger, **Jacob Krüger**, and Thomas Leich. Using Variability Modeling to Support Security Evaluations: Virtualizing the Right Attack Scenarios. In *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 10:1–9. ACM, 2020.

17 **Jacob Krüger**, Sofia Ananieva, Lea Gerling, and Eric Walkingshaw. Third International Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution 2020). In *International Systems and Software Product Line Conference (SPLC)*, page 34:1. ACM, 2020a.

18 **Jacob Krüger**, Gül Çalıklı, Thorsten Berger, Thomas Leich, and Gunter Saake. Effects of Explicit Feature Traceability on Program Comprehension. In *Software Engineering (SE)*, pages 79–80. GI, 2020b.

19 Damir Nešić, **Jacob Krüger**, Ştefan Stănciulescu, and Thorsten Berger. Principles of Feature Modeling. In *Software Engineering (SE)*, pages 77–78. GI, 2020.

20 Sebastian Nielebock, Robert Heumüller, **Jacob Krüger**, and Frank Ortmeier. Using API-Embedding for API-Misuse Repair. In *International Conference on Software Engineering Workshops (ICSEW)*, pages 1–2. ACM, 2020a.

21 Sebastian Nielebock, Robert Heumüller, **Jacob Krüger**, and Frank Ortmeier. Cooperative API Misuse Detection Using Correction Rules. In *International Conference on Software Engineering (ICSE)*, pages 73–76. ACM, 2020b.

22 Cem Sürücü, Bianying Song, **Jacob Krüger**, Gunter Saake, and Thomas Leich. Establishing Key Performance Indicators for Measuring Software-Development Processes at a Large Organization. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1331–1341. ACM, 2020.

23 Jacob Krüger, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. Program Comprehension and Developers' Memory. In *INFORMATIK*, pages 99–100. GI, 2019.

24 **Jacob Krüger**, Mustafa Al-Hajjaji, Thomas Leich, and Gunter Saake. Mutation Operators for Feature-Oriented Software Product Lines. In *International Systems and Software Product Line Conference (SPLC)*, pages 12–12. ACM, 2019a.

25 **Jacob Krüger**, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. Understanding How Programmers Forget. In *Software Engineering and Software Management (SE/SWM)*, pages 85–86. GI, 2019b.

26 Elias Kuiter, Sebastian Krieter, **Jacob Krüger**, Thomas Leich, and Gunter Saake. Foundations of Collaborative, Real-Time Feature Modeling. In *International Systems and Software Product Line Conference (SPLC)*, pages 257–264. ACM, 2019.

27 Michael Nieke, Lukas Linsbauer, **Jacob Krüger**, and Thomas Leich. Second International Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution 2019). In *International Systems and Software Product Line Conference (SPLC)*, pages 320–320. ACM, 2019.

28 Yusra Shakeel, **Jacob Krüger**, Gunter Saake, and Thomas Leich. Indicating Studies' Quality Based on Open Data in Digital Libraries. In *International Conference on Business Information Systems (BIS)*, pages 579–590. Springer, 2019.

29 Gabriel C. Durand, Anusha Janardhana, Marcus Pinnecke, Yusra Shakeel, **Jacob Krüger**, Thomas Leich, and Gunter Saake. Exploring Large Scholarly Networks with Hermes. In *International Conference on Extending Database Technology (EDBT)*, pages 650–653. OpenProceedings, 2018.

30 Sebastian Krieter, **Jacob Krüger**, Nico Weichbrodt, Vasily A. Sartakov, Rüdiger Kapitza, and Thomas Leich. Towards Secure Dynamic Product Lines in the Cloud. In *International Conference on Software Engineering (ICSE)*, pages 5–8. ACM, 2018.

31 **Jacob Krüger**, Mustafa Al-Hajjaji, Sandro Schulze, Gunter Saake, and Thomas Leich. Towards Automated Test Refactoring for Software Product Lines. In *International Systems and Software Product Line Conference (SPLC)*, pages 143–148. ACM, 2018.

32 Elias Kuiter, Sebastian Krieter, **Jacob Krüger**, Kai Ludwig, Thomas Leich, and Gunter Saake. PCLocator: A Tool Suite to Automatically Identify Configurations for Code Locations. In *International Systems and Software Product Line Conference (SPLC)*, pages 284–288. ACM, 2018.

33 Yusra Shakeel, **Jacob Krüger**, Ivonne von Nostitz-Wallwitz, Christian Lausberger, Gabriel C. Durand, Gunter Saake, and Thomas Leich. (Automated) Literature Analysis - Threats and Experiences. In *International Workshop on Software Engineering for Science (SE4Science)*, pages 20–27. ACM, 2018.

34 Ivonne von Nostitz-Wallwitz, **Jacob Krüger**, and Thomas Leich. Towards Improving Industrial Adoption: The Choice of Programming Languages and Development Environments. In *International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, pages 10–17. ACM, 2018a.

35 Ivonne von Nostitz-Wallwitz, **Jacob Krüger**, Janet Siegmund, and Thomas Leich. Knowledge Transfer from Research to Industry: A Survey on Program Comprehension. In *International Conference on Software Engineering (ICSE)*, pages 300–301. ACM, 2018b.

36 Mustafa Al-Hajjaji, **Jacob Krüger**, Fabian Benduhn, Thomas Leich, and Gunter Saake. Efficient Mutation Testing in Configurable Systems. In *International Workshop on Variability and Complexity in Software Design (VACE)*, pages 2–8. IEEE, 2017a.

37 Mustafa Al-Hajjaji, **Jacob Krüger**, Sandro Schulze, Thomas Leich, and Gunter Saake. Efficient Product-Line Testing Using Cluster-Based Product Prioritization. In *International Workshop on Automation of Software Testing (AST)*, pages 16–22. IEEE, 2017b.

38 Sebastian Krieter, Marcus Pinnecke, **Jacob Krüger**, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. FeatureIDE: Empowering Third-Party Developers. In *International Systems and Software Product Line Conference (SPLC)*, pages 42–45. ACM, 2017.

39 **Jacob Krüger**, Niklas Corr, Ivonne Schröter, and Thomas Leich. Digging into the Eclipse Marketplace. In *International Conference on Open Source Systems (OSS)*, pages 60–65. Springer, 2017a.

40 **Jacob Krüger**, Stephan Dassow, Karl-Albert Bebber, and Thomas Leich. Daedalus or Icarus? Experiences on Follow-the-Sun. In *International Conference on Global Software Engineering (ICGSE)*, pages 31–35. IEEE, 2017b.

41 **Jacob Krüger**, Sebastian Nielebock, Sebastian Krieter, Christian Diedrich, Thomas Leich, Gunter Saake, Sebastian Zug, and Frank Ortmeier. Beyond Software Product Lines: Variability Modeling in Cyber-Physical Systems. In *International Systems and Software Product Line Conference (SPLC)*, pages 237–241. ACM, 2017c.

42 **Jacob Krüger**, Ivonne Schröter, Andy Kenner, and Thomas Leich. Empirical Studies in Question-Answering Systems: A Discussion. In *International Workshop on Conducting Empirical Studies in Industry (CESI)*, pages 23–26. IEEE, 2017d.

43 Ivonne Schröter, **Jacob Krüger**, Philipp Ludwig, Marcus Thiel, Andreas Nürnberger, and Thomas Leich. Identifying Innovative Documents: Quo Vadis? In *International Conference on Enterprise Information Systems (ICEIS)*, pages 653–658. SCITEPRESS, 2017a.

44 Ivonne Schröter, **Jacob Krüger**, Janet Siegmund, and Thomas Leich. Comprehending Studies on Program Comprehension. In *International Conference on Program Comprehension (ICPC)*, pages 308–311. IEEE, 2017b.

Contents

List of Figures

List of Tables

1. Introduction

This chapter builds on publications at ICSE [Krüger, 2017, 2018a], ESEC/FSE [Krüger, 2019a; Krüger and Berger, 2020b], SAC [Krüger, 2018b], and VaMoS [Krüger et al., 2020b].

Variant-rich systems

Most modern software systems are variant-rich systems; systems that exist in different variants to fulfill varying customer requirements, hardware limitations, regulations, or other restrictions. Consequently, most organizations have to develop and maintain a portfolio of similar, yet customized variants, also referred to as a *product family* [Apel et al., 2013a; Berger et al., 2020; Fenske et al., 2013; Pohl et al., 2005; van der Linden et al., 2004, 2007]. Such variant-rich systems are common for industrial and open-source systems in any domain, including, for instance, the Linux Kernel, operating systems (e.g., Windows, Linux distributions), firmware (e.g., Marlin 3D printers, HP printers), webservers (e.g., Apache), program interpreters (e.g., GCC, Python), Android applications, databases (e.g., Oracle Berkeley DB), automotive systems (e.g., Volvo, Opel, Rolls Royce), embedded systems (e.g., Hitachi engine control, Axis cameras, Keba robotics, Danfoss frequency converters), or IoT- and web-services (e.g., at Google) [Abbas et al., 2020; Bauer et al., 2014; Berger et al., 2015, 2020; Birk et al., 2003; Bosch and Bosch-Sijtsema, 2010; Businge et al., 2018; Fogdal et al., 2016; Ham and Lim, 2019; Krüger and Berger, 2020b; Krüger et al., 2018a,b; Kuiter et al., 2018b; Liebig et al., 2010, 2011; Nolan and Abrahão, 2010; Stănciulescu et al., 2015; van der Linden, 2002; van der Linden et al., 2007; Yoshimura et al., 2006b]. The main challenge of establishing such a variant-rich system is to select a strategy for reusing existing artifacts.

Software reuse

Software reuse is one of the most fundamental concepts to decrease the development costs, improve the quality, and reduce the time-to-market of a new variant [Krueger, 1992; Krüger and Berger, 2020b; Long, 2001; Mili et al., 2000; Standish, 1984; van der Linden et al., 2007]. The basic idea is to reuse existing artifacts (e.g., code, models, requirements) that represent user-visible functionalities — called *features* [Apel et al., 2013a; Berger et al., 2015; Classen et al., 2008] — and integrate these into a new variant. While various reuse techniques with individual pros and cons are used in practice, they can be distinguished based on certain properties, for example, whether artifact reuse is planned or pragmatic [Holmes and Walker, 2012; Kulkarni and Varma, 2016; Tomer et al., 2004]. We distinguish two reuse strategies:

Clone & own

Clone & own (also called *ad hoc*, *opportunistic*, *pragmatic*, and *scavenging* reuse or *cloning in the large*) refers to developers creating a copy of a variant that they adopt to new customer requirements [Dubinsky et al., 2013; Rubin et al., 2015; Stănciulescu et al., 2015]. Using clone & own is cheap, readily available, and supported by many version-control systems (e.g., branching in Git) [Gousios et al., 2014; Krüger and Berger, 2020b; Krüger et al., 2019c; Lillack et al., 2019; Stănciulescu et al., 2015]. For this reason, most organizations and developers initially employ clone & own, until an increasing number of cloned variants challenges maintenance, for instance, because features and bug fixes must be propagated or developers loose their overview understanding of which cloned variants comprise which features [Bauer and Vetrò, 2016; Berger et al., 2020; Krüger, 2019b; Kuiter et al., 2018b; Long, 2001; Pfofe et al., 2016; Rubin et al., 2015; Yoshimura et al., 2006b]. We remark that clone & own can be considered to represent *copy & paste* (or *cloning in the small*) [Bellon et al., 2007; Cheung et al., 2016; Li et al., 2006; Narasimhan and Reichenbach, 2015; Roy et al., 2009], but it is employed on a larger scale and causes more severe consequences considering an organization's development processes and infrastructure, for instance, to enable developers to efficiently manage the cloned variants with tools [Fischer et al., 2014; Lillack et al., 2019; Linsbauer, 2016; Pfofe et al., 2016; Rubin et al., 2012, 2013].

Platform

Platform engineering (also called *systematic*, *[pre-]planned*, and *strategic* reuse) refers to developers engineering a single codebase from which they can derive customized variants [Meyer and Lehnerd, 1997]. This strategy usually involves concepts and methods from software product-line engineering [Apel et al., 2013a; Clements and Northrop, 2001; Pohl et al., 2005; van der Linden et al., 2007], employing a variability mechanism to implement variation points (e.g., using preprocessor macros, runtime parameters, or components) in the codebase to control features [Apel et al., 2013a; Gacek and Anastasopoules, 2001; Schaefer et al., 2012; Svahnberg et al., 2005], variability models [Berger et al., 2013a; Chen and Babar, 2011; Czarnecki et al., 2012; Nešić et al., 2019; Schaefer et al., 2012], as well as tools to configure and automatically derive a variant [Beuche, 2012; Horcas et al., 2019; Krueger, 2007; Meinicke et al., 2017; Thüm et al., 2014b]. A platform requires investments into technical (e.g., the platform architecture) and non-technical (e.g., development processes) aspects, before its benefits can be achieved, for instance, decreased maintenance and development costs as well as substantially reduced time-to-market [Böckle et al., 2004b; Bosch and Bosch-Sijtsema, 2010; Clements and Krueger, 2002; Krüger and Berger, 2020a,b; Krüger et al., 2016a; Lindohf et al., 2021; Schmid and Verlage, 2002; van der Linden, 2005; van der Linden et al., 2004, 2007].

Re-engineering variant-rich systems

Initially, most variant-rich systems are engineered based on clone & own, and later transformed into a platform [Berger et al., 2013a, 2020; Duszynski et al., 2011; Krüger, 2019b; Kuiter et al., 2018b; Long, 2001; Pohl et al., 2005; Yoshimura et al., 2006b]. The subsequent evolution of *re-engineering* cloned variants into a platform remains a common practical problem — considering its costs and risks, despite decades of research aiming to facilitate this process [Assunção et al., 2017; Berger et al., 2020; Clements and Krueger, 2002; Krueger, 2002; Krüger, 2019b; Krüger et al., 2016a; Laguna and Crespo, 2013; Schmid and Verlage, 2002]. Just as one example, Fogdal et al. [2016] report that introducing a platform based on re-engineering cloned variants at Danfoss took 36 months for 80 % of the code, instead of the intended 10 months for the whole re-engineering (also involving, e.g., tools and variability models). We remark that we rely on the term re-engineering in this dissertation, while other researchers have defined synonymous terms, for example, Krueger [2002] refers to *extractive product-line adoption* and Fenske et al. [2013] refer to *variant-preserving migration*.

1.1 Goals and Contributions

Scope

As described, clone & own and platform engineering require developers to adopt different tools, processes, and techniques to reuse artifacts and customize variants effectively. Choos-

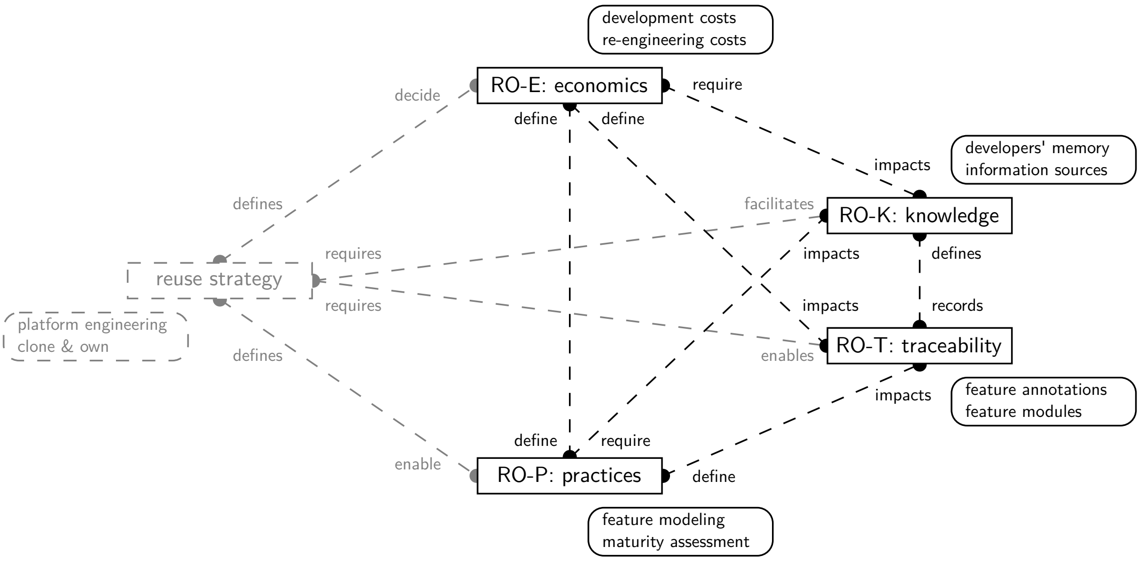


Figure 1.1: Conceptual framework of our research objectives and their relations to each other as well as to the reuse strategy employed (in gray). The boxes with rounded corners exemplify concrete instances related to the concepts.

ing either strategy is an important strategical decision that defines an organization's ability to engineer a variant-rich system. Unfortunately, researchers and practitioners are missing a detailed understanding and reliable data on various properties of these strategies that would help to decide which strategy to use, how to steer re-engineering projects, and how to facilitate the conduct of such projects [Krüger and Berger, 2020a,b]. In this dissertation, we are concerned with improving this understanding, which is why our scope is mainly on *knowledge creation* and *verification*, building on evidence-based software engineering [Dybå et al., 2005; Kitchenham et al., 2004, 2015; Shull et al., 2008]. Our results help practitioners to decide for a reuse strategy as well as to scope and execute corresponding projects, and researchers to design new techniques that are in line with real-world needs. To obtain these results, we used empirical research methods, such as surveys, experiments, and systematic literature reviews, to synthesize and strengthen the understanding of four properties (explained below) — that we found to be closely related, but miss detailed analyses: economics, knowledge, traceability, and practices [Krüger, 2017, 2018a,b, 2019a; Krüger and Berger, 2020b; Krüger et al., 2020b].

Research objectives

To investigate these four properties, we defined a respective research objective (RO) for each. Despite extensive research on software reuse and variant-rich systems [Assunção et al., 2017; C and Chandrasekaran, 2017; Heradio et al., 2016; Krueger, 1992; Laguna and Crespo, 2013; Rabiser et al., 2018; Schaefer et al., 2012], recent reviews, experience reports, and surveys agree that these properties in particular are still not well understood. In the following, we briefly motivate our research objectives and their relations, for which we sketch a coarse-grained conceptual framework in Figure 1.1 — in which we also indicate the relations to the employed reuse strategy in gray. We provide a more detailed framework and description of each objective in the corresponding chapters. Note that the central element that is common in our fine-grained conceptual frameworks are the developers, who we aim to support with all of our research objectives.

Economics

Understanding the economics (i.e., costs and benefits) of (re-)engineering variant-rich systems is essential to support an organization's decision making regarding which reuse strategy to use and what traceability techniques as well as practices to implement. For instance, cost models and scoping techniques for software product lines [Ali et al., 2009; Bayer et al., 1999; Koziolek et al., 2016; Nolan and Abrahão, 2010; Rincón et al., 2018; Thurimella and Padmaja, 2014; van der Linden, 2005] have been investigated and used for a long time, but are often based on single experiences instead of systematically elicited data [Krüger, 2016; Krüger and Berger, 2020a,b; Lindohf et al., 2021]. Consequently, it is not surprising that the most common reasoning on adopting a platform is still the experience-based rule-of-thumb that it will pay off after developing three variants [Knauber et al., 2002; van der Linden et al., 2007]. For this reason, the need to better understand the economics of (re-)engineering variant-rich systems is regularly mentioned as a an open problem in literature reviews, for example, by Laguna and Crespo [2013] or Assunção et al. [2017]. Similarly, surveys with practitioners repeatedly highlight the expenses associated with variant-rich systems and organizations' demands for improving decision support based on reliable economic data [Berger et al., 2020; Ghanam et al., 2012]. Apparently, this long-standing, highly important research problem has not been well investigated, neither by research nor by industry [Rabiser et al., 2019], arguably due to the challenges of understanding economics in software engineering and systematically eliciting datasets [Boehm, 1984; Bosu and Macdonell, 2019; Heemstra, 1992; Jørgensen, 2014; Mustafa and Osman, 2020]. In this dissertation, we shed light into such economics based on the following research objective:

Economics (Chapter 3)

RO-E *Establish a body-of-knowledge on the economics of clone & own, platform engineering, and the re-engineering of cloned variants into a platform.*

Knowledge

Numerous factors (e.g., of the system, developers, practices) impact the economics of (re-)engineering variant-rich systems. However, most costs in software engineering and particularly re-engineering can be attributed to developers' knowledge needs and program comprehension — consequently defining the extent of traceability and what practices for managing a variant-rich system are required. Systems evolve, developers forget over time [Kang and Hahn, 2009; Krüger et al., 2018e], and features are usually not traced properly in the code, which is why developers must comprehend the code again and perform feature location [Andam et al., 2017; Assunção and Vergilio, 2014; Krüger, 2019a; Krüger and Berger, 2020b; Krüger et al., 2019a; Tiarks, 2011; Wang et al., 2013; Wilde et al., 2003]; which are human-centered and expensive activities, due to the concept-assignment problem [Biggerstaff et al., 1993]. It is not surprising that literature reviews of automated feature-location techniques (and other techniques for reverse engineering knowledge) highlight the expensive adaptations required and missing evaluations [Alves et al., 2010; Bakar et al., 2015; Dit et al., 2013; Razzaq et al., 2018; Rubin and Chechik, 2013b], making it hard to judge the usability of such techniques. Moreover, studies in industry indicate that developers can, and have to, improve their knowledge regarding additional properties (e.g., quality attributes) to understand the features, similarities, and differences of variant-rich systems [Berger et al., 2015; Ghanam et al., 2012; Tang et al., 2010]. For these reasons, we focus on human factors (i.e., developers' knowledge) of re-engineering in this dissertation, which motivate, but are actually not well investigated compared to, reverse-engineering techniques [Krüger et al., 2019a; Rabiser et al., 2018]. So, we defined the following research objective:

Knowledge (Chapter 4)

RO-K *Provide an understanding of the importance and recovery of developers' knowledge for enabling the re-engineering of cloned variants into a platform.*

Traceability

Software traceability can cover various artifacts (e.g., requirements, features, tests) and dimensions (e.g., development activities, system evolution), intending to facilitate practices, and thus reduce costs, by encoding knowledge into a system and allowing tools to automatically analyze such traces [Andam et al., 2017; Antoniol et al., 2017; Ji et al., 2015; Krüger, 2019a; Krüger et al., 2019b; Nair et al., 2013; Vale et al., 2017]. We focus on feature traceability in source code, which helps developers to recover their knowledge (e.g., easing feature location), to establish a knowledge base regarding what features are implemented where in a variant-rich system, and to document the evolution of features. In particular, while optional (i.e., configurable) features in a platform are often traceable based on the used variability mechanism (e.g., preprocessor annotations), this is rarely the case for mandatory features or in cloned variants. Introducing feature traceability (e.g., while re-engineering cloned variants) is challenging and not well understood, as highlighted by literature reviews [Assunção et al., 2017; Laguna and Crespo, 2013] and industrial investigations [Berger et al., 2020]. Most importantly, Vale et al. [2017] identify several open challenges that we are concerned with, namely, applying feature traceability in practice, strengthening the empirical evidence in this regard, and understanding the differences between variability mechanisms and traceability. To tackle these challenges, we defined the following research objective:

Traceability (Chapter 5)

RO-T *Improve the evidence on how to implement feature traceability in a variant-rich system and during the re-engineering of cloned variants into a platform.*

Practices

While addressing our previous objectives, we investigated the practices (e.g., activities, techniques, tools) employed for each reuse strategy, and how these define the economics, traceability, as well as knowledge required for (re-)engineering variant-rich systems. Interestingly, particularly the software product-line community still builds heavily on process models established around two decades ago that strictly separate domain and application engineering [Apel et al., 2013a; Northrop, 2002; Pohl et al., 2005; van der Linden et al., 2007]. We found that these do neither incorporate current practices in industry nor advancements in research [Krüger and Berger, 2020a,b; Krüger et al., 2020d]. Similarly, Rabiser et al. [2018] and Berger et al. [2020] argue that process topics have not been well investigated by research recently and require an update to better support industry. An updated process model and analysis of current practices is important to support not only organizations, but also researchers to identify and scope open directions based on how variant-rich systems are engineered with modern technologies — and to understand how costs, knowledge, and traceability are connected to modern round-trip engineering practices employed for (re-)engineering variant-rich systems. So, we defined our last research objective as follows:

Practices (Chapter 6)

RO-P *Define a practice-oriented round-trip engineering process model for implementing, re-engineering, and evolving variant-rich systems.*

Dissertation contributions

By addressing our research objectives, we created and verified knowledge that helps researchers and practitioners alike. In particular, our overarching goal for practitioners is to support the decision making of organizations regarding whether they can benefit from (re-)engineering a platform, how to do this, and how to tackle the associated problems. For research, our results provide insights into open problems that require further analyses and novel techniques. In a nutshell, our contributions represent an empirics-based work that extends the existing body-of-knowledge on:

C$_1$ The costs of clone & own, platform engineering, and the re-engineering of cloned variants into a platform (Chapter 3).

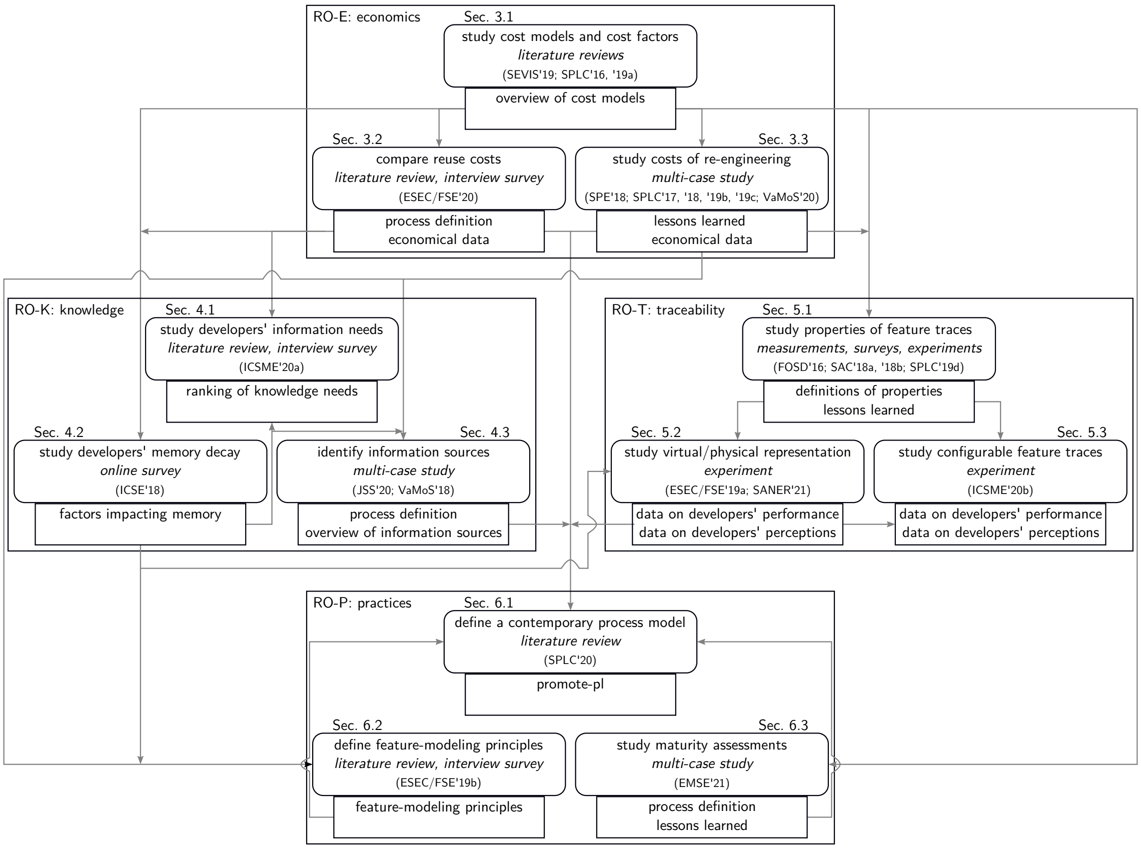

RO-E: economics
Sec. 3.1
study cost models and cost factors
literature reviews
(SEVIS'19; SPLC'16, '19a)
overview of cost models
Sec. 3.2
compare reuse costs
literature review, interview survey
(ESEC/FSE'20)
process definition
economical data
Sec. 3.3
study costs of re-engineering
multi-case study
(SPE'18; SPLC'17, '18, '19b, '19c; VaMoS'20)
lessons learned
economical data
RO-K: knowledge
Sec. 4.1
study developers' information needs
literature review, interview survey
(ICSME'20a)
ranking of knowledge needs
Sec. 4.2
study developers' memory decay
online survey
(ICSE'18)
factors impacting memory
Sec. 4.3
identify information sources
multi-case study
(JSS'20; VaMoS'18)
process definition
overview of information sources
RO-T: traceability
Sec. 5.1
study properties of feature traces
measurements, surveys, experiments
(FOSD'16; SAC'18a, '18b; SPLC'19d)
definitions of properties
lessons learned
Sec. 5.2
study virtual/physical representation
experiment
(ESEC/FSE'19a; SANER'21)
data on developers' performance
data on developers' perceptions
Sec. 5.3
study configurable feature traces
experiment
(ICSME'20b)
data on developers' performance
data on developers' perceptions
RO-P: practices
Sec. 6.1
define a contemporary process model
literature review
(SPLC'20)
promote-pl
Sec. 6.2
define feature-modeling principles
literature review, interview survey
(ESEC/FSE'19b)
feature-modeling principles
Sec. 6.3
study maturity assessments
multi-case study
(EMSE'21)
process definition
lessons learned

Figure 1.2: Overview of the goals of our studies, the methodologies (italic), key publications (in parentheses), results (square rectangles), corresponding sections in this dissertation, and (roughly) which results inspired, or contributed to, which other studies (gray arrows).

C$_2$ Developers' knowledge (Chapter 4) and feature traceability (Chapter 5) in the context of reusing, re-engineering, and evolving software.

C$_3$ Processes and recommendations for (re-)engineering variant-rich systems that involve modern software-engineering practices (Chapter 6).

As described, these contributions help to solve several long-standing research and industrial problems. We provide a more detailed discussion of each objective and the consequent contributions in the corresponding chapters. On a final note, we want to emphasize that our results are related to numerous research fields and provide insights that exceed the scope of this dissertation (e.g., on the importance of developers' memory for program comprehension).

1.2 Structure

Research overview

In Figure 1.2, we sketch the overall structure of our core chapters. We show our research objective for each section, what methods we used to address it, our key publications (alphabetical order; we show a detailed list at the beginning of each chapter), and our results. Moreover, we use the gray arrows to indicate which results motivated which studies. Note that this is a coarse overview focusing on the most prominent relations. Also, we remark that the publications seem to be out of order, but this is mostly due to the time it took to publish some of them — and because we worked on several studies in parallel based on intermediate results. For instance, our economical comparison of clone & own and platform engineering [Krüger and Berger, 2020b] took several years, while early insights confirmed that knowledge is an important cost factor — partly inspiring our study on developers' memory [Krüger et al., 2018e]. Consequently, we did **not** structure this dissertation solely based on the order of publications, but focused on improving the comprehensibility of our research and the connections therein. Still, we can see in Figure 1.2 that all of our research originates from an economical perspective, that we studied knowledge and traceability largely in parallel (they are closely connected, and thus motivated by similar results), and integrated all of our findings into practices.

Structure

More detailed, this dissertation is structured as follows:

In Chapter 2, we introduce concepts of variant-rich systems that are needed to understand this dissertation. Note that we aimed to improve the comprehensibility of this dissertation by refining our conceptual framework and introducing more specific background knowledge in the individual chapters.

In Chapter 3, we report our research on the economics of variant-rich systems. To this end, we first provide an overview of existing cost models and feature location as a major cost factor for re-engineering variant-rich systems. Then, we present our empirical comparison of developing variants via clone & own and platform engineering — confirming and refuting established hypotheses. Finally, we synthesize our lessons learned and data from five re-engineering case studies. Put very concisely, our results indicate that organizations should strive towards more systematic software reuse.

In Chapter 4, we describe our research on developers' knowledge. To this end, we report an empirical study in which we investigated developers' information needs and their motivation to remember different types of knowledge. Afterwards, we present our survey on developers' memory performance regarding their source code. Since our insights show that developers have to recover knowledge particularly for the code level, we show which information sources of software-hosting platforms (i.e., GitHub) can help recover the required knowledge on features (e.g., their locations) . Put very concisely, our results indicate that documenting and tracing knowledge is key to reduce the costs of (re-)engineering variant-rich systems.

In Chapter 5, we present our research on feature traceability. To this end, we define three dimensions of feature traces and their potential trade offs based on different types of studies. This inspired two experiments on program comprehension: First we report an experiment on the impact of different feature representations (i.e., feature annotations and feature modules) on developers' task performance and memory. Second, we present our results regarding the use of a variability mechanism (i.e., the C preprocessor) for feature traceability. Put very concisely, our results indicate that feature traceability supports program comprehension by facilitating feature location, and should ideally be separated from variability mechanisms.

In Chapter 6, we explain our research on contemporary practices for variant-rich systems. To this end, we detail a novel process model for (re-)engineering variant-rich systems that we constructed based on recent experience reports and our previous findings. We enrich our process model by providing a set of 34 feature-modeling principles, highlighting that they relate repeatedly to our other research objectives. Finally, we report our experiences of assessing the maturity of variant-rich systems, which allows to monitor the system's evolution and plan investments. Put very concisely, our results indicate that the processes and recommendations we constructed can help to systematically manage a variant-rich system, and thus to increase its value.

In Chapter 7, we conclude this dissertation by summarizing our core findings and discussing directions for future work.

Please note that we do **not** provide a separate related-work chapter. We built extensively on systematic literature reviews to synthesize knowledge reported in the related work and typically involved similar literature reviews in the process, too (e.g., to define a starting set for snowballing or to assess the completeness of our samples). Consequently, we involve the related work directly in our actual research and the corresponding chapters, which is why we decided against reporting it separately.

2. Variant-Rich Systems

This chapter builds on publications at ESEC/FSE [Nešić et al., 2019], SPLC [Krüger et al., 2020d; Strüber et al., 2019] and Empirical Software Engineering [Lindohf et al., 2021].

Variant-rich systems

A variant-rich system is a portfolio of similar variants that share common code, but include also customized features for specific customers [Berger et al., 2020; Krüger et al., 2020b; Strüber et al., 2019; Villela et al., 2014]. So, while a variant-rich system allows to customize variants to different requirements, the high similarity between the variants promotes software reuse to improve software quality and reduce costs [Krueger, 1992; Krüger and Berger, 2020b; Long, 2001; Mili et al., 2000; Standish, 1984; van der Linden et al., 2007]. In fact, many variant-rich systems originate from developers cloning (i.e., reusing) a software system and adapting it to new requirements. There are numerous different techniques [Frakes and Kang, 2005; Krueger, 1992] for reusing software, and thus to implement a variant-rich system. Adapting the established distinction of pragmatic and planned reuse [Holmes and Walker, 2012; Kulkarni and Varma, 2016; Tomer et al., 2004], we distinguish two main strategies: clone & own (pragmatic) and platform engineering (planned). In this chapter, we first introduce these two strategies in Section 2.1 and Section 2.2, respectively. Finally, we describe concepts and methods of software product-line engineering that are typically used to systematically manage a platform in Section 2.3.

2.1 Clone & Own

Clone & own

Clone & own is a simple and readily available reuse strategy that builds on cloning an existing variant and customizing that clone to new customer requirements [Dubinsky et al., 2013; Rubin et al., 2015; Stănciulescu et al., 2015]. This cloning mechanism is also well-supported by existing version-control systems (e.g., Git) and software-hosting platforms (e.g., GitHub) based on their branching and forking mechanisms [Fischer et al., 2014; Gousios et al., 2014; Krüger and Berger, 2020b; Krüger et al., 2019c; Lillack et al., 2019; Stănciulescu et al., 2015]. To exemplify this reuse strategy, we display a conceptual sketch in Figure 2.1. We can see that each variant (except for the first one) is created by cloning (e.g., forking) another variant. Over time, developers evolve each variant individually, for instance, by introducing new features, bug fixes, or performance improvements. Such changes can be propagated to other variants by synchronizing them [Strüber et al., 2019], and researchers have proposed several techniques to help developers manage and synchronize cloned variants [Fischer et al., 2014; Pfofe et al., 2016; Rubin and Chechik, 2013a; Rubin et al., 2012, 2013].

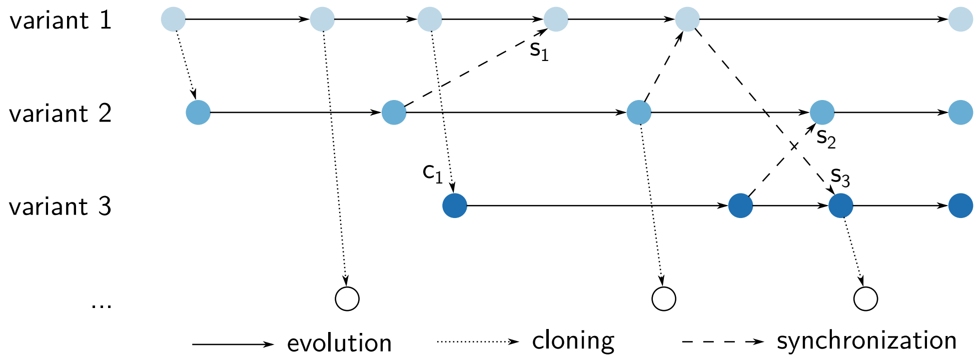


Figure 2.1: Conceptual sketch of clone & own-based variant development.

Pros of clone & own

Clone & own is a viable reuse strategy that promises several benefits besides software reuse itself [Fischer et al., 2014; Jansen et al., 2008; Rubin et al., 2012]. For instance, developers require no preparation or advanced tooling to use clone & own. On the contrary, clone & own is a cost-efficient reuse strategy for delivering new variants to customers without the need to invest into a common codebase. So, it is not surprising that many organizations start to reuse their existing variants by cloning a variant and adapting it to customers' individual feature requests—avoiding costs and allowing them to fulfill strict deadlines. Moreover, clone & own offers independence by keeping variants separated (i.e., not synchronizing them), which allows to assign them more easily to developers and keep different customer features strictly apart.

Cons of clone & own

Despite its benefits, industrial practice shows that clone & own is often insufficient to manage an ever-increasing number of variants [Bauer and Vetrò, 2016; Berger et al., 2020; Krüger, 2019b; Kuiter et al., 2018b; Long, 2001; Pfofe et al., 2016; Rubin et al., 2015; Yoshimura et al., 2006b]. Particularly, it becomes more and more challenging to keep an overview understanding of which features are implemented in which variants and to synchronize features, bug fixes, or improvements. For instance, the first variant in Figure 2.1 is cloned (c_1) for a second time right before it is synchronized (s_1). Consequently, the third variant may miss some important updates. Similarly, the later synchronizations (s_2, s_3) may be expected to align the second and third variant. However, the third synchronization (s_3) includes additional changes from the first variant, which may cause different side effects in the third variant compared to the second one. This highlights that, even if variants are synchronized, developers must inspect and test all changes in each variant individually to identify potential faults. Due to such problems, many organizations re-engineer their cloned variants towards a software platform at some point.

2.2 Software Platforms

Software platforms

A software platform represents a set of common artifacts from which a portfolio of variants can be derived [Meyer and Lehnerd, 1997]. Developing a platform requires more investments than developing a single variant, but the faster time-to-market, reduced development as well as maintenance costs, and improved software quality usually pay off in the long run [Böckle et al., 2004b; Bosch and Bosch-Sijtsema, 2010; Clements and Krueger, 2002; Krüger and Berger, 2020a,b; Krüger et al., 2016a; Lindohf et al., 2021; Schmid and Verlage, 2002; van der Linden, 2005; van der Linden et al., 2004, 2007]. Moreover, a platform enables an organization to manage its software more systematically, and to coordinate its development around smaller increments (e.g., feature teams [Ghanam et al., 2012; Holmström Olsson et al., 2012]). Thus, platforms have become a widely established practice in software engineering, facilitating or enabling various other methodologies, for instance, agile and continuous software engineering [Bosch, 2014; Meyer, 2014; Moran, 2015], software-ecosystem engineering [Bosch and Bosch-Sijtsema, 2010; Lettner et al., 2014], or software product-line engineering [Apel et al., 2013a; Pohl et al., 2005; van der Linden et al., 2007]. While such methodologies employ platforms with different ideas in mind, they are often closely related and build on similar to identical concepts to develop and manage the platform itself.

Feature

Platform engineering is usually structured around the notion of features, which are an established concept in software engineering with an intuitive meaning to most developers. For this reason, unfortunately, a multitude of definitions for the term feature exist [Apel and Kästner, 2009a; Apel et al., 2013a; Berger et al., 2015; Classen et al., 2008], spanning from high-level domain abstractions down to implementation-level concepts. In this dissertation, we rely on the definition of Apel et al. [2013a] to capture features as domain abstractions:

> *"A feature is a characteristic or end-user-visible behavior of a software system."*

Note that this definition does not enforce that a feature is optional (i.e., representing a functionality that can be enabled or disabled), which is often explicitly defined in the software product-line community. Instead, a feature is a unit of functionality, which also captures mandatory features (i.e., those that are part of every variant). We rely on this notion of features as units of functionality [Berger et al., 2015; Krüger et al., 2019c], mainly for three reasons:

1. To recognize that mandatory features are an essential part of any variant-rich system and provide most of the cost savings of reusing software.
2. To represent that mandatory features must also be developed and maintained, which is why they should be systematically managed, too.
3. To reflect that most software-engineering methodologies (e.g., for agile software engineering) follow this notion (e.g., in the product backlog [Sedano et al., 2019]).

So, features can serve as an abstraction for specifying, documenting, comprehending, managing, and reusing any functionality of a platform (or any other variant-rich system).

Asset

According to our definition, features are domain abstractions belonging to the so-called problem space [Apel and Kästner, 2009a; Apel et al., 2013a; Czarnecki, 2005]. To use the platform, features must be implemented — mapping them to the solution space. We refer to the actual implementations of a feature as assets [Northrop, 2002; Pohl et al., 2005]. Note that asset may refer to various types of artifacts, such as source code, models, documentation, or tests.

2.3 Software Product-Line Engineering

Software product-line engineering

Software product-line engineering [Apel et al., 2013a; Czarnecki, 2005; Northrop, 2002; Pohl et al., 2005; van der Linden et al., 2007] defines methods and tools to systematically engineer and manage a software platform, allowing organizations to mass-customize (i.e., reuse and configure) variants. In the following, we discuss the three adoption strategies (Section 2.3.1) and two engineering processes (Section 2.3.2) that have been defined by the software product-line community. Then, we describe individual concepts used to engineer a platform systematically: variability models (Section 2.3.3), variability mechanisms (Section 2.3.4), and configuring (Section 2.3.5).

2.3.1 Adopting a Platform

Adoption strategies

The software product-line community distinguishes three core strategies for adopting a platform [Apel et al., 2013a; Berger et al., 2013a; Clements and Krueger, 2002; Fenske

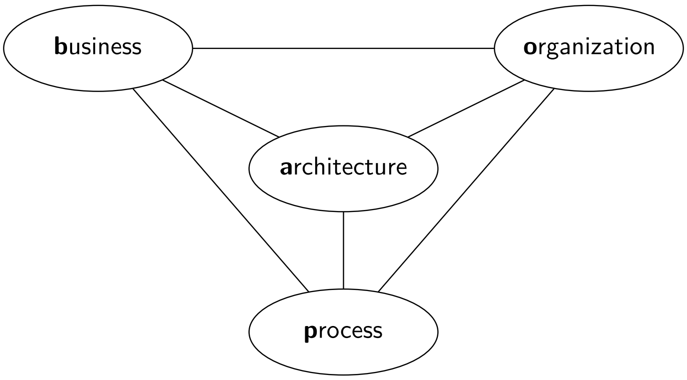


Figure 2.2: Overview of the BAPO concerns based on van der Linden et al. [2004].

et al., 2013; Krueger, 2002; Schmid and Verlage, 2002]: proactive, incremental, and re-engineering. Note that, while we rely on these rather intuitive terms, the terminology varies within the software product-line community (e.g., extractive [Krueger, 2002], variant-preserving mapping [Fenske et al., 2013]). Precisely, each adoption strategy is defined as follows (combinations are possible):

Proactive: In this strategy, an organization basically starts from scratch by planning and engineering a completely new platform. Consequently, this strategy requires high investments in the beginning, but it also promises the most long-term savings and a fast break-even point (i.e., an established rule-of-thumb is after three variants [Knauber et al., 2002; van der Linden et al., 2007]). Nonetheless, a proactive adoption is a risky, idealized, and rather academic strategy that rarely occurs in its pure form in practice.

Incremental: In this strategy, an organization develops a single variant and incrementally advances it into a platform by adding new features. As a result, this strategy is less expensive compared to a proactive adoption and follows a more agile methodology.

Re-engineering: In this strategy, an organization already has a set of similar variants, for instance, a variant-rich system that is developed via clone & own. While this strategy poses the least risks (i.e., the variants are already established in the market), it also requires additional investments for re-engineering that may not pay off in the future.

We focus on the re-engineering of cloned variants towards a platform, since experience reports and research indicate that this is the most relevant strategy in practice [Berger et al., 2013a, 2020; Duszynski et al., 2011; Fogdal et al., 2016; Krüger, 2019b; Kuiter et al., 2018b; Long, 2001; Pohl et al., 2005; Yoshimura et al., 2006b]. Still, many of our findings are just as relevant for other adoption strategies.

BAPO model To adopt a platform and the corresponding software product-line concepts, most organizations have to implement substantial changes regarding four dimensions. These dimensions, which we display in Figure 2.2, are known as the BAPO concerns [van der Linden, 2002, 2005; van der Linden et al., 2004, 2007] and have been elicited during industry-academia collaborations. In detail, the concerns cover:

Business: A platform requires different financial and strategical planning compared to individual projects. For example, organizations that previously developed and sold individual systems have to plan the platform funding, which promises long-term benefits for all customers and the organization, but is not payed for by customers.

Architecture: Adopting a software platform requires an architecture that manages the features' assets. Typically, software product-line concepts are adopted for this pur-

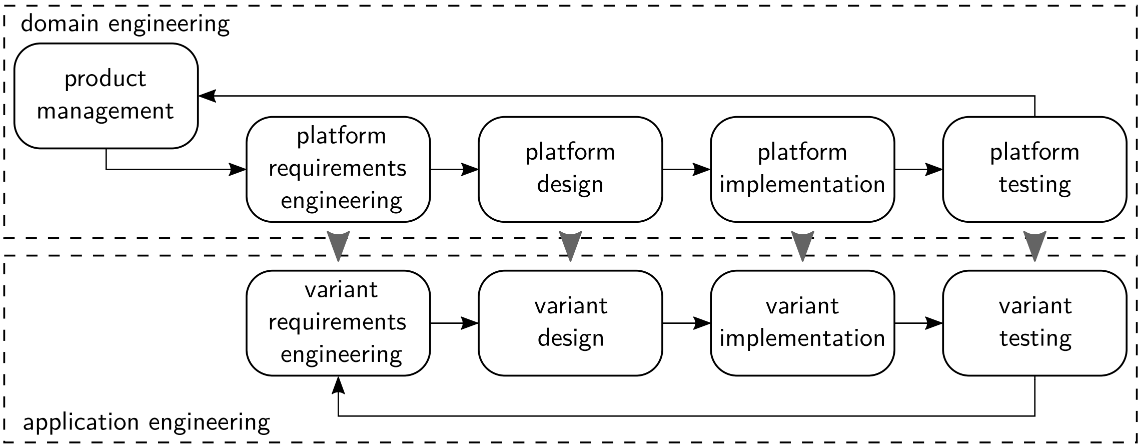


Figure 2.3: Typical software product-line process model based on Pohl et al. [2005].

pose, for instance, variability mechanisms (cf. Section 2.3.4), variability models (cf. Section 2.3.3), and configurator tools (cf. Section 2.3.5).

Process: Software product-line engineering is usually separated into two different processes that an organization has to adopt: domain and application engineering. We describe these processes in more detail in Section 2.3.2.

Organization: Employing a platform and the corresponding processes requires the definition of appropriate roles and responsibilities. For example, a platform team is usually responsible for maintaining the platform itself, while feature teams develop and contribute individual features.

Note that these concerns are not independent, but impact each other. For illustration, consider the following short examples from an organization we studied [Lindohf et al., 2021]:

Business – Organization: If a new system is marketed as a platform, the organization adapts its structures accordingly.

Business – Process: Whether a new variant is marketed as platform-based or as individual system, directly impacts the change propagation to the platform.

Architecture – Business: Features that are visible to, and can be selected by, customers are easily configurable in the architecture (e.g., based on plug-ins).

Architecture – Process: Technical processes (e.g., testing, continuous integration) are directly connected to the platform architecture.

Architecture – Organization: Specific organizational units are responsible for certain parts of the platform architecture, which is partitioned based on these responsibilities.

Process – Organization: The employed processes are structured around organizational units, which, in turn, are defined based on the processes.

Overall, the BAPO model is a helpful means to identify and understand the dependencies between the different concerns of platform engineering.

2.3.2 Engineering Processes

Process models

In Figure 2.3, we display a typical software product-line process model. As we can see, most of such process models distinguish between domain and application engineering [Apel et al.,

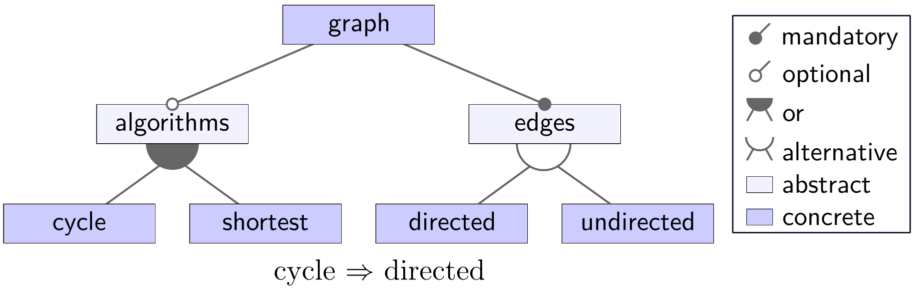


Figure 2.4: A feature model based on Lopez-Herrejon and Batory [2001].

2013a; Pohl et al., 2005; van der Linden et al., 2007]. The individual processes comprise five and four activities, respectively.

Domain engineering

Domain engineering defines the process of developing the software platform (i.e., developing for reuse). The first step during domain engineering is the product management, which essentially covers the business concern of the BAPO model (cf. Section 2.3.1). Consequently, this step is concerned with scoping the platform (i.e., defining the variant portfolio), organizing the funding, and defining a market strategy. Using the defined scope, the next step is to elicit actual platform requirements and derive features from those. So, mandatory and optional features, as well as their dependencies, are defined and documented, for instance, by using a variability model (cf. Section 2.3.3). Next, the platform architecture is designed and implemented, for which a variability mechanism (cf. Section 2.3.4) is used to enable that optional features can be configured. Finally, the platform is tested and can then be used to derive actual variants by reusing the implemented features.

Application engineering

Application engineering defines the process of developing a customer-specific variant (i.e., developing with reuse). First, the customer requirements for a new variant must be elicited. Then, the design of the variant is defined by mapping the requirements to features. Ideally, all required features have already been implemented in the platform, in which case the variant can simply be configured and derived (cf. Section 2.3.5). Otherwise, new features may be introduced into the platform, or directly into the variant if they shall not be integrated into the common codebase. Finally, the resulting variant is tested before delivering it to the customer.

2.3.3 Variability Modeling

Variability model

A variability model allows an organization to capture the features of a domain (e.g., for scoping) or platform (e.g., for configuring) as well as the dependencies between them. Consequently, variability models are a key artifact for managing a software platform, documenting features (ideally also tracing to their assets), and defining constraints that must be enforced while configuring variants. Various different notations for variability models have been proposed and are used in practice [Berger et al., 2013a; Chen and Babar, 2011; Czarnecki et al., 2012; Schaefer et al., 2012], for instance, feature models [Kang et al., 1990; Nešić et al., 2019], decision models [Schmid et al., 2011], or UML [Object Management Group, 2017]. In research and practice, feature models are the most established notation for variability modeling [Benavides et al., 2010; Berger et al., 2013a; Lettner et al., 2015; Thüm et al., 2014a]. For this reason, we also focus on feature models, and their graphical representation as feature diagrams [Apel et al., 2013a; Schobbens et al., 2006], in this dissertation.

Feature model

We show a small example feature model based on the well-known graph library [Lopez-Herrejon et al., 2011] in Figure 2.4. A feature model captures the features of a platform in a tree-like structure, and defines all valid configurations by specifying constraints as follows.

The tree structure of a feature model represents a hierarchy of parent-child constraints, which enforce that all parents of a selected feature are also selected. Features may be optional (i.e., `algorithms`), allowing developers to disable or enable them for a variant, or mandatory (i.e., `graph`, `edges`). Additionally, features can be grouped into or-groups (i.e., `cycle`, `shortest`) and alternative-groups (i.e., `directed`, `undirected`). All constraints that cannot be defined with those constructs can be added as cross-tree constraints. Cross-tree constraints are usually expressed as propositional formulas over features, for instance, to express imply (i.e., `cycle` ⇒ `directed`) or exclude dependencies between features in different sub-trees of the feature model — and are typically listed below the actual model. Finally, abstract features (i.e., `algorithms`, `edges`) can be used to improve the structure of a feature model. However, they have no implementation (i.e., assets), which means that only concrete features actually contribute to variants. We remark that other notations of feature models provide additional constructs that are not relevant for us [Apel et al., 2013a; Benavides et al., 2010; Berger et al., 2013b]. For instance, some notations allow to use other types of features besides Boolean or to assign feature attributes.

2.3.4 Variability Mechanisms

Implementation techniques

Variability mechanisms are implementation techniques for variation points; points at which a platform can be configured by enabling or disabling optional features to derive a customized variant. Numerous mechanisms can be used to implement variation points [Apel et al., 2013a; Bachmann and Clements, 2005; Berger et al., 2014b; Gacek and Anastasopoules, 2001; Svahnberg et al., 2005; Zhang et al., 2016], for instance, preprocessors, components, plug-ins, version-control systems, feature-oriented programming [Prehofer, 1997], or simply runtime parameters. Each variability mechanism exhibits different properties with own pros and cons.

Dimensions of variability

Apel et al. [2013a] classify variability mechanisms along three dimensions:

Binding time refers to the point in time at which features are included (i.e., bound) into a variant. Generally, we can distinguish compile-time binding (i.e., features are selected before, and disabled ones ignored while, compiling), load-time binding (i.e., features are selected at program start), and runtime binding (i.e., features are selected during program execution). In that order, examples for variability mechanisms that support each binding time are preprocessors, plug-ins, and runtime parameters. However, a variability mechanism may support different bindings times (e.g., parameters). Note that more fine-grained classifications and various terminologies regarding binding times exist [Apel et al., 2013a; Berger et al., 2015; Rosenmüller, 2011].

Technology distinguishes whether a variability mechanism is language-based or tool-based. A language-based mechanism, such as feature-oriented programming, relies on the capabilities (potentially based on extensions) of the programming language itself to implement variation points. In contrast, a tool-based mechanism, such as a preprocessor, introduces external tools to implement and manage variation points.

Representation refers to how variation points are expressed in the source code. Annotation-based mechanisms (e.g., preprocessors) add annotations to the source code that map to features. If a feature is disabled, the corresponding code can be removed or simply ignored when deriving a variant. Composition-based mechanisms (e.g., feature-oriented programming) implement features in separated assets (e.g., a class or module for each feature). When deriving a variant, all assets belonging to an enabled feature are composed (i.e., integrated into each other to build the variant).

Next, we exemplify two variability mechanisms in more detail that we used in our research: preprocessors and feature-oriented programming.

```
1  public class Graph {
2   private List<List<Integer>> edges;
3   // ...
4   public void addEdge(
5      int from, int to) {
6    /* if [ directed ] */
7    edges.get(from).add(to);
8    /* else [ directed ] */
9    edges.get(from).add(to);
10   edges.get(to).add(from);
11   /* end [ directed ] */
12  }
13  // ...
14 }
```

(a) Annotated codebase.

```
public class Graph {
 private List<List<Integer>> edges;
 // ...
 public void addEdge(
    int from, int to) {
  edges.get(from).add(to);
 }
 // ...
}
```

(b) Preprocessed variant.

Figure 2.5: Conceptual example of using a preprocessor.

Preprocessors

Preprocessors

Preprocessors, most prominently the C preprocessor [Kernighan and Ritchie, 1978], arguably represent the most widely used variability mechanism in industrial and open-source systems [Apel et al., 2013a; Hunsen et al., 2016; Liebig et al., 2010, 2011; Medeiros et al., 2015]. While some preprocessors support a variety of additional purposes (e.g., macro definition, file inclusion), we focus on the concept of conditional compilation. Conditional compilation allows to implement variation points by annotating the source code with so-called directives (e.g., `#ifdef`). Each directive involves a macro that defines a (sometimes arbitrarily complex) condition under which it is true or false. For instance, the C preprocessor's `#ifdef` is shorthand for evaluating a single constant (i.e., a feature name), while its `#if` accepts any regular expression over features connected via logical operators (e.g., `||`, `&&`). While preprocessing, each macro is compared against the provided configuration to evaluate its condition. Based on this evaluation and the corresponding directive (e.g., `#ifdef` and `#ifndef` are opposites for the same condition), the preprocessor prunes the source code to involve only enabled source code. Note that many preprocessors are purely text-based tools, which is why they can be adopted for any form of text. For this reason, each preprocessor typically defines an own syntax for, and set of, directives — often inspired by those of the C preprocessor (i.e., `#ifdef`, `#if`, `#ifndef`, `#else`, `#elif`, and `#endif`).

Code example

In Figure 2.5, we show a conceptual example (aligning to Figure 2.4) to illustrate how a preprocessor works. Since we display Java code, we adopted the annotation syntax of a corresponding preprocessor, namely Munge.[2] In Figure 2.5a, we sketch how directed and undirected edges could be added to a graph. This code can be compiled into two versions by configuring the feature `directed`. If enabled, the annotation in line 6 becomes true, while the one in line 8 is false. Thus, lines 9–10 are removed from the source code if the feature `directed` is enabled (cf. Figure 2.5b). Note that the feature `undirected` does not appear, but is implicitly specified by the `/* else */` in line 8. This is only made explicit in the feature model we display in Figure 2.4, and shows that there can be comprehension problems (e.g., regarding what feature the `/* else */` refers to) if the developer does not know about the alternative dependency. Note that another implementation that defines both features individually (i.e., ending `directed` and starting `undirected` instead of the `/* else */`) would actually allow to derive four variants. This would violate the feature model and allow to configure two faulty variants (i.e., both features could be enabled or disabled at

[2] https://github.com/sonatype/munge-maven-plugin

```
public class Graph {
 private List<List<Integer>> edges;
 // ...
 public void addEdge(
     int from, int to) {
  // to refine in features
  this.refresh();
 }
 // ...
}
```

(a) Feature `graph`.

```
public class Graph {
 private List<List<Integer>> edges;
 // ...
 public void addEdge(
     int from, int to) {
  edges.get(from).add(to);
  original(from, to);
 }
 // ...
}
```

(b) Feature `directed`.

```
public class Graph {
 private List<List<Integer>> edges;
 // ...
 public void addEdge(
     int from, int to) {
  edges.get(from).add(to);
  // to refine in features
  this.refresh();
 }
 // ...
}
```

(c) Composed variant.

Figure 2.6: Conceptual example of using feature-oriented programming.

the same time). Due to such problems, preprocessors have often been criticized in research, even though empirical studies do not fully support the critique [Fenske et al., 2020].

Feature-Oriented Programming

Feature-oriented programming

Instead of annotating features in a single codebase, feature-oriented programming [Prehofer, 1997] relies on physically separating their assets into modules. A feature can refine any other feature, which is why feature modules may comprise a number of assets that specify extensions to the codebase. To derive a variant, these assets are composed by merging their structures (e.g., based on feature-syntax trees with classes and methods as nodes [Apel and Lengauer, 2008; Apel et al., 2013b]). For this purpose, the composition mechanism relies on a keyword-like method (e.g., `original()`) to identify at what position in the structure feature refinements are integrated. If this method is missing, the refinement represents either new code or overwrites the existing code (i.e., similar to inheritance). Since a variant is derived by iteratively composing feature modules that extend or overwrite existing assets, it is important to define a proper composition order. This order can be freely defined by developers, but more advanced tools build upon feature models to derive a suitable order (e.g., based on the feature hierarchy). Over time, several tools have been adopted to support feature-oriented programming, for instance, AHEAD [Batory, 2006; Batory et al., 2004], FeatureHouse [Apel et al., 2009, 2013b], or FeatureIDE [Meinicke et al., 2017; Thüm et al., 2014b].

Code example

In Figure 2.6, we display a conceptual example (again aligning to Figure 2.4) to explain more intuitively how feature-oriented programming works. Our example resembles FeatureHouse [Apel et al., 2009, 2013b], but we adapted it to improve its readability. In Figure 2.6a, we can see that the base feature `graph` has one method, which shall be refined by other features and involves a method call for refreshing the graph. We display the feature `directed` in Figure 2.6b, which has the same structure (i.e., class, method) as the base feature to enable composition. The `original()` method defines at which position the refinement shall be introduced into the previously composed assets. In this case, line 6 of Figure 2.6b shall be placed before the code it is composed into. As a results, a composition of both modules would lead to the code we display in Figure 2.6c.

2.3.5 Configuring

Configuring

Based on the previous concepts, a platform allows developers to configure and derive variants [Apel et al., 2013a; Meinicke et al., 2017]. For this purpose, developers can use configurator tools that present all configuration options (i.e., features) in a GUI, typically as a list of checkboxes that can be selected. While configuring remains a manual process (except for deriving the exact same variant again), configurator tools guide the process by using the defined feature dependencies (e.g., specified in the feature model) to identify whether a configuration becomes invalid (i.e., it violates dependencies) [Benavides et al., 2010]. There are different resolution strategies to handle invalid configurations, particularly if customers can configure the variants themselves [Thüm et al., 2018]. Many advanced configurator tools actually support developers by propagating their decisions, meaning that they automatically select and deselect features to prevent invalid configurations (e.g., selecting all features that are required by the one a developer selected) [Apel et al., 2013a; Hubaux et al., 2012; Krieter, 2019; Krieter et al., 2018b]. Using a complete configuration, a variant can be automatically derived from the platform. While new customer requirements may still require glue code, the ability to derive variants fully automatically allows to assemble assets with almost no costs, to reuse configurations (e.g., for continuous delivery [Humble and Farley, 2010], and to automatically test variants.

2.4 Summary

Chapter summary

In this chapter, we introduced the basics of variant-rich systems and particularly software product-line engineering. First, we distinguished between clone & own and platform engineering as the two strategies for reusing software. We discussed how these strategies are employed and what pros as well as cons they have to highlight how they are used to engineer variant-rich systems. Then, we described the different concepts related to software product-line engineering that are relevant for this dissertation. Namely, we introduced adoption strategies, engineering processes, variability modeling, variability mechanisms, and configuring. Note that we focused on explaining those concepts that are fundamental to all parts of this dissertation. We introduce further concepts in the chapters for which they are relevant or in which we investigated them in great detail for the first time. For this purpose, we define a detailed conceptual framework at the beginning of each chapter to establish key terms and explain the required background knowledge as far as needed.

3. Economics of Software Reuse

This chapter builds on publications at ESEC/FSE [Krüger and Berger, 2020b], ICSE [Krüger, 2018a], SEVIS [Krüger et al., 2019a], SPLC [Åkesson et al., 2019; Debbiche et al., 2019; Krüger et al., 2016a, 2017a, 2018a; Kuiter et al., 2018b; Strüber et al., 2019], VaMoS [Krüger and Berger, 2020a], Empirical Software Engineering [Lindohf et al., 2021], and Software: Practice and Experience [Krüger et al., 2018d].

Chapter structure

In this chapter, we investigate the economics of software reuse (**RO-E**). First, we discuss cost models for software product-line engineering to establish an intuition on how to decide for a reuse strategy and to identify open challenges (Section 3.1). Second, we provide and compare empirical data on the economics of clone & own and platform engineering (Section 3.2). Finally, we report experiences and data from five case studies in which we re-engineered real-world, variant-rich systems towards systematically managed platforms (Section 3.3). Building on these insights, the contributions in this chapter allow practitioners to make empirics-based decisions on whether (re-)engineering a platform is useful, and with which reuse strategy a new variant should be developed. For researchers, we contribute insights regarding the most expensive and most important challenges of (re-)engineering variant-rich systems that require further investigations and novel techniques. In particular, these challenges highlight our motivation for investigating our other research objectives.

Conceptual framework of economics

We display a more detailed overview of our conceptual framework regarding reuse economics in Figure 3.1. Within a *project*, an organization develops a new variant that is defined by a number of *requirements*. Depending on these requirements, the organization must decide which *reuse strategy* to employ (i.e., clone & own or a platform), potentially involving the re-engineering of an existing variant-rich system. The selected reuse strategy defines the concrete *process* used for developing, and finally delivering, the specified *variant*. Such a development process involves *activities*, for instance, implementing features or fixing bugs, which are performed by the organization's *developers*. All activities cause *costs* (i.e., money spent) that are influenced by the *cost factors* (a.k.a., cost drivers) relating to the project, such as the number of developers or the size of the variant [Boehm, 1984; Heemstra, 1992; Leung and Fan, 2002]. In this chapter, we show that cost factors that are important in the context of software reuse relate particularly to the other objectives in our conceptual framework. Therefore, the economics of software reuse should be used to define these objectives, which consequently cause costs.

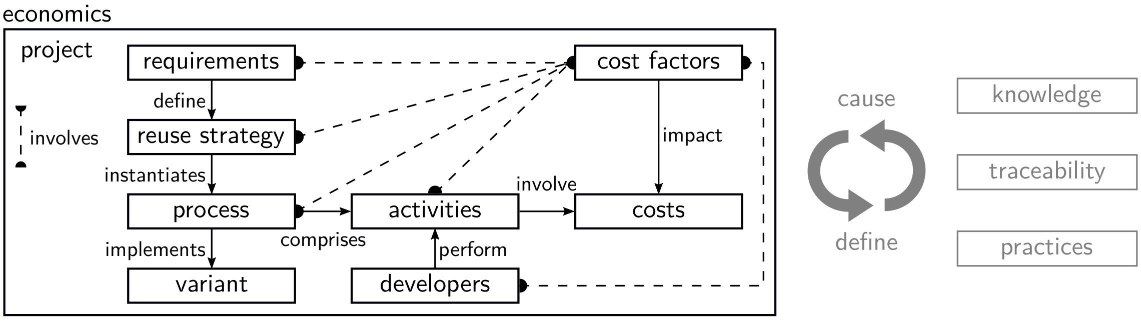


Figure 3.1: Details of the economics objective in our conceptual framework (cf. Figure 1.1).

3.1 Cost Estimation for Variant-Rich Systems

Decision making

Developing a new variant, and particularly (re-)engineering a variant-rich system, requires an organization to decide for a specific development process and reuse strategy. Such a decision causes costs, is risky, and has long-term impact on an organization's structure. For this reason, most organizations conduct a domain and market analysis to compare between different options before deciding how to engineer a variant. While the specifics of such an analysis can vary (e.g., employed methods, considered properties), cost estimations are usually a core aspect and key to manage a project effectively [Bayer et al., 1999; Berger et al., 2020; Biffl et al., 2006; Boehm and Huang, 2003; Dolado, 2001; Jørgensen, 2014; Koziolek et al., 2016; Northrop, 2002; Trendowicz, 2013]. Regarding variant-rich systems, it is particularly challenging to estimate costs and compare reuse strategies, since clone & own is readily available and causes few adoption costs, while a platform requires large upfront investments that cannot be directly assigned to a specific project or customer. Consequently, an organization must compare the investments into a platform against its uncertain benefits in the future [Krüger et al., 2016a; Martinez et al., 2015; Schmid and Verlage, 2002].

Section contributions

In this section, we investigate cost estimation for (re-)engineering variant-rich systems by addressing three sub-objectives of **RO-E** [Krüger, 2016, 2018a; Krüger et al., 2016a, 2019a]:

RO-E$_1$ *Identify and compare existing software product-line cost models.*

Cost models provide guidance for estimating the costs and understanding the economic impact of a development or reuse strategy [Boehm, 1984; Heemstra, 1992; Jørgensen and Boehm, 2009; Leung and Fan, 2002]. We surveyed existing software product-line cost models, since these compare platform engineering with single-system development or clone & own. Analyzing these cost models helps to elicit relevant cost factors, understand the models' limitations, decide which model to select, and identify opportunities for improvement.

RO-E$_2$ *Analyze cost estimations in the context of (re-)engineering variant-rich systems.*

While surveying the cost models, we found that few involve the re-engineering of variant-rich systems. To tackle this issue, we analyze the cost functions defined in the SIMPLE cost model [Böckle et al., 2004a,b; Clements et al., 2005], particularly in the context of such a re-engineering and of economical cost curves. By discussing these adaptations, we provide an intuition for practitioners and researchers regarding what properties and cost factors are important to adapt.

RO-E$_3$ *Discuss feature location in the context of re-engineering economics.*

Feature location [Dit et al., 2013; Duszynski, 2010; Rubin and Chechik, 2013b; Strüber et al., 2019] is expensive, and a key activity regarding the re-engineering of a variant-rich system. We relate manual feature location and the consequent program comprehension to the results we obtained for our previous sub-objectives. Using our insights, we motivate our remaining research objectives in more detail.

In the remainder of this section, we first report our survey of existing cost models for software product lines in Section 3.1.1. Then, we analyze the SIMPLE cost model and its adaptations for re-engineering variant-rich systems in Section 3.1.2. To motivate our research objectives, we discuss our insights in the context of feature location in Section 3.1.3.

3.1.1 RO-E$_1$: Cost Models for Software Product-Line Engineering

Cost models

Software cost estimation is the process of predicting the costs, benefits, and risks of developing, extending, or re-engineering a software system [Boehm, 1984; Heemstra, 1992; Leung and Fan, 2002]. Various methods with different pros and cons can be used to estimate costs, of which the reliable ones are cost models, expert judgment, and analogies to historical data [Boehm, 1984; Heemstra, 1992; Jørgensen, 2014; Jørgensen and Boehm, 2009; Jørgensen and Shepperd, 2007; Leung and Fan, 2002]. Typically, different methods are combined to improve the reliability of an estimate. In the following, we focus on cost models, which are mathematical functions whose variables represent cost factors. By analyzing existing cost models, we can obtain an overview of what cost factors researchers and practitioners consider relevant in the context of software reuse. Note that we involve expert judgments and analogies in Section 3.2.

Literature survey

Numerous cost models have been proposed to help organizations estimate the costs of developing a new software system and of employing different reuse techniques [Boehm et al., 2000; Jørgensen and Shepperd, 2007; Lim, 1996; Mili et al., 2000]. We conducted a literature survey of software product-line cost models [Krüger, 2016] using an automated search on five digital libraries: Google Scholar, ACM Digital Library, IEEE Xplore, Springer Link, and Science Direct. To this end, we defined the following search string:

```
(‘‘software product line’’ OR ‘‘software product family’’) AND
(‘‘cost estimation’’ OR ‘‘cost model’’ OR ‘‘investment’’)
```

In the returned publications, we identified three literature surveys on software product-line cost models [Ali et al., 2009; Blandón et al., 2013; Charles et al., 2011] and two systematic literature reviews on software product-line economics that involve cost models [Heradio et al., 2013; Khurum et al., 2008]. From these five literature surveys, we extracted all cost models and matched these with the remaining publications. Moreover, we used forwards and backwards snowballing [Wohlin, 2014] on all included publications to add novel cost models and their extensions (e.g., by Tüzün and Tekinerdogan [2015]). We included any publication that defines a cost model for software product-line engineering. Note that we did not have access to some publications at that point in time (e.g., to those by Schmid [2002], Matsumoto [2007], or Nonaka et al. [2007a,b]), which is why these were excluded in our original literature survey. To tackle this issue, we checked the availability of all publications again. We also included additional literature surveys that were not available to us before (e.g., by Thurimella and Padmaja [2014]), were published more recently (i.e., by Heradio et al. [2018]), or that involve cost models, but only as a background or a side topic (e.g., by Parmeza [2015]).

Results and Findings

Identified cost models

We display the mapping of the ten literature surveys and 14 cost models (described and refined in 26 publications) we identified in Table 3.1. Note that we consolidate all publications relating to or extending a cost model, for instance, Tüzün and Tekinerdogan [2015] add an

Table 3.1: Overview of software product-line cost models surveyed in the literature. The cost-models, references, and literature surveys are ordered by publication year and alphabetical. Our previous literature survey is highlighted by the gray column.

	literature survey									
cost model / references	Nóbrega et al. [2006] ★	Khurum et al. [2008]	Ali et al. [2009]	Charles et al. [2011]	Blandón et al. [2013]	Heradio et al. [2013]	Thurimella and Padmaja [2014] ★	Parmeza [2015] ★	Krüger [2016]	Heradio et al. [2018] ★
Withey [1996]	○	○	●	○	○	●	●	●	●	○
Poulin [1997]	●	○	●	●	●	●	●	●	●	●
Schmid										
[Schmid, 2002]	○	●	○	○	○	○	○	○	○	○
[Schmid, 2003]	○	○	●	○	○	○	○	●	●	○
[Schmid, 2004]	○	●	○	○	○	○	○	○	○	○
ABC Analysis										
[Cohen, 2003]	○	○	●	○	○	●	●	○	●	●
Peterson [2004]	○	●	●	○	●	●	○	○	●	●
(q)COPLIMO										
[Boehm et al., 2004]	●	●	●	●	●	●	●	●	●	●
[In et al., 2006]	○	●	●	●	●	●	●	●	●	●
[Heradio et al., 2012]	○	○	○	○	○	●	○	○	●	○
SIMPLE										
[Böckle et al., 2004a]	○	●	○	○	○	●	○	●	○	○
[Böckle et al., 2004b]	●	●	○	○	●	●	●	○	●	○
[Clements et al., 2005]	○	○	●	●	●	●	●	●	●	●
[Pohl et al., 2005]	○	○	○	○	○	○	●	○	○	○
[Tüzün and Tekinerdogan, 2015]	○	○	○	○	○	○	○	○	●	○
Tomer et al. [2004]	●	○	○	○	○	○	○	○	○	○
SoCoEMo-PLE(2)										
[Ben Abdallah Ben Lamine et al., 2005a]	●	○	○	○	○	●	○	●	○	○
[Ben Abdallah Ben Lamine et al., 2005b]	○	●	●	●	○	●	○	○	●	○
[Ben Abdallah Ben Lamine et al., 2005c]	○	○	●	●	○	○	○	○	●	○
Ganesan et al. [2006]	○	●	●	○	○	○	○	○	●	○
Wesselius [2006]	○	○	●	○	○	●	○	○	●	○
Matsumoto [2007]	○	●	○	○	○	○	○	○	○	○
Nonaka										
[Nonaka et al., 2007a]	○	●	○	○	○	○	○	○	○	○
[Nonaka et al., 2007b]	○	●	○	○	○	○	○	○	○	○
InCoME										
[Nóbrega, 2008]	○	○	○	○	○	○	○	●	○	○
[Nóbrega et al., 2008]	○	○	●	○	○	●	○	●	●	○

●: covered – ○: not covered – ★: not included in our previous literature survey experience factor to SIMPLE. Moreover, not all cost models mentioned or referenced in a literature survey are actually discussed in that survey, for example, Blandón et al. [2013] focus on a comparison of COPLIMO and SIMPLE. We can see that software product-line cost models have been proposed particularly in the 2000s, aligning to a bibliographic analysis of the software product-line community by Heradio et al. [2016]. By triangulating the findings of all surveys and verifying these findings with the cost models, we identified three major challenges.

Diversity of cost factors

First, existing software product-line cost models are diverse and have varying levels of granularity regarding the cost factors they consider [Ali et al., 2009; Charles et al., 2011; Krüger, 2016; Nóbrega et al., 2006; Thurimella and Padmaja, 2014]. For instance, SIMPLE defines cost functions on an abstract level (e.g., organizational costs) that must be estimated by an expert, and thus are more suitable for an overarching understanding of costs. In contrast, other cost models (e.g., COPLIMO) are far more fine-grained in their cost factors (i.e., lines of code reused as black-box), and thus also more complex. Similarly, some cost models cover only the adoption phase of a platform, disregarding that the platform must be maintained throughout its whole life-cycle, including quality assuring, feature enhancing, and bug fixing [Ali et al., 2009; Krüger, 2016]. Such diversity makes it harder to select a cost model for a specific scenario and organization, particularly since it is unclear for what reasons what cost factors are considered. Interestingly, despite this diversity, most cost models consider knowledge about a variant-rich system and locating features (or assets), either directly as cost factors (e.g., "unfamiliarity factor" [Boehm et al., 2004]) or by indirectly referring to them (e.g., included in "skill" [Withey, 1996]) [Böckle et al., 2004a,b; Clements et al., 2005; Nóbrega, 2008; Nóbrega et al., 2008; Peterson, 2004; Pohl et al., 2005; Poulin, 1997; Schmid, 2002, 2003]. Notably, Tüzün and Tekinerdogan [2015] focus solely on integrating an estimation of developers' knowledge into SIMPLE.

Missing validations and empirical data

Second, validating cost models is a challenging task, and most of the software product-line cost models build on single experiences or fictional data instead of real-world validations [Ali et al., 2009; Jørgensen and Shepperd, 2007; Khurum et al., 2008; Krüger, 2016; Leung and Fan, 2002; Nolan and Abrahão, 2010]. The main challenge for validating cost models is the availability of reliable, empirical data that exceeds single studies, allows for deeper insights into cost factors, and represents real projects [Ali et al., 2009; Khurum et al., 2008; Krüger and Berger, 2020a,b; Mustafa and Osman, 2020]. Usually, such economic data is not published, because it represents critical business information that organizations do not want to share [Koziolek et al., 2016; Yoshimura et al., 2006a]. Moreover, there is a missing understanding of most cost factors and their actual impact on costs. For this reason, several cost models define rating scales for such cost factors, which, unfortunately, are rarely reliable and sparsely based in empirical evidence [Ali et al., 2009; Jørgensen, 2014; Leung and Fan, 2002].

Cost model for re-engineering variant-rich systems

Third, existing software product-line cost models consider the re-engineering of cloned variants towards a platform insufficiently [Koziolek et al., 2016; Krüger et al., 2016a]. Few models mention that a platform may be adopted from a set of existing variants, and most focus solely on the proactive adoption of a platform. Even though some cost models include re-engineering scenarios (e.g., SIMPLE), these scenarios are usually on an abstract level and neither explain how to adapt cost factors nor what the impact of existing assets or knowledge on the economics of a platform is. Consequently, we are lacking a detailed understanding of how the "remains" of previous projects (e.g., knowledge, experiences, assets) can lead to varying results [Jørgensen, 2004, 2007; Krüger and Hebig, 2020; Tüzün and Tekinerdogan, 2015] — which requires further analyses before an actual evidence-based cost model for re-engineering variant-rich systems can be defined.

RO-E_1: Cost Models for Software Product-Line Engineering

We analyzed existing software product-line cost models and learned:

- *At least 14 cost models for platform engineering have been proposed, of which COPLIMO and SIMPLE seem to be the most prominent ones.*
- *The cost factors considered in the cost models are highly diverse, rarely well understood or measurable, and often focused solely on adopting a platform.*
- *We miss reliable empirical data for investigating cost factors as well as constructing and validating cost models.*
- *Particularly re-engineering cloned variants into a platform is rarely considered explicitly in cost models.*

Threats to Validity

Completeness

We did not conduct this study as a systematic literature review. Instead, we relied on ten existing surveys to identify software product-line cost models and understand their properties. We mitigated the threat of missing relevant cost models by employing an automated and a snowballing search to improve completeness. In Table 3.1, we can see that a few cost models seem particularly prominent — namely, Poulin's model, (q)COPLIMO, SIMPLE, and SoCoEMo-PLE(2). Of these, COPLIMO and SIMPLE are arguably the most established ones [Blandón et al., 2013; Lindohf et al., 2021]. Furthermore, of all cost models covering the whole life-cycle of a platform, we [Krüger, 2016] found reports on (partial) considerations in practice only for COPLIMO [Nolan and Abrahão, 2010] and SIMPLE [Koziolek et al., 2016; Nolan and Abrahão, 2010; Tang et al., 2010]. This improves our confidence that we covered the most relevant cost models and that the findings, which we derived from the surveys as well as the cost models, are reasonable.

3.1.2 RO-E_2: The SIMPLE Cost Model and Re-Engineering

SIMPLE

Since we found that re-engineering economics are rarely considered in existing software product-line cost models, we [Krüger, 2016; Krüger et al., 2016a] now build on SIMPLE (the *Structured Intuitive Model for Product Line Economics*) [Böckle et al., 2004a,b; Clements et al., 2005] to discuss these economics. SIMPLE itself is actually not a fine-grained algorithmic model, but provides a classification of costs (called *cost functions*) and exemplifies relevant cost factors. For each cost function, a concrete cost estimation must be provided, for instance, by using a judgment-based estimate or another algorithmic cost model. We selected SIMPLE for our discussion, since its high level of abstraction allows us to more intuitively explain adaptations for re-engineering. Moreover, SIMPLE is one of few cost models defining example scenarios, and one concretely for re-engineering variant-rich systems.

Scenarios and cost functions

SIMPLE defines a general scenario in which an organization may intend to estimate the costs of a platform. One instance of this general scenario covers the re-engineering of (cloned) variants into a platform as follows:

> *"An organization has a set of products in the marketplace that were developed more or less independently. It wishes to explore the possibility of redeveloping them using a product line engineering approach."*
>
> [Böckle et al., 2004a,b]

To estimate the costs of such a scenario, SIMPLE defines five core cost functions (C):

C_{org} represents *organizational* costs for establishing platform engineering in an organization involving, for instance, training, reorganization, or process improvements.

C_{cab} represents the costs of developing the *core asset base* involving, for instance, commonality and variability analysis, introducing tools, or designing the platform architecture.

C_{unique} represents the costs of implementing *unique* parts of a variant involving, for instance, glue code or features that shall not be integrated into the platform.

C_{reuse} represents the costs of *reusing* features of the platform in a variant involving, for instance, identifying, integrating, and testing assets.

C_{evo} represents the costs of *evolving* the platform in terms of new releases involving, for instance, bug fixes, feature enhancements, or quality improvements.

Adoption costs in SIMPLE

In Equation 3.1, we display how the first four cost functions relate to each other considering a number n of distinct variants v_i that an organization intends to develop in their platform (i.e., causing the adoption costs $C_{platform}$).

$$C_{platform} = C_{org} + C_{cab} + \sum_{i=1}^{n}(C_{unique}(v_i) + C_{reuse}(v_i)) \tag{3.1}$$

To decide between platform engineering and clone & own, an organization has to compare the costs for a platform to those of developing a new variant independently, represented by the function $C_{c\&o}(v_i)$. The *savings* that can be achieved by developing variants with a platform instead of clone & own can then be estimated using Equation 3.2.

$$C_{savings} = \sum_{i=1}^{n} C_{c\&o}(v_i) - C_{platform} \tag{3.2}$$

Finally, an organization can calculate the return on investment (ROI) of a potential platform by comparing its savings to the required investments, as we show in Equation 3.3.

$$ROI = \frac{C_{savings}}{C_{org} + C_{cab}} \tag{3.3}$$

Evolution costs in SIMPLE

SIMPLE considers evolution and maintenance based on the costs of releasing a new revision of the platform and its variants (C_{evo}). For this purpose, three cost functions are defined and summed according to the formula we display in Equation 3.4.

$$C_{evo} = \sum_{i=1}^{n}(C_{cabu}(v_i) + C_{unique}(v_i) + C_{reuse}(v_i)) \tag{3.4}$$

In this formula, $C_{cabu}(v_i)$ represents the costs of *updating the core asset base*, which may include asset improvements, feature enhancements, or bug fixes, among others. Since any update is potentially propagated to all variants of the platform, updates can cause side effects in any variant [Bogart et al., 2016; Cotroneo et al., 2019]. For this reason, SIMPLE also considers the costs for updating unique parts (e.g., adapting glue code and feature interactions) and re-integrating reused features for each variant again. In the following, we discuss the five core cost functions with respect to re-engineering variant-rich systems.

Organizational Costs (C_{org})

Organization of re-engineering

To establish platform engineering, an organization must invest into the BAPO concerns (cf. Section 2.3), three of which are non-technical and relate to the organizational costs of SIMPLE. Contrary to proactive engineering, re-engineering a platform from cloned variants involves developers that have a knowledge base about their established processes for cloning (e.g., using branches in Git). This has important implications for the cost functions of SIMPLE. On the one hand, an organization has to train its developers to employ different processes, use new tools, and communicate differently. Particularly, this requires investments into unifying workflows and programming styles as well as implementing traceability across different assets to ensure that the platform is compatible throughout all processes [Böckle et al., 2002; Gacek et al., 2001; Mansell, 2006; Northrop, 2002; Pohl et al., 2005; van der Linden, 2002, 2005; van der Linden et al., 2004]. Consequently, the organizational costs of SIMPLE are firstly defined by the investments needed to change the developers' culture, and the costs heavily depend on whether the developers can be convinced. On the other hand, an organization can build upon its experiences and developers' knowledge, for instance, to define which features are within the platform's scope and have also an appropriate quality. This reduces the overall risks (the variants are established in the market) and investments compared to proactive engineering — if we exclude the investments into the cloned variants.

Costs for the Core Asset Base (C_{cab})

Platform re-engineering

The fact that an organization is considering to re-engineer its variants into a platform has three major implications [Clements and Krueger, 2002; Krüger, 2016; Krüger et al., 2016a; Schmid and Verlage, 2002]: First, the organization successfully developed variants with clone & own, reducing the risk that the platform may fail in the market. Consequently, a domain and market analysis may be less extensive, but the organization still needs to decide which features to re-engineer into the platform. Second, the economical burden of developing with clone & own is high enough for the organization to consider to re-invest into a pure re-engineering project that may yield only long-term benefits. So, the organization must decide which features have the right trade off between investments into the re-engineering and pay off during maintenance and future development. Finally, the organization may (but does not have to) reuse existing assets and developers' knowledge to reduce the costs of developing the platform compared to a proactive adoption. However, there may be quality problems and inconsistencies between existing assets that are expensive to resolve — potentially to the point at which implementing them anew is the better solution. As a result, the economical decisions for re-engineering the core asset base of a platform arguably comprise more facets than in proactive engineering.

Costs for New Variants (C_{unique} & C_{reuse})

Developing variants

In SIMPLE, variant development consists of two cost functions to account for unique and reused features of a variant. Arguably, these two cost functions change the least for re-engineering a variant-rich system compared to the previous two cost functions, since developing a new variant based on the re-engineered platform is the same as developing it from any other platform with the same properties (e.g., quality, features, variability mechanism). Still, these cost functions indicate important decisions for a re-engineering project. For example, an organization must assess the costs of re-engineering a feature into the core asset base against the costs of developing it anew as a unique part of a variant or as a reusable feature of the platform. Since this decision is a trade off between the costs for re-engineering features into the core asset base or re-developing them, a feature-based estimation perspective seems more relevant for re-engineering variant-rich systems — in contrast to the mostly variant-based perspective of existing software product-line cost models.

Maintenance Costs (C_{evo})

Maintaining the platform

Maintaining a re-engineered platform does not differ from maintaining a proactively engineered one, which is why no adaptations are required for the corresponding cost function.

However, since maintenance burdens are often a primary reason to re-engineer a platform from cloned variants, it is key to estimate the corresponding costs. In contrast to the costs of adopting and using a platform, such costs must be compared over the time in which either strategy is used. For example, an organization may decide against a platform if re-engineering it does not pay off within a few years. Since quality issues and design flaws are major pitfalls of software reuse, the additional investments into re-engineering a platform can help an organization to improve its software development. Still, a re-engineered platform comprises more complexity than individual cloned variants, which is why new types of flaws may be introduced that require novel testing capabilities [Engström and Runeson, 2011; Fenske and Schulze, 2015; Ghanam et al., 2012; Mukelabai et al., 2018b; Strüber et al., 2019].

Re-Engineering Costs as Economical Cost Curves

Fixed and variable costs

In economics, cost curves are used to model the costs of producing a product [Dorman, 2014; Eiteman and Guthrie, 1952; Viner, 1932]. We build upon two concepts of cost curves to intuitively relate the re-engineering cost functions of SIMPLE to each other, the defined scenario, and cost estimations:

Fixed costs (C_f) cover all costs that do not depend on the number of products, and thus stay constant during production. We remark that such costs are fixed only for a defined period of time, and later investments may be required to improve production.

Variable costs (C_v) cover all costs of developing a number of products, and may cause a different shape of the cost curve depending on the marginal costs for any new product.

In Equation 3.5, we associate these concepts (top) to the cost functions of SIMPLE (bottom).

$$\begin{aligned} C &= \quad C_f \quad + \quad C_v * n \\ C_{platform} &= \overbrace{C_{org} + C_{cab}} + \overbrace{\sum_{i=1}^{n}(C_{unique}(v_i) + C_{reuse}(v_i))} \end{aligned} \tag{3.5}$$

We can see that organizational costs and developing the core asset base are independent from the number of variants, which is why they represent fixed costs. Costs for developing unique features or reusing features of the platform depend on the number of variants in which those are used, which is why they represent variable costs.

Cost curves

In Figure 3.2, we display simplified cost curves comparing clone & own, an appropriate platform (i.e., one that reaches the *break-even point* to achieve a return on investment), and an inappropriate platform (i.e., one that does not reach the break-even point). The necessary investments to establish a platform are often called *adoption barrier* [Clements and Krueger, 2002; Krüger et al., 2016a] and represent the fixed costs of SIMPLE (ΔC_f). Experiences indicate that the fixed costs can already pay off after proactively developing three variants with a platform compared to developing the same variants with clone & own [McGregor et al., 2002; Pohl et al., 2005; van der Linden et al., 2007]. However, since platforms are usually re-engineered in practice, this assumption may be challenged, considering that the organization already invested into developing variants and then into a re-engineering project. Furthermore, as we exemplify in Figure 3.2, varying investments into a platform (e.g., its quality, reuse of cloned variants) impact the achievable benefits for developing and maintaining variants later on (ΔC_v). As a result, a platform may be an inappropriate strategy, for instance, because the organization invested too much to reach a break-even point or did not invest enough to justify the investments with the gained benefits. So, organizations that (re-)engineer a variant-rich system face the question, *which investments*

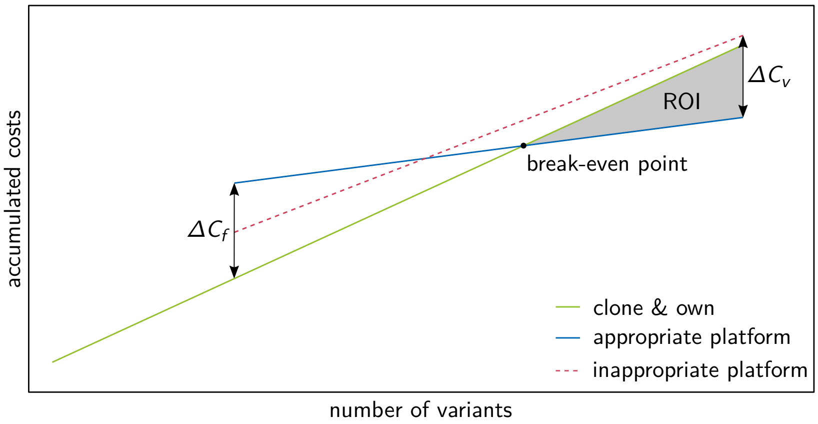


Figure 3.2: Simplified cost curves for (re-)engineering variant-rich systems comparing different scenarios (i.e., clone & own, an appropriate platform that would pay off, and an inappropriate platform that would not pay off).

promise which benefits, and can thus be justified to achieve a return on these investments? Reflecting on the various cost factors as well as missing empirical and systematically elicited data (cf. Section 3.1.1), we can see that further research is needed to understand how to decide this question and guide practitioners.

RO-E$_2$: The SIMPLE Cost Model and Re-Engineering

We discussed SIMPLE to show that the economics of re-engineering a variant-rich system differ mainly in terms of the required investments (fixed costs) involving, for instance, ensuring platform quality, re-engineering existing assets, and improving developers' knowledge. It is key to assess the trade-offs between investing more into the fixed costs and the potential benefits on variable as well as maintenance costs.

3.1.3 RO-E$_3$: Economics and Feature Location

Feature location

As indicated, feature location (i.e., locating the source code that implements a feature) is a core activity while re-engineering a variant-rich system that is usually required because features are not explicitly traced in the source code [Assunção and Vergilio, 2014; Dit et al., 2013; Rubin and Chechik, 2013b; Strüber et al., 2019; Wilde et al., 2001; Xue et al., 2012]. Due to missing traceability and fading knowledge [Krüger and Hebig, 2020; Krüger et al., 2018e; Parnin and Rugaber, 2012], developers have to identify, locate, map, and comprehend the behavior of features as well as their variations among the variants that shall be re-engineered—summarized as feature location in most definitions [Krüger et al., 2019a]. The obtained knowledge helps the organization to scope its platform, decide which features to re-engineer, and refine its economic assessments. Since the program comprehension involved in feature location requires considerable mental effort and time, feature location is considered to be one of the most expensive activities in software engineering [Biggerstaff et al., 1993; Poshyvanyk et al., 2007; Tiarks, 2011; Wang et al., 2013]. Unfortunately, but not surprising, automated feature-location techniques are rarely reliable: it is expensive to impossible to adapt them to domain specifics [Biggerstaff et al., 1993; Kästner et al., 2014] and they lack accuracy [Ji et al., 2015; Krüger et al., 2018b; Wilde et al., 2003]. In the following, we [Krüger et al., 2019a] report a systematic literature review of existing empirical studies on manual feature location to account for these limitations and establish a better understanding of the economical consequences. We connect our insights to those of our previous sub-objectives to motivate our remaining research objectives [Krüger, 2018a; Strüber et al., 2019].

Methodology and Results

Research questions

While automated feature location has been researched extensively [Assunção and Vergilio, 2014; Dit et al., 2013; Razzaq et al., 2018; Rubin and Chechik, 2013b], little and mostly recent research aims to understand how developers actually locate features. To better understand feature location and its impact on the economics of re-engineering a variant-rich system, we conducted a systematic literature review of empirical studies investigating manual feature location. To guide our systematic literature review, we defined two research questions regarding manual feature location:

RQ$_1$ What topics have been investigated in existing studies?

RQ$_2$ What topics have been neglected in existing studies?

Building on these two research questions, we analyzed whether the economics of feature location have been investigated (RQ$_1$), and what topics remain open regarding our other research objectives (RQ$_2$). To answer our research questions, we synthesized the goals and research questions defined in all included studies into topics (e.g., developers' search patterns) using an open-card-like sorting method [Zimmermann, 2016].

Automated search

To identify relevant publications, we conducted an automated search through DBLP, which we last updated on November 13, 2017. Note that DBLP covers publications of most publishers in computer sciences, including ACM, IEEE, Springer, Elsevier, and Wiley. Aiming to capture all relevant publications, we defined the following search string:

> "feature location"

Using this search string, we obtained 271 publications, which we inspected based on title, abstract, and then the whole document to decide whether they were relevant.

Selection criteria

We deemed any publication relevant that fulfilled the following inclusion criteria:

IC$_1$ The publication is written in English.

IC$_2$ The publication has been peer-reviewed.

IC$_3$ The publication reports an empirical study.

IC$_4$ The publication is concerned (at least in parts) with manual feature location.

We did not employ an additional quality assessment, but relied on the peer-review criterion.

Snowballing and check

To complement the selected publications, we used Google Scholar for backwards and forwards snowballing, last updated on November 15, 2017. For every new publication we selected, we again employed snowballing — meaning that we did not stop after a specific number of iterations. After this snowballing, we also checked our set of publications using Scopus on January 26, 2018. This time, we used a more specific search string to see whether we missed any prominent publication:

> manual AND "feature locat*"

With this search, we obtained a total of 37 publications, but no relevant ones that we did not already include. As a result, we are confident that our systematic literature review covers the most important research on manual feature location, providing a reliable foundation to answer our research questions.

Selected publications

We provide an overview of the eight publications we selected in Table 3.2. During our automated search, we found only two publications, with snowballing leading to four more.

Table 3.2: Overview of the eight publications on manual feature location. We combine studies and their extensions (i.e., of Wang et al. [2013] and our own [Krüger et al., 2019c]). The last four columns map the topics we identified to each study.

reference	methodology	loc		subjects	F	Pe	Pr	ST
Wilde et al. [2003]	case study	2,350	Fortran	1	○	●	○	●
Revelle et al. [2005]	multi-case study	2,100 3,000	C Java	2	●	○	●	○
Wang et al. [2011, 2013]	experiment	72,911 2,314 43,957 18,755	Java Java Java Java	20 2 * 18	●	●	●	○
Jordan et al. [2015]	field study	3 mill.	COBOL	2	●	●	○	●
Damevski et al. [2016]	field study	—		$67 + \tilde{6}00$	○	○	●	●
Krüger et al. [2018b, 2019c]	multi-case study	53,015 18,079	C++ Java	2 2	●	○	○	○

F: **F**actors – Pe: **Pe**rformance – Pr: **Pr**ocesses – ST: **S**earch **T**ools

We remark that we added our own study involving manual feature location [Krüger et al., 2018b, 2019c] that had been accepted, but not indexed. In Table 3.2, we can see that most studies involved authors' experiences or work (i.e., case studies), resulting in a small number of subjects. Other researchers observed developers in practice (i.e., field studies), with only one set of actual experiments. The small number of publications on manual feature location and their focus on individual experiences highlight that we are likely missing a detailed understanding of how developers locate features in practice.

Findings and Discussion

Investigated topics

Next, we are concerned with answering RQ_1. To this end, we identified identical and related terms from each publication to derive topics. For instance, several studies referred to *search tools*, *factors*, or *actions*. We show our mapping of topics and publications in the last four columns of Table 3.2. In the following, we describe all topics with a focus on those related to this dissertation (i.e., factors, processes).

Factors summarize all influences (e.g., system properties, developer characteristics) that impact feature location — representing cost factors in our conceptual framework (cf. Figure 3.1). Wang et al. [2013] provide the most detailed insights into such factors based on their experiments, defining three different types of factors ordered from most to least important: (1) human factors, such as knowledge, experience, or preferences; (2) task properties, such as the system under investigation or feature that shall be located; and (3) in-process feedback, such as the results of a search query. As we can see, human factors and particularly knowledge have been identified to have the greatest impact on feature location, which is also supported by the studies of Revelle et al. [2005] and Jordan et al. [2015].

Performance is concerned with the effectiveness, pros, and cons of manual feature location, for instance, compared to automated techniques. Unfortunately, this topic is heavily mingled with the other topics, which complicates deriving concrete insights regarding the effectiveness of manual feature location with respect to specific factors.

Processes describe how developers execute manual feature location (relating to activities in our conceptual model). The studies agree that feature location is a loop between *searching*, *extending*, and *validating* feature seeds. Regarding the search and extension of feature seeds, Wang et al. [2013] further identify three different patterns: (1) information retrieval-based patterns involve search tools that allow developers to locate keywords they deem relevant; (2) execution-based patterns rely on developers executing the program with a specific feature activated and setting suitable breakpoints; and (3) exploration-based patterns refer to developers following static dependencies in the code, such as method calls. In the study of Damevski et al. [2016], 97 % of the subjects relied on information retrieval-based patterns and only 3 % used execution-based patterns to extend seeds. Both patterns require developers to have or obtain a deeper understanding of the system and its domain in order to know suitable keywords or set breakpoints.

Search Tools are used by developers during feature location, for example, to identify seeds based on keywords. The findings regarding this topic are inconclusive, but seem to support the argument that automated feature-location techniques lack effectiveness and are expensive to adopt to domain specifics.

In summary, we can see that particularly the process of manual feature location has been investigated in detail. Interestingly, several results are inconclusive or are not detailed enough to actually derive a deeper understanding regarding the economics of feature location. Still, we found supportive evidence that feature location is an expensive activity while re-engineering variant-rich systems, and is heavily knowledge dependent.

Neglected topics

As described, several of the topics we identified have not been investigated in great detail. Next, we discuss such open gaps to address RQ_2 and detail our motivation to tackle our remaining research objectives.

Economics have not been covered by any of the studies we investigated (besides few reports of the time spent), particularly not for re-engineering variant-rich systems. Among other issues, the missing knowledge regarding such economics prevents us from understanding what activities are more expensive, require more support, or can be facilitated with what techniques (e.g., feature traceability). So, we do not have enough empirical data to actually understand the challenges and problems of locating features in practice, hampering software maintenance and re-engineering.

Factor Studies in the selected publications are limited to identifying factors that impact feature location, but without analyzing any of these factors in detail. For instance, most studies agree that knowledge and experience have the greatest impact on feature location, but the effect sizes are unclear. Better understanding such factors, not only for feature location, but whole re-engineering projects, would provide guidance for organizations to decide where to invest. For instance, an organization may decide to invest more into knowledge recovery to reduce the later costs of re-engineering and maintaining a platform from cloned variants.

Comparisons to Automated Techniques have only been conducted by Wilde et al. [2003] in a small setup. We argue that more extensive analyses and comparisons of manual feature location are needed to properly understand developers' tool needs and potentially improve automated techniques.

Search Tools studies have been mostly limited to simple tools that allow to search keywords in source code. Due to new technologies (e.g., searches in version-control systems or software-hosting platforms), novel studies that utilize such technologies for feature location are highly valuable to reflect modern practices.

Overall, it seems that the economic impact of feature location is still not well understood and requires more analyses — considering that it is arguably the most expensive and challenging activity needed to initiate the re-engineering of a variant-rich system.

Decision support

To improve an organization's confidence whether to re-engineer a platform or not, decision support would be helpful, but requires further research [Krüger, 2018a]. We found that the topics investigated as well as neglected for feature location align to our research objectives. In particular, we argue that practitioners require a deeper understanding of the economics of (re-)engineering variant-rich systems based on reliable empirical data (**RO-E**). We further found that especially the knowledge factor seems to impact the economics of variant-rich systems and feature location (**RO-K**), which can be tackled by establishing feature traceability (**RO-T**). Finally, even though we have an understanding of developers processes for feature location, we are missing reliable insights for (re-)engineering variant-rich systems (**RO-P**). As a consequence, the existing body-of-knowledge regarding feature location reflects well on the challenges of whole re-engineering projects and motivate our research objectives further.

RO-E$_3$: Economics and Feature Location

We found that the economics and processes of manual feature location have been studied to some extent. The results highlight that both properties are highly knowledge-dependent and require further analyses.

Threats to Validity

Search strategy

Threats to the internal validity of our systematic literature review are that we focused on one research area, used rather narrow search strings, and searched only two databases. Consequently, we may have missed publications that are closely related to manual feature location, but use a different terminology or are not indexed in these databases. We mitigated this threat by employing snowballing. Moreover, we consciously used two databases that index various publishers and peer-reviewed publications, ensuring a certain quality.

Derived topics

We derived topics that are investigated or missing in the identified studies. Other researchers may identify other topics, particularly with respect to those requiring further research. However, we focused on the topics of this dissertation, and thus the practical and economical implications of feature location. While this threatens the external validity, other researchers can replicate our analysis and derive additional topics by investigating the selected publications.

3.2 Clone & Own Versus Platform

Studying reuse economics

Software reuse has been intensively studied in the last three decades [Assunção et al., 2017; Barros-Justo et al., 2018; Bombonatti et al., 2016; C and Chandrasekaran, 2017; Fenske et al., 2013; Heradio et al., 2016; Laguna and Crespo, 2013; Rabiser et al., 2018], involving different techniques (e.g., testing, variability mechanisms, variability modeling) for (re-)engineering variant-rich systems. Unfortunately, the economics of software reuse have not been studied based on systematically elicited data, but are usually only mentioned as a motivation or a byproduct. In this section, we [Krüger and Berger, 2020b] present a systematic study of reuse economics by investigating the activities, costs, cost factors, and benefits (e.g., monetary savings) of developing a new variant with either reuse strategy — clone & own or platform engineering. For this purpose, we collected and triangulated data from two sources: First, we conducted an interview survey with 28 practitioners of a large organization (i.e., *Axis AB*) that employs both reuse strategies. Second, we used a systematic literature review to complement the interviews with empirical data from existing case studies and experience reports. Using these two sources, we mitigated the problem that costs in software engineering can hardly be quantified and assigned to specific activities [Heemstra, 1992; Jørgensen, 2014; Jørgensen and Moløkken-Østvold, 2004; Trendowicz, 2013]. More precisely, we combine two widely used, reliable cost estimation methods — expert judgment (i.e., interviews) and analogies to historical data (i.e., systematic literature review) — to improve the validity of our data and elicit a reasonable basis for understanding the economics of software reuse [Boehm, 1984; Heemstra, 1992; Jørgensen, 2004, 2014; Jørgensen and Boehm, 2009; Moløkken and Jørgensen, 2003]. Our study is of the rare breed that tackles the problem of software economics based on empirical evidence reported for more than 100 organizations, providing novel insights that confirm and refute established hypotheses.

Section contributions

In detail, we contribute a dataset of systematically elicited, empirical data on the economics of engineering variants with clone & own and a platform, which we use to derive evidence for confirming or refuting established hypotheses on software reuse. For this purpose, we defined three sub-objectives to **RO-E**:

RO-E$_4$ *Identify the processes and activities of reusing software for a new variant.*

While collaborating with practitioners [Kuiter et al., 2018b; Lindohf et al., 2021; Nešić et al., 2019], we experienced that the concepts of pure clone & own or a full-fledged platform are rarely employed. Instead, most organizations combine both strategies or employ the one that seems more convenient for a new variant. Due to these experiences, we first aimed to understand the processes and activities of software reuse employed in practice.

RO-E$_5$ *Understand the costs associated with the activities identified.*

Second, we aimed to understand the costs the identified activities cause in either reuse strategy. Differences in these costs provide an understanding of which activities are more costly, and thus challenging, for which reuse strategy. So, the results can help to decide which reuse strategy to employ, and to identify what activities require more research and tool support.

RO-E$_6$ *Determine the impact of cost factors on either reuse strategy.*

Finally, we aimed to investigate the cost factors that impact either reuse strategy. Understanding cost factors helps to tune reuse strategies by understanding which cost factors may be altered to potentially reduce costs (e.g., investing in higher platform quality to achieve more long-term benefits). For this purpose, we analyzed cost factors based on economical costs and benefits (i.e., reduced costs).

Our data and interview guides are available as an evaluated open-access replication package.[3] Next, we first report how we elicited data based on our interview survey (Section 3.2.1) and systematic literature review (Section 3.2.2). Then, we discuss potential threats to validity of our methodology in Section 3.2.6. Finally, we present and discuss the results for each sub-objective individually in Section 3.2.3, Section 3.2.4, and Section 3.2.5.

Discussion structure

We remark that we use median values to discuss the elicited data to provide intuitive examples for costs and savings that have been reported. However, we have to be careful with these, since actual costs heavily depend on an organization's and a project's properties, which is reflected by the large ranges some of our data spans. Since our data aligns overall, we still argue that these examples are good intuitions to compare the reuse strategies. In our discussion, we display how our data relates to established hypotheses (e.g., by Knauber et al. [2002]) on clone & own and platform engineering by denoting confirmations as 👍, refutations as 👎, and inconclusive results as 👉.

[3] https://doi.org/10.5281/zenodo.3993789

Table 3.3: Overview of the interviews we conducted at Axis on software-reuse economics.

id	phase	hours	interviewees	strategy	data
I_1	EXP	$\sim$0.5	system architect	platform	qualitative
I_2	EXP	$\sim$0.5	software engineer	c & o	qualitative
I_3	EXP	$\sim$0.5	release engineer	platform	qualitative
I_4	EXP	$\sim$0.5	technical lead	c & o → platform	qualitative
I_5	EXP	$\sim$0.5	technical lead	c & o → platform	qualitative
I_6	EXP	$\sim$0.5	project manager	c & o → platform	qualitative
I_7	EXP	$\sim$0.5	2 software engineers	c & o + platform	qualitative
I_8	PD	>3	firmware architect	c & o + platform	qualitative
I_9	PD	>3	software engineer	platform	qualitative
I_{10}	PD	$\sim$1	system architect	platform	qualitative
I_{11}	PD	$\sim$1	software engineer	c & o + platform	qualitative
I_{12}	PD	$\sim$1	2 software engineers	c & o + platform	qualitative
I_{13}	CA	$\sim$1	firmware developer	c & o + platform	qualitative & quantitative
I_{14}	CA	$\sim$1	software developer	c & o	qualitative & quantitative
I_{15}	CA	$\sim$1	technical lead	c & o + platform	qualitative & quantitative
I_{16}	CA	$\sim$1	technical lead	c & o + platform	qualitative & quantitative
I_{17}	CA	$\sim$1	software developer	c & o + platform	qualitative & quantitative
I_{18}	CA	$\sim$1	technical lead	platform	qualitative & quantitative
I_{19}	CA	$\sim$1	system architect	c & o	qualitative & quantitative
I_{20}	CA	$\sim$1	technical lead	c & o + platform	qualitative & quantitative
I_{21}	CA	$\sim$1	software developer	platform	qualitative & quantitative
I_{22}	CA	$\sim$1	system architect	c & o	qualitative & quantitative
I_{23}	CA	$\sim$1	software developer	c & o + platform	qualitative & quantitative
I_{24}	CA	$\sim$1	software developer	c & o + platform	qualitative & quantitative
I_{25}	CA	$\sim$1	firmware architect	c & o + platform	qualitative
I_{26}	CA	$\sim$1	software architect	c & o → platform	qualitative

EXP: **EXP**loration – PD: **P**rocess **D**efinition – CA: **C**ost **A**ssessment
c & o: clone & own – +: both strategies – →: re-engineering

3.2.1 Eliciting Data with an Interview Survey

Interview survey

Initially, we conducted an interview survey with practitioners to elicit economical data for addressing our sub-objectives. We summarize these interviews in Table 3.3, and use the displayed identifiers as references within this section.

Interviewees

Interviewees

Our interview survey built on a collaboration with *Axis Communications AB*. Axis is a large, international organization that develops network equipment, particularly network cameras and various server infrastructures. We collaborated closely with two contacts: A system architect (I_8) and a product manager (I_9) who were interested in analyzing the economics of their reuse strategies, and who have an overview understanding of Axis' practices. With our two contacts, we identified 26 interviewees that we found to be knowledgeable experts. As we can see in Table 3.3, these interviewees had different roles (e.g., software engineers, technical leads), which allowed us to obtain a broader understanding of the reuse practices employed at Axis. Most of our interviewees developed Axis' large portfolio of network cameras and stated to have between three to more than 20 years of experiences with software reuse in their current position (excluding previous employments). While we aimed to interview each interviewee individually, we twice interviewed two of them in the same interview (i.e., I_7, I_{12}). We regularly discussed our findings and goals with our two contacts.

Design of the Interviews

Interview survey structure

We conducted 26 interviews throughout three consecutive phases: exploration, process definition, and cost assessment. For the first two phases, we had open discussions with semi-structured guides, and took notes. For the last phase, we constructed a semi-structured interview guide based on the obtained insights and our research questions. We recorded and transcribed 13 of these interviews, and took notes for all of them (one interviewee asked us not to record the interview). The three phases were structured as follows:

Exploration: We started with seven interviews of roughly half an hour each. In this phase, we aimed to explore the development processes and reuse practices at Axis. Based on our insights, we discussed with our contacts how to design the remaining study.

Process definition: We conducted five semi-structured interviews to define the concrete development processes we identified. While most of these interviews took around one hour, we discussed our findings extensively with our contacts, which is why those interviews took far longer. In the end, we constructed a unified reuse process (i.e., combining clone & own and platform engineering) that comprises 10 activities.

Cost assessment: Finally, we conducted 14 semi-structured interviews to collect quantitative and qualitative data on the reuse economics at Axis. With each interviewee, we iterated through the process we constructed and asked them to distribute the overall development costs (in percent) of a new variant across the 10 activities. Finally, we asked each interviewee to assess the impact of cost factors we identified in the previous phases on a seven-point Liker scale, ranging from strongly reduces to strongly increases costs. We elicited data for both reuse strategies if an interviewee had worked with both, and asked particularly those to explain the differences they experienced.

We allowed our interviewees to look up data during the interviews to enrich their knowledge, for instance, on the size of the platform. Still, most interviewees relied on their expertise.

Elicited Data

Survey data

During the first two phases, we elicited qualitative data (i.e., natural-language descriptions) on the development processes, reuse practices, and cost factors at Axis. In the last phase, we elicited qualitative and quantitative data. Namely, we elicited one or two (if experienced with both reuse strategies) datasets for every interviewee, comprising estimated distributions (percent for each activity) of the costs of developing a new variant as well as assessments of cost factors' impact (Likert ranking). We could not fully elicit this data, because:

- One interviewee (I_{23}) was not confident in estimating cost distributions, since they always joined running projects.
- For the same reason, one interviewee (I_{24}) was not confident in assessing the costs of the first two activities we identified (i.e., SV and DR in Table 3.5).
- Two interviewees (I_{25}, I_{26}) did not develop variants, but maintained Axis' platform—which is why we did not elicit quantitative data, but obtained qualitative insights.

We elicited eight and seven datasets for clone & own and platform engineering, respectively. For cost factors, we elicited one additional dataset for each reuse strategy from I_{23}. Still, we miss three values of individual cost factors for which our interviewees were not confident to provide an assessment (team size once for either strategy, number of teams for clone & own).

Table 3.4: Overview of the 58 publications on economics of variant-rich systems.

source	reference	rm	organizations (subjects)	strategies
V	Incorvaia et al. [1990]	MCS	5	c & o + platform
K	Bowen [1992]	ER	IBM	platform
	Lim [1994]	MCS	HP	platform
K	Henry and Faller [1995]	ER	Matra Cap Systems	c & o
R	Brownsword and Clements [1996]	CS	CelsiusTech	platform
V	Ganz and Layes [1998]	ER	ABB	platform
	Rine and Sonnemann [1998]	IS	83 (109)	platform
R	Bass et al. [1999]	ERs	Cummins, Dt. Bank, HP, Nokia, Philips, US NRO	platform
V	Lee et al. [2000]	ER	LG	c & o → platform
KR	Clements and Northrop [2001]	ERs	Cummins, CelsiusTech, Market Maker, US NRO	platform
R	Clements et al. [2001]	ER	US NRO	platform
V	Frakes and Succi [2001]	QE	4 (4)	c & o
R	Gacek et al. [2001]	ER	Market Maker	platform
R	Clements and Northrop [2002]	IS	Salion	platform
R	Cohen et al. [2002]	ER	DoD-NUWC	platform
R	Buhrdorf et al. [2003]	ER	Salion	platform
	Ebert and Smouts [2003]	ER	Alcatel	c & o → platform
	Faust and Verhoef [2003]	CS	Dt. Bank	c & o → platform
V	Bergey et al. [2004]	ER	Argon	platform
	Staples and Hill [2004]	CS	Dialect Solutions	c & o → platform
R	BigLever Software, Inc. [2005]	ER	Engenio	c & o → platform
R	Clements and Bergey [2005]	ER	TAPO, RCE	c & o → platform
K	Pohl et al. [2005]	ERs	HP, Lucent, Siemens	c & o → platform, platform
KR	Hetrick et al. [2006]	ER	Engenio	c & o → platform
	Kolb et al. [2006]	CS	Testo AG	c & o → platform
	Slyngstad et al. [2006]	IS	Statoil ASA (16)	platform
V	Jensen [2007]	ER	OTs	c & o → platform
R	Jepsen et al. [2007]	ER	Danfoss	c & o → platform
K	van der Linden et al. [2007]	ERs	AKVAsmart, Bosch, DNV, Market Maker	platform
	Jansen et al. [2008]	MCS	2	c & o
K	Kapser and Godfrey [2008]	MCS	Apache, Gnumeric	c & o
R	Krueger et al. [2008]	ER	HomeAway	platform
	Lucrédio et al. [2008]	S	57 (57)	platform
	Sharma et al. [2008]	IS	1 (11)	platform
	Jensen [2009]	ER	Overwatch Systems	c & o → platform
R	Li and Chang [2009]	ER	FISCAN	c & o → platform
R	Li and Weiss [2011]	ER	FISCAN	c & o → platform
R	Otsuka et al. [2011]	ER	Fujitsu QNET	c & o → platform
V	Quilty and Cinnéide [2011]	ER	ORisk	platform
R	Zhang et al. [2011]	CS	Alcatel-Lucent	c & o → platform
K	Dubinsky et al. [2013]	IS	3 (11)	c & o + platform
R	Lanman et al. [2013]	ER	US Army	platform
K	van der Linden [2013]	ER	Philips	platform
	Bauer et al. [2014]	IS	Google (49)	c & o
R	Clements et al. [2014]	ER	General Dynamics, Lockheed	platform
R	Dillon et al. [2014]	ER	US Amry	c & o → platform
K	Duc et al. [2014]	IS	Multiple (10)	c & o
R	Gregg et al. [2014]	ER	DoD	c & o → platform
R	Gregg et al. [2015]	CS	DoD	c & o → platform
M	Bauer and Vetrò [2016]	IS	Google, 1 (108)	platform, c & o + platform
KM	Bogart et al. [2016]	IS	Eclipse, R, node.js (28)	platform
KMR	Fogdal et al. [2016]	ER	Danfoss	c & o → platform
M	Nagamine et al. [2016]	CS	Mitsubishi	platform
K	Walker and Cottrell [2016]	IS	Multiple (59)	c & o
M	Cortiñas et al. [2017]	ER	Enxenio	platform
KM	Kuiter et al. [2018b]	ER	TA, HCP	c & o → platform
KMR	Martinez et al. [2018]	MCS	OSS (6)	c & o → platform
M	Ham and Lim [2019]	ER	Samsung	c & o → platform

K: **K**nowledge – R: **R**esources – M: **M**anual search – V: **V**alidation with related work
CS: **C**ase **S**tudy – ER: **E**xperience **R**eport – IS: **I**nterview **S**urvey – MCS: **M**ulti-**C**ase **S**tudy – S: **S**urvey – QE: **Q**uasi-**E**xperiment
rm: **r**esearch **m**ethod – c & o: clone & own – +: both strategies – →: re-engineering

3.2.2 Eliciting Data with a Systematic Literature Review

Systematic literature review

Besides our interview survey, we conducted a systematic literature review [Kitchenham et al., 2015] to elicit data and experiences that have been reported for other organizations. For this purpose, we focused on a qualitative analysis of the identified publications, omitting the typical publication statistics of systematic literature reviews. We provide an overview of all 58 publications that we selected in Table 3.4.

Search Strategy

We conducted our systematic literature review based on five different sources:

Literature sources

Source$_1$ – Knowledge: First, we identified publications that we deemed relevant based on our knowledge. After applying our selection criteria (explained shortly), we included 15 publications for this study. We mark these publications with a **K** in Table 3.4.

Source$_2$ – Manual Search: We investigated particularly recent publications to address our sub-objectives. To this end, we manually searched through the last three completed editions (in July 2019) of relevant journals and conferences. Via DBLP, we considered the 2016–18 editions of ESE, ESEC/FSE, ICSME, IST, JSS, IEEE Software, SPE, SPLC, TOSEM, and TSE, as well as the 2017–19 editions of ICSE, ICSR, and VaMoS. We mark the eight publication we included from this manual search with an **M** in Table 3.4.

Source$_3$ – Resources: We know of four collections from the software product-line community that include real-world case studies on (re-)engineering experiences. The collections we considered are the (1) ESPLA catalog [Martinez et al., 2017], (2) SEI technical reports on software product lines,[4] (3) BigLever case-study reports,[5] and (4) SPLC Hall of Fame.[6] We mark the 24 publications we included from these resources with an **R** in Table 3.4.

Source$_4$ – Backwards Snowballing: From the previous three sources, we included 38 publications and used these as starting set for backwards snowballing. We did not conduct a defined number of iterations, but snowballed on every newly included publication. After all iterations, we included 12 publications (unmarked in Table 3.4).

Source$_5$ – Validation with Related Work: We validated the completeness of our systematic literature review against related overviews of case studies and experience reports on (re-)engineering variant-rich systems. Some introductions to software product-line engineering include a number of practice reports [Krueger and Clements, 2013; Northrop, 2002; Pohl et al., 2005; van der Linden et al., 2007]. Often, we already included the original publications from which these practice reports stem from, so we used these overviews to verify the data and included only new or updated data. Barros-Justo et al. [2018] present a systematic literature review on reuse benefits that have been transferred to industry. In their mapping study, Bombonatti et al. [2016] investigate the impact of software reuse on non-functional properties. Mohagheghi and Conradi [2007] conducted a systematic literature review on the benefits reported for software reuse in industrial contexts. Finally, the Software Engineering Institute [2018] published a catalog of software product-line engineering publications on various topics. Our study is complementary to such overviews, since we aim to collect and synthesize empirical data on the costs of (re-)engineering variant-rich systems— whereas none of these overviews presents such an analysis. We remark that we employed snowballing on the publications we included from these overviews, too. In Table 3.4, we mark the seven newly included publications with a **V**.

Since we did not have access to all publications we found, we excluded a minority of potentially relevant publications. Note that we did not employ an automated search, since these are problematic to replicate and would arguably need to be too broad (e.g., involving

[4] https://www.sei.cmu.edu/publications/technical-papers

[5] https://biglever.com/learn-more/customer-case-studies/

[6] https://splc.net/fame.html

any publication mentioning software reuse and costs) to even retrieve all results [Kitchenham et al., 2015; Krüger et al., 2020c; Shakeel et al., 2018].

Selection Criteria

Literature selection

For each publication, we checked the following four inclusion criteria:

IC_1 The publication is written in English.

IC_2 The publication describes empirical findings on the costs of reusing software.

IC_3 The publication is concerned with clone & own, platform engineering, both, or the migration from one to the other.

IC_4 The publication reports experiences or actual data on the costs, not only estimates for planning the (re-)engineering of a variant-rich system.

Furthermore, we checked the following two exclusion criteria:

EC_1 The publication does not clarify whether the reuse built on clone & own or a platform.

EC_2 The publication only cites costs reported in previous publications without providing new or updated data on its own (cf. $Source_5$).

To check EC_1, we read each publication and classified the reuse strategy based on keywords. For instance, using components or the C preprocessor implies a platform, while copying systems or modules (i.e., not just copy & paste) implies clone & own.

Quality Assessment

Quality assessment

The economics of software reuse are usually mentioned as a motivation or byproduct of experience reports and case studies. So, we were interested in data that is rarely collected systematically, and not reported prominently. For this reason, we decided to skip a quality assessment, since the included publications have no common goal or research methodology that we could base that assessment on. Moreover, for our goal of structuring and classifying previous experiences, a quality assessment is also less important [Kitchenham et al., 2015].

Data Extraction

Data extraction

From each included publication, we extracted standard bibliographic data, namely authors, title, venue, and year. To address our sub-objectives, we further extracted the reuse strategy employed (cf. Selection Criteria), the research method used, the organizations as well as subjects (e.g., for surveys) involved, as well as described cost factors, qualitative insights, and quantitative data. In detail, we identified instances in the publications that concretely state experienced or measured costs, benefits, and problems of either reuse strategy. We used a semi-structured document to collect the data and added references to trace each instance back to its source. To identify synonyms and analyze our data, we relied on an open-card-like sorting method [Zimmermann, 2016].

Elicited Data

Elicited data

In total, we included 58 publications for our analysis. We expected, and can see in Table 3.4, that only few (10) publications report costs and benefits of clone & own [Jansen et al., 2008; Kulkarni and Varma, 2016]. Moreover, most of these publications solely discuss the pros and cons of this reuse strategy, while only four provide a total of five instances of quantified data regarding its economics [Frakes and Succi, 2001; Henry and Faller, 1995; Incorvaia et al., 1990; Otsuka et al., 2011]. Far more publications are concerned with the re-engineering from clone & own towards a platform (23) and the economics of a platform itself (28).

Table 3.5: Overview of the activities we elicited at Axis.

id	**activity**
SV	**S**coping the **V**ariant based on customer requirements
DR	**D**efining the **R**equirements to specify what must be implemented
FEV	**F**inding an **E**xisting **V**ariant that is similar to the requirements
DF	**D**esigning the **F**eatures needed to implement the requirements
PI	**P**lanning the **I**mplementation of the features
IF	**I**mplementing the **F**eatures of the scoped variant
QA	**Q**uality **A**ssuring the implemented variant
BF	**B**ug **F**ixing the variant
PBF	**P**ropagating **B**ug **F**ixes to other variants and/or the platform
CD	**C**oordinating the **D**evelopment between developers and teams

3.2.3 RO-E$_4$: Development Processes and Activities

Development process

We summarize the 10 core activities we identified at Axis in Table 3.5, and the unified development process we constructed in Figure 3.3. This process integrates clone & own and platform engineering, since we found that Axis employs the same activities for each new variant, only with varying details.

Results

Results development process

Research usually assumes that organizations employ either clone & own or platform engineering. In contrast, we found that Axis employs a combination of both strategies. To develop a new variant, Axis starts by scoping the required features based on novel customer requirements. Using this scoping, a variant (or the complete platform) that is similar to the requirements is derived into a separate clone. The developers design and implement the features on this clone until they can release the new variant. At this point, Axis has to make a core decision: On the one side, the developers and platform engineers can immediately integrate the new variant into the platform to incorporate its features. We referred to such variants as *short-living* clones (i.e., platform engineering), which is most commonly applied at Axis. On the other side, the new variant may be kept outside of the platform, to which we referred to as *long-living clone* (i.e., clone & own). Such variants may diverge from the platform, increasing the integration costs — if the variant is integrated at all. The main difference between these strategies at Axis is that short-living clones are maintained by platform maintainers, while the long-living clones are maintained be the variant developers. Despite such differences, both reuse strategies build on the same 10 activities we display in Table 3.5, and we use the identifiers to refer to these activities.

Discussion

Integrated process

Similar processes to ours have been identified for other organizations [Dubinsky et al., 2013; Krüger et al., 2020d; Varnell-Sarjeant et al., 2015] and open-source communities [Krüger et al., 2018b, 2019c; Stănciulescu et al., 2015], but they are rarely considered relevant in research 👎. So, it may be problematic to directly transfer research, for example, on migrating clone & own into a full-fledged platform, to industry. Still, these findings indicate that the combination of both reuse strategies occurs regularly in practice. Particularly, it seems that organizations developing a variant-rich system (should) strive for platform engineering to some degree, for instance, employing clone & own with a clone-management framework [Rubin et al., 2015] or tools for change propagation [Pfofe et al., 2016].

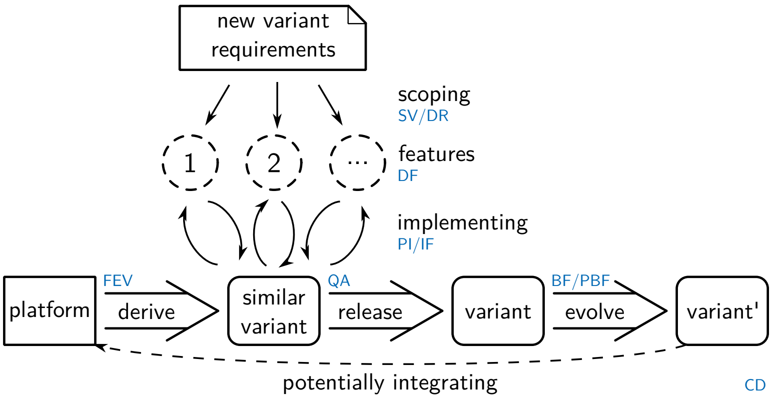


Figure 3.3: Axis' development process with blue abbreviations relating to Table 3.5 (CD impacts all activities).

Selecting a process

Considering that organizations employ an integrated reuse process, the question seems not to be whether to employ pure clone & own or platform engineering, but for what variant which strategy is more beneficial? For example, Axis rapidly integrates short-living clones, often before any major changes happened on the platform. These clones are usually used to implement well-defined features that can benefit several variants and customers. Long-living clones, in contrast, are sometimes only integrated after years or even not at all. We found that Axis employs this form of clone & own mostly to advance independent variants into completely new markets or to test innovative features. However, an apparent problem is that if such features are considered highly valuable and shall be integrated after long co-evolution, this becomes a far more expensive process 👍.

RO-E$_4$: Development Process and Activities

We constructed a variant-development process (cf. Figure 3.3) with 10 activities (cf. Table 3.5) that integrates clone & own and platform engineering. One strategy is selected for a concrete variant, but both may be employed for the same variant-rich system.

3.2.4 RO-E$_5$: Costs of Activities

Development costs

Next, we analyze the costs of clone & own and platform engineering. For this purpose, we use qualitative insights and relative, quantitative data because: (1) Eliciting precise data on development costs is problematic, which we mitigate with qualitative insights. (2) Absolute values are not representative, since they can be in completely different orders of magnitude for a specific organization (e.g., large organization versus start-up). Also, we combine the results of our interview survey and systematic literature review to improve our quantification. (3) We avoid repetitions and clarify relations or discrepancies in the data.

Results

Cost distributions

In Figure 3.4, we show how our interviewees assessed the cost distributions for developing a new variant with clone & own or a platform. Since not all interviewees reported a total of 100 % (min 68 %, max 133 %, avg 99.6 %), we normalized the values to compare them. To mitigate biases, we verified the normalized distribution with each interviewee and asked whether we forgot to elicit important activities. Some of our interviewees mentioned integration, but we purposefully excluded this assessment at this point.

Data in the literature

From our systematic literature review, we extracted quantitative data on the costs of activities, benefits, and total costs of platform engineering, which we show in Figure 3.5.

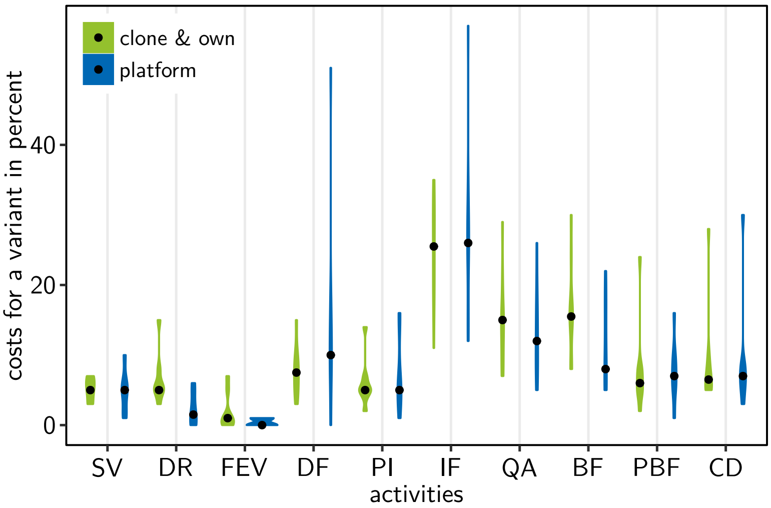


Figure 3.4: Relative cost distributions for developing a variant (activities in Table 3.5) with clone & own or a platform, elicited with our interview survey.

The numbers in parentheses show how many values we found (we display only those with three or more occurrences), and the dots indicate medians. In total, we found quantitative data for three activities (left side): **feature development** (FD), **quality assurance** (QA), and **variant development** (VD). We also found data for three benefits of platform engineering (in the middle), namely **identified bugs** (IB), **staffing** (S), and **time-to-market** (TM). On the right of Figure 3.5, we display the five data points reported on the total cost savings of using a platform — which indicate that implementing new variants upon an established platform can save around 52 % of the costs. We found that the values reported for re-engineering (compared to clone & own) and platform engineering (compared to individual systems) do not differ much, and thus summarized them for simplicity.

Discussion

Paragraph structure

Next, we discuss our data based on related activities (e.g., to set up development). To map the data, we reference the abbreviations in Table 3.5 and the previous paragraph.

Set up development

SV, DR, FEV. For developing a new variant from their platform, one interviewee stated:

> *"That's something that you understand way in the beginning when you get the requirement[s] for the project. You understand now, it's a derivative of this one, which will be very obvious [...]"*

We can see in Figure 3.4 that this statement aligns to all activities related to setting up the development of a variant. Defining requirements (DR) and finding a similar variant (FEV) are less expensive for platform engineering than for clone & own, while the initial scoping (SV) is similarly costly for both 👍. Also, we find confirmations in our systematic literature review, including that a platform can improve developers' knowledge on the variability of a variant-rich system [Bowen, 1992; Clements and Northrop, 2001; Cohen et al., 2002; Faust and Verhoef, 2003], that this knowledge is key for clone & own [Bauer et al., 2014; Dillon et al., 2014; Duc et al., 2014; Faust and Verhoef, 2003], and that scoping and finding variants is problematic for clone & own [Dubinsky et al., 2013; Faust and Verhoef, 2003].

Developing a variant

DF, PI, IF, FD, VD. Clone & own can already considerably reduce the development costs for a new variant [Bauer and Vetrò, 2016; Incorvaia et al., 1990; Kapser and Godfrey, 2008; Walker and Cottrell, 2016] 👍. For example, Henry and Faller [1995] report that clone & own

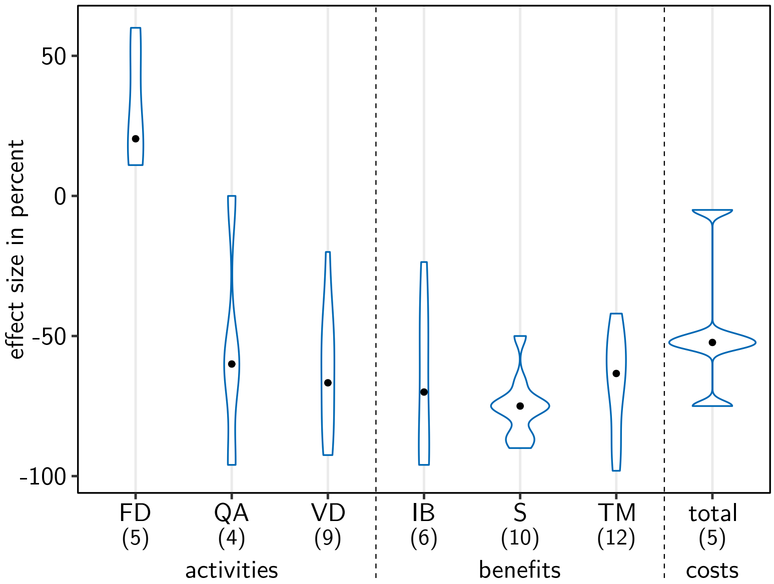


Figure 3.5: Effects of platform engineering on activities' costs (left, **RO-E$_5$**), benefits (middle, **RO-E$_6$**), and total development costs (right) elicited from our systematic literature review for re-engineering or proactively adopting a platform

reduces development costs by 35 %. Still, also confirming established hypotheses, most insights from our systematic literature review suggest that platform engineering reduces development costs even further [Bauer and Vetrò, 2016; Bergey et al., 2004; Brownsword and Clements, 1996; Buhrdorf et al., 2003; Clements and Northrop, 2001, 2002; Clements et al., 2014; Cohen et al., 2002; Quilty and Cinnéide, 2011; Sharma et al., 2008; Slyngstad et al., 2006; van der Linden, 2013] 👍. We can see in Figure 3.5 that the nine studies reporting corresponding data (VD) indicate median savings of around 67 %

Developing features

Besides such benefits, we also found confirmations for the hypothesis that developing new features for reuse (FD) is usually more expensive than developing them for a single variant [Bergey et al., 2004; Lim, 1994; van der Linden, 2013] 👍. The data we extracted from five publications suggests a median increase of approximately 20 %. Our interviewees confirm this tendency: We can see in Figure 3.4 that designing (DF) and implementing (IF) features is considered more expensive for platform engineering than for clone & own, with drastic outliers towards high expenses. Consequently, a platform will only pay off if its features are reused in multiple variants. One of our interviewees summarized this insight by describing the need to align a feature's implementation to the platform architecture:

> *"For short-living clones, we have to design [...] it to be able to be used by others. A long-lived clone, with that, we can ignore that."*

Efficiency

Five publications indicate that an organization can develop more features with a platform, but they do not report concrete costs for a single feature [BigLever Software, Inc., 2005; Ebert and Smouts, 2003; Hetrick et al., 2006; Staples and Hill, 2004] 👍. For example, Fogdal et al. [2016] describe that Danfoss could develop more than 2,500 feature in a year instead of fewer than 300 before adopting platform engineering. Unfortunately, it is unclear where this benefit originated from. Still, seeing that feature development is more expensive for a platform, it must be caused by other factors (e.g., more reuse, higher software quality).

Propagating bugs and fixes

BF, PBF, QA. Researchers usually argue that propagating bug fixes (PBF) to other variants is a major challenge of clone & own. This is confirmed by several publications [BigLever Software, Inc., 2005; Dillon et al., 2014; Dubinsky et al., 2013; Faust and Verhoef, 2003; Fogdal et al., 2016; Hetrick et al., 2006; Kuiter et al., 2018b; Li and Chang, 2009] — indicating that longer co-evolution of variants is the problem, requiring that bug fixes are propagated and also adapted to the other variant 👍. Platforms, in contrast, are challenging to test in their entirety [Bauer and Vetrò, 2016; Bogart et al., 2016; Sharma et al., 2008], but argued not to require such propagation [Ebert and Smouts, 2003; Staples and Hill, 2004].

Propagating in a platform

Interestingly, our data is contradicting this argumentation. In Figure 3.4, we can see that our interviewees think bug fixing (BF) is more expensive in clone & own, while propagating the fixes is more costly for platform engineering 👎. One interviewee explained:

> *"Propagate bug fixes, of course, is longer for the short-living clones because we would actually have to do it. For long-living clones, we don't do it at all."*

Obviously, propagating bug fixes is important, but researchers must re-evaluate its use in practice. Apparently, propagating changes between cloned variants may not be intended, and thus is no problem — while a platform requires developers to always investigate all relevant feature dependencies and adapt the bug fix accordingly (cf. Section 3.2.5). Nonetheless, the data from our systematic literature review (median: -60 %) and interview survey confirm that platform engineering can drastically decrease the costs for quality assurance (QA) 👍.

Coordinating development

CD. Coordinating is a core activity to ensure the success of a variant-rich system. Interestingly, we found contradicting insights considering that research usually argues that clone & own allows for independence, while a platform requires clearly defined roles and responsibilities (e.g., which developer owns a feature) 👉. In most cases, coordination is only mentioned as a problem in clone & own [Bauer and Vetrò, 2016; Faust and Verhoef, 2003] and platform engineering [Bauer and Vetrò, 2016; Dillon et al., 2014; Duc et al., 2014; van der Linden et al., 2007]. We found only one publication to support the argument that a platform facilitates coordination [Jepsen et al., 2007]. The data from our interview survey also indicates this ambiguity. We can see in Figure 3.4 that our interviewees consider coordination similarly costly for clone & own (5 %) and platform engineering (7 %).

Integrating variants

Integration. We elicited four cost estimations each for the integration and re-engineering of cloned variants into a platform, which align to the insights from our systematic literature review. Not surprisingly, it can become far more time consuming to re-integrate a long-living clone into a platform than a short-living one 👍. The costs heavily depend on the amount (i.e., delta) of co-evolution between variants and platform, which is also mentioned in other publications [BigLever Software, Inc., 2005; Dillon et al., 2014; Duc et al., 2014; Hetrick et al., 2006; Kapser and Godfrey, 2008; Kuiter et al., 2018b]. One of our interviewees summarized this situation, highlighting their preference for platform engineering:

> *"I think a lot of time is wasted on the long-living clones, because, if you wait one-and-a-half years until you merge, everything [has] changed, maybe. The new Linux kernel, a new version of something else, and then suddenly, your branch is just not working anymore. The longer you wait, the more pain it is. [...] It's always better to be up-to-date with master."*

This statement also indicates the causes for the higher costs, such as updating old features, understanding the co-evolution, or fixing outdated bugs and dependencies.

RO-E$_5$: Costs of Activities

Our results strengthen the evidence that successful software reuse heavily depends on a platform [Lucrédio et al., 2008; Rine and Sonnemann, 1998]. Furthermore, our systematic literature review indicates overall savings of around 52 % and that:

- *Setting up variant development is cheaper with platform engineering.*

- *Developing reusable features for platform engineering is more expensive (+20 %), but pays off with decreased variant-development costs (-67 %) — which is why a platform outperforms clone & own (-35 %).*
- *Platform engineering increases software quality, consequently reducing the costs for quality assurance (-60 %).*
- *Co-evolving variants (and platform) result in higher integration costs.*
- *Surprisingly, propagating bug fixes is more expensive for a platform.*
- *Coordinating development is similarly costly for both reuse strategies.*

3.2.5 RO-E$_6$: Cost Factors and Benefits

Cost factors

Lastly, we analyzed cost factors and benefits that relate to either reuse strategy. Again, we consider our data in combination and structure this section based on related insights.

Results

Results cost factors

We display the Likert-scale ratings for the cost factors we elicited during our interview survey in Figure 3.6. A negative rating (e.g., a variant requires a larger delta) indicates that our interviewees consider that factor to increase development costs. Contrary, a positive rating (e.g., the amount of reusable code for a variant) indicates that our interviewees consider that factor to reduce development costs. We separate the assessment for clone & own and platform engineering, and show average values in the middles to allow for easier comparisons.

Discussion

Code reuse

Reuse & Delta. First, we investigated the cost factors of reusable and newly required code (the delta). Not surprisingly, either strategy benefits from more code being reusable for a variant, while larger deltas cause additional costs 👍. Interestingly, reuse impacts both strategies similarly, but we can see one outlier for clone & own indicating a negative impact:

> *"Basically, for us it would be more of an effort to remove things, stuff we don't need compared to just having it there."*

While scalability and change propagation have been investigated for clone & own, this issue of removing unwanted features is less known. Also, it is interesting that our interviewees consider larger deltas to have a smaller impact on platform engineering, contradicting our finding that developing a new platform feature is more expensive (cf. Section 3.2.4) 👈.

Development staff

Developers & Staffing. During our study, we found the number of developers to be an important cost factor. We can see in Figure 3.5 that ten publications report that fewer staff (S) is required to develop a new variant based on a platform (median: -75 %) [Bass et al., 1999; Clements and Northrop, 2001; Clements et al., 2001; Cortiñas et al., 2017; Faust and Verhoef, 2003; Fogdal et al., 2016; Krueger et al., 2008; Li and Chang, 2009; Li and Weiss, 2011; Pohl et al., 2005]. This insight is in line with the established hypothesis that a (maintained) platform allows the same number of developers to develop more variants compared to clone & own 👍. Similar to Kolb et al. [2006], one of our interviewees stated:

> *"We had fewer products and fewer developers in the company, the platform was in a horrible state, so you [couldn't] really use it to release. [It] got more stable, but also the products that were using the platform increased exponentially. Instead of having 10 products on a lousy platform, you have a hundred products on a good platform."*

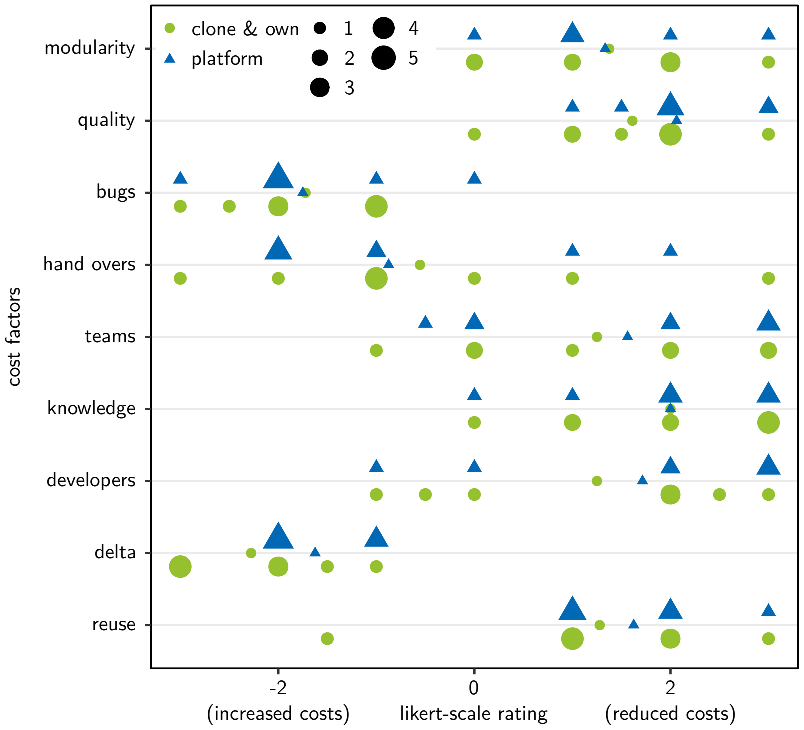


Figure 3.6: Likert-scale ratings for cost factors we elicited during our interview survey.

In both cases, the organization's growth was so large that it required even more staff to address new customer demands. For developing a single variant, our interviewees consider having more developers as beneficial—with the outliers representing cases in which the teams became too large and coordination was challenged.

Developers' knowledge

Knowledge. Developers require detailed knowledge to develop a variant-rich system, not only to comprehend source code, but also features, variability mechanisms (platform), and existing variants (clone & own). Particularly for clone & own, knowledge loss about variants is considered a major problem that can motivate the re-engineering towards a platform [Berger et al., 2020]. Two publications also raise the issue that missing knowledge is a major challenge for establishing a platform [Bauer and Vetrò, 2016; Slyngstad et al., 2006]. Our interviewees support these insights, indicating that knowledge is the cost factor with most impact on either reuse strategy. To tackle this problem, Axis has specific policies:

> *"[...] we try to have teams with experienced people together with new people."*

The results show that developers' knowledge is a primary cost factor in software reuse and for developing variant-rich systems. Interestingly, we do not know of a hypothesis or research that is concerned with knowledge in the context of re-engineering variant-rich systems ☜— motivating our own research in this direction (cf. Chapter 4).

Developers' collaboration

Teams & Hand Overs. In close relation to their knowledge, developers usually have to collaborate across different teams (e.g., variant development versus platform maintenance) to develop a variant-rich system. Consequently, our interviewees stated that clearly defined teams and the corresponding hand overs are important cost factors:

> *"I think that the more people you have, it becomes a lot of coordination, and also responsibilities [are] not as clear. If you are three people, it's hard to hide."*

We can see in Figure 3.6 that the number of teams with certain responsibilities is considered similarly positive for both reuse strategies. Not surprisingly, additional hand overs between the teams (e.g., from variant development to platform maintenance) are perceived slightly negative. Still, for collaboration overall, we found no major differences between clone & own and platform engineering, which aligns to our previous insights ☞.

Software quality

Bugs & Quality. Researchers and practitioners expect software reuse to improve the quality of the software and to reduce the number of bugs. We found three publication that support this hypothesis for clone & own [Bauer et al., 2014; Frakes and Succi, 2001; Walker and Cottrell, 2016], while three others state that quality is a problem [BigLever Software, Inc., 2005; Hetrick et al., 2006; Jansen et al., 2008] ☞. In contrast, numerous publications support this hypothesis for platform engineering [Clements and Northrop, 2001; Dillon et al., 2014; Ebert and Smouts, 2003; Ganz and Layes, 1998; Jepsen et al., 2007; Kolb et al., 2006; Li and Chang, 2009; Li and Weiss, 2011; Lim, 1994; Quilty and Cinnéide, 2011; Sharma et al., 2008; Slyngstad et al., 2006; Staples and Hill, 2004; van der Linden, 2013] 👍. Surprisingly, only one publication mentions quality as a costly challenge to ensure successful platform engineering [Kolb et al., 2006].

Platform quality

Our interviewees provided further supportive experiences in this regard. For example, one interviewee stated that Axis pushed strongly against clone & own to avoid quality and compatibility problems that could originate from long-living clones:

> *"I guess we tried to kill them off because it is a hassle to maintain. [...] If it's not tested every day, if it's not daily rebuilt and checked, [...] something is rotting in the code, it's not being compatible anymore with the platform."*

We found similar insights for **RO-E$_5$**, where quality assurance was considered less expensive for platform engineering. Despite the high quality of platform-based variants, our interviewees also stated that getting to this point was expensive. Initially, the platform had a low quality, and thus the developers did not trust it. So, it is not surprising that our interviewees considered the quality of the variant-rich system to be one of the most important cost factors, particularly for establishing platform engineering 👍.

Bugs

A major benefit assumed for platform engineering is that the improved quality leads to fewer bugs. We can confirm this hypothesis, seeing that several publications report a considerable decrease in bugs identified in a platform (median: -70 %) [Bass et al., 1999; Clements and Northrop, 2001; Clements et al., 2001, 2014; Fogdal et al., 2016; Lim, 1994; Otsuka et al., 2011; Quilty and Cinnéide, 2011] 👍. For clone & own, we find similar insights, with two studies indicating a reduction in the number of bugs of 35 % [Henry and Faller, 1995] to 66.7 % [Otsuka et al., 2011] 👍. Our interviewees expected that the number of bugs in a variant-rich system has a similar impact on the costs of developing a variant for either reuse strategy.

Independence of variants

Modularity & Dependencies. A benefit of clone & own is the independence of variants, allowing developers to freely implement features, test them, and reply faster to customer requests. Similar to other cases [Dillon et al., 2014; Dubinsky et al., 2013; Duc et al., 2014; Staples and Hill, 2004; Walker and Cottrell, 2016], Axis uses clone & own to innovate:

> *"If it's a new business, we don't want it [in the platform] because we don't want to maintain it."*

Moving faster to new markets is usually considered as the main reason to use clone & own instead of a platform 👍. Surprisingly, two publications state that a platform facilitated this innovation even more [Clements and Northrop, 2001]. Particularly, Kolb et al. [2006] state that only a platform enabled their organization to develop highly complex variants for new markets. One of our interviewees provided a similar insight:

> *"I would not be able to have such a complex product if I would not be able to reuse."*

This opposes established hypotheses 👎, and we need to better understand what enables an organization to move to new markets: independent variants or an established platform?

Dependencies

While the independence of clone & own is assumed to free developers of dependencies, we found contradicting insights in our systematic literature review 👉. For example, Bowen [1992] reports that a platform can actually resolve dependencies between clones, which several other publications mention as a problem [Bauer and Vetrò, 2016; Bauer et al., 2014; Jensen, 2009; Walker and Cottrell, 2016]. On the contrary, we found little evidence for the assumption that platforms cause dependency problems [Bauer and Vetrò, 2016; Kuiter et al., 2018b]. The core problem of dependencies in a platform may be best analyzed by Bogart et al. [2016], aligning to a statement of one of our interviewees:

> *"Since we're not part of the platform [...], they can sometimes break things they think [...] no one is using [...]. Then, we found out they broke something that we actually use."*

Together with policies and unintended side effects, we again identified missing knowledge as a problem that can easily lead to misbehaving or completely missing features that break some variants. However, this situation is not unique to platform engineering, but may occur in any variant-rich system in which changes are propagated. Our interviewees consider a modular structure of the variant-rich system to have a positive impact on resolving such dependency issues, and thus to reduce costs 👍.

Time-to-market

Time-to-market. Software reuse, and particularly platform engineering, is assumed to reduce the time-to-market for a new variant. We found four publications that confirm this hypothesis for clone & own [Bauer et al., 2014; Duc et al., 2014; Jansen et al., 2008], with Otsuka et al. [2011] reporting a reduction of around 30 % 👍. However, a platform (median: -63 %) can drastically outperform clone & own in this regard [Bass et al., 1999; Bergey et al., 2004; Clements and Northrop, 2001; Cohen et al., 2002; Ebert and Smouts, 2003; Ganz and Layes, 1998; Jensen, 2007, 2009; Kolb et al., 2006; Li and Chang, 2009; Li and Weiss, 2011; Lim, 1994; Otsuka et al., 2011; Sharma et al., 2008; Slyngstad et al., 2006; van der Linden, 2013; van der Linden et al., 2007], since a new variant can ideally be derived instantly — if all required features have already been implemented 👍.

RO-E$_6$: Cost Factors & Benefits

For the cost factors of software reuse, our data shows that:

- *More code reuse reduces and more new code increases costs, with platform engineering being impacted more positively than clone & own.*
- *The ideal number of developers who should develop a variant is challenging to assess for either reuse strategy, but a platform allows the same staff to develop more features and variants in the same time.*
- *Developers' knowledge about a variant-rich system is a core cost factor, independently of the reuse strategy employed.*
- *Coordinating development activities is challenging, and thus having teams with well-defined roles is beneficial for either reuse strategy. However, hand overs between these teams is associated with a slight negative impact on costs.*
- *Ensuring the quality of a platform is key for its success and leads to fewer bugs (-70 %). Still, clone & own benefits similarly from a higher system quality.*

- *Independently managed variants can be developed with clone & own and platform engineering. Surprisingly, both reuse strategies suffer from dependencies and rippling effects, which is why they can both benefit from modularity.*
- *Clone & own reduces the time-to-market (-30 %), but platform engineering can considerably outperform it (-63 %), since a wide range of features can be combined more easily.*

3.2.6 Threats to Validity

Threats to validity

In the following, we describe threats to the validity of our study and how we aimed to mitigate them. Note that most of these threats relate to correctly eliciting costs and assigning them to the right reuse strategy as well a activities. We employed several measures to ensure the credibility of the economical data we collected.

Construct Validity

Interview terminology

Most of our 28 interviewees have not been familiar with the research terminology on software reuse. We aimed to mitigate the threat of misunderstandings by investigating the terminology used at Axis during our exploratory interviews. Since our two contacts were familiar with both terminologies, they helped us clarify all research terms and adapt them to the ones used at Axis. Also, at least one author was present during each interview, allowing interviewees to clarify any construct they did not understand. Furthermore, each interview started with an introduction of the purpose and current scope of the study.

Terminology in publications

Similarly, the 58 publications we considered use different terminologies depending on the domain and authors. We carefully read each publication and used keywords to categorize them according to the two reuse strategies. To mitigate the threat that we may have falsely classified publications, and thus the corresponding data, we used an open-card-like sorting method to unify synonyms. In the end, we aligned the terminologies at Axis and in the publications to triangulate the data from both sources.

Internal Validity

Eliciting software costs

Precisely assigning costs to activities and assessing cost factors in software engineering is challenging to impossible. Also, copying or deriving an existing variant and delivering it directly to a customer has close to zero costs, while the costs of developing and propagating new features may be distributed among variants. Consequently, software-engineering economics are harder to quantify and assign compared to manufacturing, particularly because most organizations do not track their costs on this level of detail. We aimed to mitigate this threat by triangulating data from two sources that rely on different, reliable cost estimation methods: historical data and expert judgment [Boehm, 1984; Jørgensen, 2014].

Data verification

With our systematic literature review, we aimed to avoid threats that may be caused by conducting our interview survey only at a single organization. So, we tried to strengthen the internal validity of our study by relying on the experiences of skilled software engineers and adding data reported in research. We intended to improve the completeness of our systematic literature review by using five sources and verifying it against related work. Furthermore, we discussed regularly with our two contacts to make sense of our data and results. After the actual study, we also discussed our findings with three practitioners from another organization (i.e., Grundfos) that uses platform engineering to conduct a sanity check. Even though all these measures indicated that our findings are reasonable, we may still have collected wrong or misinterpreted data, which threatens our results.

External Validity

Generalizing the interview survey

Obviously, we can hardly generalize an interview survey at one organization directly for other organizations. However, many software platforms exhibit similar properties, even when comparing open-source and industrial ones [Hunsen et al., 2016]. In addition, most organizations employ similar practices for clone & own and platform engineering [Berger et al., 2020; Dubinsky et al., 2013]. Consequently, while we cannot overcome this threat, the results of our interview survey are still relevant for other organizations.

Data of other organizations

With our systematic literature review, we aimed to improve particularly the external validity of our study, adding data from over 100 different organizations. The systematic literature review comprises publications reporting on different domains, levels of maturity, programming languages, and countries. We argue that our systematic literature review is a suitable method to mitigate threats to the external validity of our study, especially since its results are similar to our interview survey.

Conclusion Validity

Data availability

While we cannot fully overcome the aforementioned threats, we argue that we employed appropriate mitigation strategies to obtain reliable results. Our findings represent an extensive body of knowledge on the economics of software reuse that can guide organizations and help researchers to scope their work. For confidentially reasons, we cannot release the interview recordings or transcripts. Still, we reported our methodology in detail and make all relevant artifacts publicly available to allow others to replicate and verify our study.

3.3 Feature-Oriented Re-Engineering

Re-engineering economics

Our previous findings provide an understanding of the economics of (re-)engineering variant-rich systems and empirical data on the costs associated with developing variants based on either reuse strategy. Considering that our findings indicate that an organization should strive to adopt platform engineering, we now investigate the economics and challenges of re-engineering a platform in more detail. For this purpose, we synthesize the results of multiple case studies in which we re-engineered real-world variant-rich systems into platforms [Åkesson et al., 2019; Debbiche et al., 2019; Krüger and Berger, 2020a; Krüger et al., 2017a, 2018d; Kuiter et al., 2018b].

Section contributions

More precisely, we synthesize our cases to address the following sub-objectives of **RO-E**:

RO-E$_7$ *Identify and analyze the activities of re-engineering variant-rich systems.*

While re-engineering has been extensively studied in the past, the activities performed are reported inconsistently and on coarse levels [Assunção et al., 2017]. To tackle this problem, we synthesize the activities we executed during each of our cases and discuss their purpose. Similar to **RO-E$_4$**, this overview helps to better understand how such a re-engineering can be executed.

RO-E$_8$ *Assess the costs of the identified activities and compare between the cases.*

For three of our cases, we involved economic assessments based on experiences, and for two of these by documenting efforts. We employed different strategies, particularly concerning the variability mechanism, and thus traceability, employed. Our results help to better understand the economics of re-engineering a platform and the impact of deciding for a specific architecture.

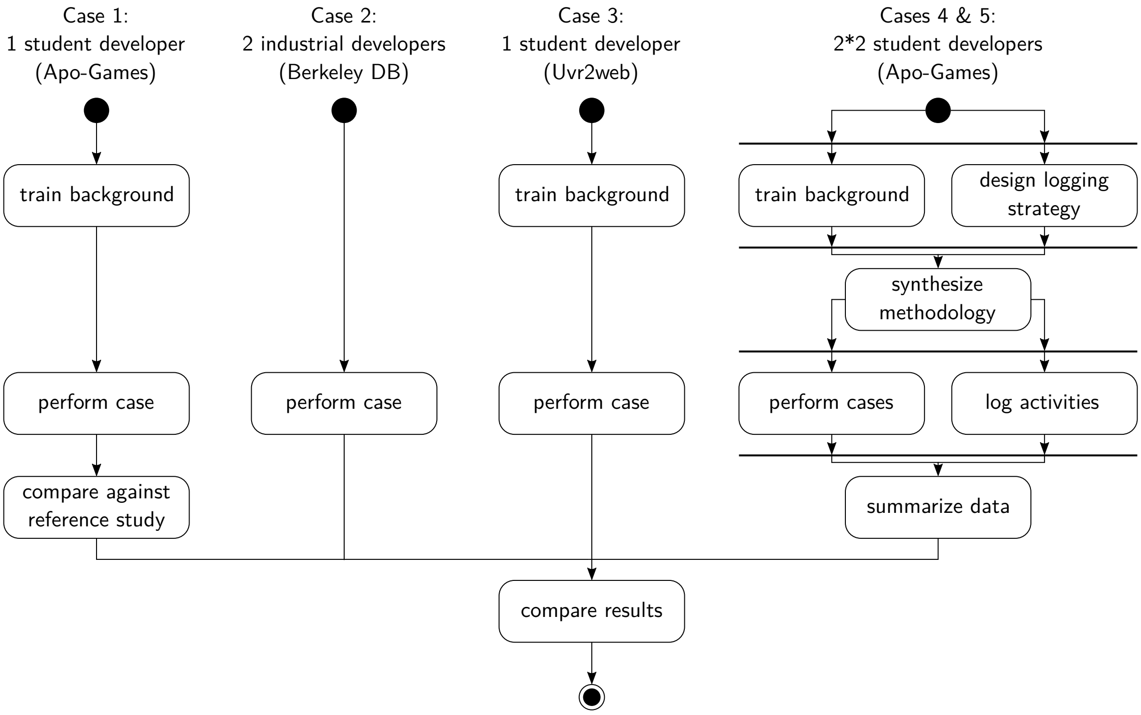


Figure 3.7: Overview of the methodologies for our five re-engineering cases.

RO-E$_9$ *Discuss lessons learned by synthesizing the experiences of all cases.*

Finally, we synthesize the experiences we obtained from each case, shedding light into potential pitfalls and challenges an organization may face. These insights help to avoid costly rework and can raise the awareness regarding critical decisions. We hope that these experiences guide organizations during their decision making and researchers in tackling important problems.

In the following, we describe our methodology in Section 3.3.1 and threats to validity in Section 3.3.5. Afterwards, we present and discuss the results for each of our sub-objectives in Section 3.3.2, Section 3.3.3, and Section 3.3.4, respectively.

3.3.1 Eliciting Data with a Multi-Case Study

Multi-case study design

Our methodology is based on a multi-case study design [Bass et al., 2018; Leonard-Barton, 1990; Runeson et al., 2012] comprising five cases that we compare against each other. Each of the cases was performed by different developers on varying systems, helping us to obtain transferable findings and verify those obtained in different cases. Except for Case 3, we advised the developer teams in regular discussions in order to react to new problems and experiences, which means that we followed an action-research-like methodology [Davison et al., 2004; Easterbrook et al., 2008; Staron, 2020]. In Figure 3.7, we display an overview of the concrete methodology we employed in each case. We adapted the methodologies based on the scenario we focused on in each case, and the experiences we gained in previous cases. In Table 3.6, we summarize the system properties and scenarios.

Subject Systems

Subject systems

We [Strüber et al., 2019] found no existing dataset on the evolution of variant-rich systems that provides a reliable ground truth, supports the scenarios we are concerned with, or involves real-world cloned variants. This has been a challenging problem for research, since variants derived from a platform are not ideal to simulate clone & own, for instance, because they

Table 3.6: Overview of our five re-engineering cases, the systems used, and scenario employed. Case 2 comprises several personal variants developed for individuals. In Case 5, the developers did analyze all, but re-engineered only the asterisked, variants.

	system(s)	year	loc	scenario		
Case 1	ApoClock	2012	3,615	feature location for clone & own Android Apo-Games	[Krüger et al., 2017a]	
	ApoDice	2012	2,523			
	ApoSnake	2012	2,965			
	ApoMono	2013	6,487			
	MyTreasure	2013	5,360			
Case 2	Berkeley DB	2014	229,419	re-engineering an annotation-based C database into a partly composition-based platform	[Krüger et al., 2018d]	
Case 3	Personal	—	—	re-engineering web-based clone & own heat control software into a platform	[Kuiter et al., 2018b]	
	TempLog	2013	6,837			
	Uvr2web	2013	5,148			
Case 4	ApoClock	2012	3,615	re-engineering clone & own Android Apo-Games into an annotation-based platform	[Åkesson et al., 2019]	[Krüger and Berger, 2020b]
	ApoDice	2012	2,523			
	ApoSnake	2012	2,965			
	ApoMono	2013	6,487			
	MyTreasure	2013	5,360			
Case 5	ApoCheating	2006	3,960	re-engineering clone & own Java Apo-Games into a composition-based platform	[Debbiche et al., 2019]	
	ApoStarz	2008	6,454			
	ApoIcarus*	2011	5,851			
	ApoNotSoSimple*	2011	7,558			
	ApoSnake*	2012	6,557			

do not involve dead code that may occur in actually cloned variants [Debbiche et al., 2019; Schultheiß et al., 2020]. To tackle such problems, we focused on using real-world systems that allowed us to still conduct manual analyses. In the end, we relied on two sets of cloned variants (i.e., Apo-Games, heat control software) that we re-engineered towards platforms. Moreover, we used a platform-like variant-rich system (i.e., Berkeley DB) to investigate the impact of different variability mechanisms during the re-engineering towards a full-fledged platform.

Apo-Games

To address the problem that real-world cloned variants are not publicly available, we [Krüger et al., 2018a] contributed the Apo-Games—which are 89 quite successful games in total (i.e., used for programming competitions, between 100 and 50,000 downloads for the Android games). The Apo-Games dataset[7] includes 20 Java and five Android games that have been developed by a single developer using clone & own. All games range from 1,659 to 19,558 lines of code and have been developed between 2006 and 2013. Due to the lack of other datasets, it is not surprising that the Apo-Games have already been used extensively in research, for instance, to compare the effectiveness of analysis techniques on simulated and real-world cloned variants [Schultheiß et al., 2020], to propose refactorings for re-engineering

[7] https://bitbucket.org/Jacob_Krueger/apogamessrc

variants into a platform [Fenske et al., 2017a], and to design recovery techniques for platform architectures [Lima et al., 2018] or domain knowledge [Reinhartz-Berger and Abbas, 2020].

Berkeley DB For one of our cases, we relied on the industrial Berkeley DB,[8] an embedded database management system developed by Oracle. We relied on the C version that uses preprocessor annotations to control features. While this system does not represent cloned variants, we focused on understanding the pros and cons of using annotations or composition in the corresponding case. In that context, the results are comparable to those we obtained for re-engineering cloned variants, and they motivate particularly our traceability objective (**RO-T**). Similar to the Apo-Games, Berkeley DB has been used extensively in research [Apel et al., 2013b; Kästner et al., 2007; Liebig et al., 2010; Rosenmüller et al., 2009; Tešanović et al., 2004], since it represents a relevant case for practice.

Uvr2web In one case, we re-engineered cloned variants of a web-based heating control software (for the UVR1611 Data Logger Pro) that have been developed by a student for two different organizations and multiple private users, with each variant including individual features. The cloned variants comprise various technologies (e.g., Arduino, MySQL, Apache server, PHP, Java, C++, C#) depending on the customized features they include. Uvr2web is successfully running in practice, with more than 20 private users contacting the student developer regarding the publicly available software. Moreover, the variants of one organization have been downloaded over 300 times with 30 customers using the online database. While we make most of the artifacts for Uvr2web publicly available,[9] we cannot share all details and artifacts, since some are critical for the organizations, and thus are confidential.

Cases

Case 1 In Case 1 [Krüger et al., 2017a], we have been concerned with locating features in a set of cloned variants. The case itself was performed by one student developer who obtained the necessary background during a series of lectures on platform engineering. As we show in Table 3.6, we picked the five Android Apo-Games for this case, mainly because we had a reference study on refactoring automatically located features (using code-clone detection [Bellon et al., 2007; Roy et al., 2009]) that used the same variants [Fenske et al., 2017a]. The student developer also used a code-clone detection tool (i.e., Clone Detective, which is part of the ConQAT framework [Deissenboeck et al., 2008; Juergens et al., 2009]) to identify feature candidates, but inspected and extended those candidates manually (cf. Section 3.1.3). Initially, we located features and constructed a feature model for a single variant. We incrementally extended our analysis by mapping the results to the remaining variants. During the case, we documented our experiences, particularly regarding the efforts, as well as data on the features to compare our results to the reference study.

Case 2 In Case 2 [Krüger et al., 2018d], we re-engineered Berkeley DB from purely annotative variability towards composition, establishing platform engineering based on automated tooling and feature modeling (using FeatureIDE [Meinicke et al., 2017]) in the process. The re-engineering itself was performed by two experienced industrial developers who had the necessary background knowledge on platform engineering and Berkeley DB. As starting point, we relied on previous work of other researchers that performed similar re-engineerings [Benduhn et al., 2016; Kästner et al., 2007]. During the case, we documented our activities and processes in order to understand how features can be re-engineered. Moreover, we documented the efforts in terms of which activities we had to repeat and the problems we faced.

[8] https://www.oracle.com/database/technologies/related/berkeleydb.html
[9] https://github.com/ekuiter/uvr2web
https://github.com/ekuiter/uvr2web-spl/tree/master/spl/artifacts

Case 3

In Case 3 [Kuiter et al., 2018b], we analyzed the re-engineering of the Uvr2web variants that was performed by the original (student) developer. The developer implemented the variants for personal use at first, but later for two organizations. During a series of lectures on platform engineering, they decided that re-engineering the variants into a platform would be beneficial to fulfill new feature requests and solve existing problems, for instance, regarding bug fixing. Since the available tooling for platform engineering was not designed for integrating various technologies, the developer first implemented new tooling to model features and derive variants from the platform, using multiple variability mechanisms, such as the C preprocessor, a build system, and plug-ins. Together with the developer, we reiterated through the re-engineering process and version history to elicit the activities performed and experiences.

Cases 4 & 5

From the start, we designed Cases 4 [Åkesson et al., 2019] and 5 [Debbiche et al., 2019] to be comparable [Krüger and Berger, 2020a], utilizing the experiences we obtained from the previous cases to focus particularly on understanding the economics of re-engineering cloned variants into a platform. To this end, we worked with two teams, each with two student developers, who collaborated on the design of the methodology, but performed their cases individually. They first conducted separate literature surveys to obtain the background knowledge they required to perform their cases and derive a logging framework to document their activities and efforts. Then, we synthesized the proposed methodologies and frameworks based on our previous experiences and insights regarding cost models (cf. Section 3.1.1) to define what to document, aiming to obtain comparable data. Besides documenting in the version-control systems based on commits, we derived the logging template we display in Table 3.7, comprising three sections:

Information describes an activity's meta-data, namely its type, a short name, an identifier, the original variant the activity was employed on (if applicable), the start date, the end date, as well as a short description in natural language.

Data references the commits relating to an activity and provides applicable metrics, namely the amount of person-hours (ph) spent as well as the lines of code and files changed.

Artifacts contains information regarding the artifacts relating to an activity, namely the input, output, and tools used (e.g., source code, feature model, diff tool).

Activity Description documents summaries regarding three further properties of an activity: its complexity, importance, and dependencies to other activities.

Using the methodology and logging framework we established, both teams performed their case and logged their activities. As subject systems, each team used five variants of the Apo-Games with similar sizes. Each team worked independently, with one re-engineering the Android Apo-Games into an annotation-based platform using Antenna,[10] while the other re-engineered Java variants towards composition using feature-oriented programming [Prehofer, 1997] and the composer FeatureHouse [Apel et al., 2009, 2013b]. The teams could use any tool they wanted, with FeatureIDE being the primary tool. During the whole case study, we had weekly meetings to discuss the progress and challenges, allowing us to react and adapt the methodology. After both teams finished their case, they analyzed their results individually without considering the other team's data. Only after this analysis, we synthesized the findings of both teams, focusing on the economics of re-engineering they documented. All of our (partly evaluated) data and artifacts are available in open-access repositories.[11]

[10] http://antenna.sourceforge.net
[11] https://bitbucket.org/easelab/aporeengineering

Table 3.7: Concrete instance of our logging template for re-engineering Cases 4 and 5.

information	
activity type:	preparatory analysis
activity:	removing unused code
activity id:	A10
variant ids:	V2, V3, V4, V5
start date:	2019-03-11
end date:	2019-03-12
description:	Identifying code that is not used in the variants. We removed unused code to facilitate analyzing the variants.
data	
total hours:	12
commits:	7 35351f7035e22907d30828cd82a475d6fd012d75 1397a2c35632c474e361da003d7c8027f3d659e7 ...
loc added:	0
loc removed:	11,670
loc modified:	0
files added:	0
files removed:	78
files modified:	133
artifacts	
input:	source code
output:	refactored source code
tools:	Eclipse, UCDetector, IntelliJ
activity description	
complexity:	The activity is of relatively low complexity, thanks to the available tools.
importance:	This activity is very important because failure to detect unused code means the developer will spend time transforming source code that is never used.
dependencies:	n/a

3.3.2 RO-E$_7$: Re-Engineering Activities

Re-engineering activities

In this section, we report which activities the developers performed in each case and derive activity types to provide a uniform classification. We show a summary of all activities performed and their mapping to activity types as well as our cases in Table 3.8.

Results

Activity types

We elicited a diverse set of activities from the developers of each case, with various levels of granularity and varying terminologies. For instance, some of the developers simply referred to domain engineering, while others more specifically referred to feature or variability modeling. To allow us to compare between all five cases, we derived nine common activity types (ATs) by abstracting the activities and building on previous research:

AT$_1$ **Platform engineering training** summarizes all activities related to getting familiar with the methods and tools (potentially extending them with own implementations) of platform engineering.

AT$_2$ **Domain analysis** summarizes all activities related to understanding the domain of the subject systems, for instance, by running and playing the Apo-Games.

AT$_3$ **Preparatory analysis** summarizes all activities related to improving the quality of a legacy system (e.g., removing unused code) or obtaining data (e.g., identifying code clones) to support the actual re-engineering.

Table 3.8: Mapping of the activities performed in each case.

activity	activity types	case				
		1	2	3	4	5
research on platform engineering	AT_1	●	○	●	●	●
research/extend tools	AT_1	●	●	●	●	●
run legacy system	AT_2; AT_4	●	●	●	●	●
translate comments to English	AT_3	○	○	○	○	●
pairwise compare variants	AT_3; AT_4; AT_6	●	○	○	●	●
remove unused code	AT_3	○	○	○	○	●
reverse engineer class diagrams	AT_2; AT_5	○	○	○	●	●
review source code	AT_2; AT_4; AT_6	●	●	●	●	●
create feature model	AT_7	●	●	●	●	●
re-engineer source code to features	AT_8; AT_9	○	●	●	●	●
test re-engineered platform	AT_9	○	●	●	●	●

AT_4 **Feature identification** summarizes all activities related to identifying which features exist within a system.

AT_5 **Architecture identification** summarizes all activities related to understanding the architecture of a legacy system, and defining a new one for the platform.

AT_6 **Feature location** summarizes all activities related to locating the source code implementing an identified feature (at class, method, statement, or sub-statement level).

AT_7 **Feature modeling** summarizes all activities related to modeling the commonalities and variability of the platform (i.e., mandatory and optional features).

AT_8 **Transformation** summarizes all activities related to the actual transformation of the source code and implementation of the platform.

AT_9 **Quality assurance** summarizes all activities related to validating the re-engineered platform (e.g., deriving variants) and typical quality assurance (e.g., unit testing).

Using these activity types, we can classify the activities the developers performed.

Intertwined activities

Of all activities we elicited from our cases, we identified 11 that we could clearly distinguish. However, we can see in Table 3.8 that these activities are heavily intertwined, with several activities mapping to multiple activity types and vice versa. For example, familiarizing with the subject systems by running them (e.g., playing the Apo-Games, testing a Berkeley DB benchmark) contributes to domain analysis (AT_2) and feature identification (AT_4). We remark that the extent to which an activity was performed varied between cases. For instance, in Cases 4 and 5, the developers used a pairwise comparison of variants only as a preparatory analysis to facilitate their later activities (AT_3). However, in Case 1, the developer used the pairwise comparison explicitly to identify actual features (AT_4) and incrementally refine as well as map feature locations to other variants (AT_6).

Activities

We can briefly summarize each elicited activity as follows:

Research on platform engineering: During this activity, the (student) developers familiarized with the concepts of platform engineering (e.g., through lectures, literature surveys), ensuring that they had the necessary knowledge to preform their case.

Research/extend tools: For this activity, the developers had to identify (e.g., FeatureIDE), extend (e.g., FeatureC [Krüger et al., 2018d]), or even implement (e.g., for Case 3) tools that they wanted or needed to use.

Run legacy system: In this activity, the developers executed their subject systems with the goal of understanding their behavior and identifying potential features — representing a top-down analysis [Xue, 2011; Xue et al., 2012].

Translate comments to English: Since the Apo-Games comprise German comments, one team of developers decided to translate these to English, aiming to gain a better understanding based on the original developer's documentation.

Pairwise compare variants: This activity involves diffing the source code of cloned variants and mapping located features to other variants based on code-clone detection.

Remove unused code: One developer team found that their set of the Apo-Games comprised a high ratio of common code, but also that a lot of this code was not actually used, which is why they removed it before the actual transformation.

Reverse engineer class diagrams: During this activity, the developers automatically extracted class diagrams of the variants to understand their architecture and guide the design of the platform.

Review source code: In this activity, the developers analyzed the source code of their subject systems to identify features, locate the corresponding source code, and document the results to enable the actual transformation — representing a bottom-up analysis [Xue, 2011; Xue et al., 2012].

Create feature model: To document the features identified and specify their dependencies in the resulting platform, all developers constructed feature models.

Re-engineer source code to features: This activity refers to the actual transformation of the legacy source code towards the platform, which we found to differ significantly for our cases, due to the variability mechanisms we considered.

Test re-engineered platform In all of our cases in which we re-engineered a platform, this activity was heavily intertwined with the actual re-engineering, since most developers immediately tested the platform after integrating a feature.

For most of these activities, we experienced pitfalls that we discuss in Section 3.3.4.

Discussion

Elicited activities and their types

Even though we considered different subject systems, tools, programming languages, and variability mechanisms, all developers performed similar activities. This indicates that our results represent useful abstractions of more fine-grained activities and can be used to guide the re-engineering of systems into a platform. For instance, one developer team pointed out that a more detailed analysis of the activities employed (e.g., individual refactorings) could improve our understanding of that activity's properties and economics. Still, particularly the developers in Cases 4 and 5 highlighted that our abstraction into activity types was necessary to compare their findings. For this reason, we argue that we define a comprehensible set of activities and activity types to support developers in understanding how to re-engineer a platform.

Process implications

Most research on platform (re-)engineering relies on waterfall-like process models established decades ago [Krüger et al., 2020d]. For instance, they often strictly separate domain and application engineering [Apel et al., 2013a; Clements and Northrop, 2001; Pohl et al., 2005] or detection, analysis, and transformation phase [Assunção and Vergilio, 2014; Assunção et al., 2017]. In our cases, we elicited similar activities, but experienced that our developers constantly switched between these, for instance, because they identified new features (AT_4) or located new code belonging to a feature (AT_6) while reviewing the source code during their domain analysis (AT_2). As a consequence, an iterative (re-)engineering process with constant updates seems more reasonable in practice.

RO-E_7: Re-Engineering Activities

Regarding the activities the developers performed in our cases, we found that:

- *We could define abstract activity types to represent and classify more fine-grained re-engineering activities.*
- *The different activity types are intertwined during the actual re-engineering.*
- *Established, waterfall-like process models seem unreasonable for practice.*

3.3.3 RO-E_8: Economics of Re-Engineering

Re-engineering efforts

Next, we investigate the efforts the developers spent and recorded during Cases 4 and 5, individually and through a cross-case analysis. We summarize these efforts in terms of person-hours and as ratio in Table 3.9. Since it is challenging to precisely track these efforts, we checked the estimates against the corresponding version-control histories. We integrated the efforts for transforming (AT_8) and quality assuring (AT_9) for these cases, since both teams continuously added new features to their platforms and tested them in parallel. As a result, it is not possible to clearly separate these efforts (e.g., in Case 5 the team specified only transformation efforts and stated that this included 50 % quality assurance).

Results

Efforts Case 4

We can see in Table 3.9 that the developer team of Case 4 recorded a total of 496 ph. They performed extensive platform engineering training (18.15 %) as well as domain analysis (16.53 %). A main cost factor impacting the domain analysis was the need to understand Android. For this reason, the existing tools for platform engineering were insufficient (similar to Case 3), and the team had to invest substantial effort into integrating different tools (e.g., FeatureIDE and IntelliJ IDEA). Nonetheless, the team invested most efforts into the actual transformation and quality assurance (48.39 %).

Efforts Case 5

For Case 5, we can see that the team spent more than half of its total of 371.5 ph on transforming and quality assuring the re-engineered platform (55.72 %). Considering the remaining activities, most effort went into preparatory analyses (e.g., removing unused code, translating comments to German) and feature location (aligning to our analysis in Section 3.1.3). All other activities required considerably less effort.

Cross-case comparison

Comparing the efforts of both cases, we can see that the developer team in Case 4 required almost 125 ph more than the team in Case 5. Particularly, they invested more time at the beginning of their case, namely during the platform-engineering training and domain analysis. As mentioned, the particular reason for this was the need to integrate different tools to be able to re-engineer an Android-based platform. However, the team in Case 5 did not fully track their platform engineering training, since they already started a literature survey before the study design.

Transformation efforts

Interestingly, the efforts for transforming and quality assuring is similar in both cases (207 ph to 240 ph or 55.72 % to 48.39 %, respectively), even though both teams employed different variability mechanisms for their re-engineered platform (i.e., annotations versus composition). Unfortunately, the team in Case 5 could transform only three of the intended five variants (asterisked in Table 3.6), due to time constraints. The main problem the team experienced was the complexity of using a composition-based variability mechanism.

Table 3.9: Efforts documented for Cases 4 and 5 in terms of person-hours (ph) spent.

id	**activity type**	**Case 4**		**Case 5**	
		ph	**%**	**ph**	**%**
AT_1	platform engineering training	90.00	18.15	16.00	4.31
AT_2	domain analysis	82.00	16.53	18.00	4.85
AT_3	preparatory analysis	40.00	8.06	49.25	13.26
AT_4	feature identification	22.00	4.44	22.25	5.99
AT_5	architecture identification	5.00	1.00	2.00	0.54
AT_6	feature location	7.00	1.41	50.00	13.46
AT_7	feature modeling	10.00	2.00	7.00	1.88
AT_8	transformation	180.00	36.29	103.50	27.86
AT_9	quality assurance	60.00	12.10	103.50	27.86
total		496.00		371.50	

Compared to annotations, composable modules require larger refactorings, merges, and adaptations of the source code to enable a configurable platform. A re-appearing challenge for the team was to localize bugs after integrating a new feature, which became more complex since previously connected code was now separated. In addition, the team highlighted that composition-based variability mechanisms are less established in practice, resulting in problems to find practical resources and guides for such a re-engineering project.

Feature location efforts

In addition to the mentioned platform engineering training and domain analysis, feature location is among the activities with larger differences. While the team in Case 5 spent 50 ph (13.46 %) on this activity, the team in Case 4 spent only 7 ph (1.41 %). We thought this may be the result of different analysis strategies, but during our discussions it became clear that the issue is again connected to the variability mechanisms used. For feature-oriented programming (Case 5), the team first required to fully identify and locate all features of the cloned variants to be able to transform them into meaningful modules. In contrast, the preprocessor (Case 4) allowed the other team to add variability ad hoc by step-wise extending feature locations with additional annotations whenever necessary. This insight is interesting, since it indicates that the efforts for feature location depend on the variability mechanism—even though some of this effort may be hidden within the actual transformation.

Other activity efforts

For all remaining activities, we can see that the person-hours spent and their ratios are comparable between both cases. This is not surprising, since most of these activities are rather independent of project specifics, and thus similar for both cases. For instance, both teams relied on similar strategies to identify features (AT_4) and architectures (AT_5). So, we can summarize that the main differences between the two cases are caused by preparatory analysis (AT_3) and feature location (AT_6). These activities also contribute to most efforts after the actual transformation and quality assurance.

Discussion

Unexpected cost factors

In all of our cases (not only Cases 4 and 5), the developers experienced that certain project properties can cause unexpectedly high efforts. Concretely, in Case 4, the missing tool support for Android-based platforms drastically increased the costs. In Case 5, the developers relied heavily on code comments, and experienced that removing unused code facilitated other activities. For both cases, the developers experienced that missing knowledge about platform engineering and their subject systems challenged their re-engineering projects. These findings align to our other three cases (e.g., in Cases 2 and 3 new or extended tools were required), and highlight that various project properties can have unexpected impact on the costs of re-engineering. This challenges the research community to better understand these cost factors and improve tool support to facilitate important activities.

Cost factor tools

As a more concrete example, consider the activities for which the developers could rely on established tools: architecture identification (AT_5) and feature modeling (AT_7). In Cases 4 and 5, both teams used plug-ins of the Eclipse framework to automatically reverse engineer class diagrams, reducing the required effort considerably. For feature modeling, they could rely on FeatureIDE's comprehensive feature-model editor, and the actual modeling was straightforward if the necessary information was collected. Unfortunately, existing tools poorly support most other activities, leading to considerably increased efforts. A particularly expensive activity was the manual feature location that required extensive knowledge and could only be supported through pairwise comparison (AT_3) using code-clone detection tools.

Variability mechanisms

While the efforts in Cases 4 and 5 are comparable, our experiences support the argument that re-engineering a platform using an annotation-based variability mechanism is more suitable in practice. Comparing the experiences of both teams, the ability to add annotations ad hoc seems to be a major benefit, meaning also that the developers required less knowledge before the transformation. In more detail, feature-oriented programming is complex, developers need to learn it, and it seems too expensive for transforming small, scattered features, as in the Apo-Games. A preprocessor is a simpler variability mechanism, and its annotations do not require the developers to refactor small, scattered features into modules. However, the developer team in Case 4 also highlighted a common problem with annotation-based variability: the code becomes less readable and poorly structured, suggesting that introducing composition to some extent would be helpful to improve the quality of the platform. So, while it seems more reasonable to initiate a re-engineering project using an annotation-based variability mechanism, the developers should also consider to decompose features of the platform — if their granularity allows for this.

RO-E$_8$: Re-Engineering Economics

For the economics of re-engineering cloned variants into a platform, our data shows:

- *Various project properties (e.g., missing tool support) heavily impacted the costs.*
- *Developers may (or even cannot) be aware of all of those properties (e.g., reliability of comments, code quality, available tools, knowledge) and their trade offs.*
- *While some automation (e.g., for architecture identification) worked well, better tools for other activities are still needed.*
- *Re-engineering cloned variants into a composition-based platform was more challenging and costly than the re-engineering into an annotation-based one.*
- *Using annotations required less effort during feature location.*
- *Among all other activities, preparations (e.g., platform engineering training, domain analysis, preparatory analysis) and the transformation (including quality assurance) were the most costly ones.*
- *The costs for most other activities were similar for both cases.*

3.3.4 RO-E$_9$: Lessons Learned

Lessons learned

During our five cases, we collected various related lessons learned that provide additional evidence for our previous findings and motivate our other research objectives. In the following, we briefly discuss these lessons (first benefits, then challenges), reporting practical insights

Table 3.10: Mapping of the lessons we learned through each case.

type	**lesson learned**	**case**				
		1	**2**	**3**	**4**	**5**
benefits	delivering novel variants	○	○	●	○	●
	improving code quality	○	●	○	○	●
	managing variability	○	●	●	●	●
	reduced codebase	○	○	●	●	●
	removing unused code	○	○	○	○	●
challenges	available tools	●	●	●	●	●
	deciding about features	●	○	●	●	●
	intertwined activities	○	●	●	●	●
	unexpected efforts	○	○	○	●	●
	variability mechanisms	○	●	○	●	●

on pitfalls and opportunities of re-engineering variant-rich systems that an organization should take into account to analyze the corresponding economics. For researchers, these lessons define open research opportunities. In Table 3.10, we display a mapping of each case to the lessons we learned through it. We remark that we focused on eliciting challenges, particularly since benefits could mainly be identified through the use of a platform in practice — which we could not evaluate or simulate except for Case 3.

Benefits

Delivering novel variants

In two of our cases, we experienced that introducing variability by re-engineering features into a platform immediately allowed us to configure new, reasonable variants. For Case 3, this allowed the developer to deliver new variants to customers without reworking another cloned variant. In Case 5, we could configure 56 games, instead of only the three the developers re-engineered fully into the platform, significantly increasing the variant portfolio. We remark that we successfully configured, derived, and executed all 56 games, but not all of these configurations were fully functional (e.g., due to features not interacting correctly with each other). Still, we could configure and run the previously cloned variants from the platform.

Improving code quality

Re-engineering a variant-rich system is also an opportunity to improve the quality of assets (e.g., source code, documentation) — which we previously found to be a major success and cost factor for platform engineering. For instance, we introduced feature modules into Berkeley DB, refactored complex code structures (e.g., undisciplined annotations [Fenske et al., 2020; Liebig et al., 2011; Medeiros et al., 2015; Schulze et al., 2013]), and refined the existing feature model we built upon. However, improving assets also leads to additional costs. So, an organization may use the opportunity to improve the quality of its variant-rich system, but has to analyze the trade offs in terms of investments and benefits.

Managing variability

For all cases in which we re-engineered a variant-rich system, we experienced that platform engineering improved our ability to understand and manage the existing variability. As a consequence, it did become easier to update, add, and fix features, especially since we did not have to implement each feature multiple times for individual variants. Moreover, a feature model provides a better starting point to identify which features are relevant for a new customer. While re-engineering, we were also able to reduce the variability by removing unnecessary features, thus decreasing the complexity of the platform.

Reduced codebase

A major benefit we experienced while re-engineering cloned variants is the smaller codebase we had to maintain afterwards — a direct consequence of most other benefits (e.g., reducing

Table 3.11: Statistics on the cloned variants (legacy loc) and re-engineered platforms.

case	legacy loc	platform		
		implemented features	loc	loc ratio
3	11,985	52	10,584	88.31 %
4	20,950	42	10,278	49.06 %
5	19,966	23	13,932	69.78 %

variability, removing unused code). We display statistics on the number of features we implemented and the reduction of the codebase (loc ratio) for our three corresponding re-engineering cases in Table 3.11. As we can see, each re-engineered platform was smaller than the legacy systems, which is not surprising since we merged redundant code. These size ratios of the resulting platforms are reasonable: We achieved the smallest reduction (Case 3: 11.69 %) for the web-based clones that use various technologies, and thus could not be fully merged. For the other two cases, annotations achieved a far higher reduction (Case 4: 50.94 %), since these allowed for more merges with few annotations to add variability. In contrast, feature-oriented programming (Case 5: 30.22 %) required more glue code to handle feature interactions and fine-grained features, as well as for enabling composition (i.e., defining refinement methods) — partly equalizing for removing unused code.

Removing unused code

We already described the activity of removing unused code. In Case 5, the developers performed an automatic dead-code analysis using the Eclipse plug-in UCDetector.[12] This alone reduced the codebase of the cloned Apo-Games by almost 40 % (11,670 out of 30,380 LOC). We found that the dead code seemed to represent a reoccurring theme we also identified before: developers cloning a variant without removing unused features, for instance, those representing enemy entities. Again, an organization may use the re-engineering of a variant-rich system to address such unused code that may otherwise cause unwanted feature interactions, requires maintenance, and may confuse developers.

Challenges

Available tools

We already highlighted the challenges of finding and using tools that support platform (re-)engineering through all activities and technologies. Even though numerous tools for platform engineering exist [Horcas et al., 2019] few are actually available or in a practically usable state. As a consequence, we relied mainly on the open-source tool FeatureIDE as a general development environment for platform engineering, which is similar to the industrial tools pure::variants [Beuche, 2012] and Gears [Krueger, 2007]. However, none of the available and functional tools provided sufficient support for the re-engineering process. Moreover, existing tools are often limited to certain variability mechanisms or programming languages, and complex to extend. This challenged several of our cases in which we had to combine different or even implement our own tools, for instance, when we required support for annotation-based and composition-based variability in parallel (Case 2), multi-technology projects (Case 3), or Android (Case 4).

Deciding about features

For all four cases in which we analyzed or re-engineered cloned variants, we found it challenging to decide which features exist, what code belongs to them, and whether they should be part of the platform. For instance, for the Apo-Games (Case 4), the same features vary heavily between the individual variants, since these implement completely different types of games and concepts. This diversity made it hard to re-engineer reasonable features, but this arguably depends on the variants' similarity and developers' knowledge. Similarly,

[12] https://marketplace.eclipse.org/content/unnecessary-code-detector

we experienced that top-down and bottom-up analyses are complementary, and should be combined to decide about features. Interestingly, in neither case did tool support (e.g., code-clone detectors) help particularly well, they usually only provided initial seeds. In Case 1, we focused on this feature identification and location problem in particular: For the reference study [Fenske et al., 2017a], features were simply refactored based on code clones. Not surprising, considering the concept-assignment problem [Biggerstaff et al., 1993], our manual analysis revealed several differences that are otherwise masked by the code clones, such as a higher number of common features, a better separation of feature code, and more meaningful features in general. Identifying and locating features are well known problems for platform engineering, and our experiences suggest that the involved developers have to agree on a specific notion of features [Berger et al., 2015; Classen et al., 2008], use manual analysis, and build upon their knowledge of the variants. For Case 3, we could rely on the knowledge of the original developer, which did not solve, but considerably facilitated the problem of deciding about features.

Intertwined activities

During our re-engineering cases, the developers constantly iterated through intertwined activities, which contrasts established process models for (re-)engineering platforms that focus on waterfall-like processes (cf. Section 2.3.2). The iterations facilitated various activities, and were partly unavoidable, since developers may always locate new feature code during the actual transformation. This experience led to two major insights: First, as aforementioned, an updated, iterative process model for (re-)engineering variant-rich systems is required to better understand the corresponding practices and processes. Second, the interconnection between re-engineering activities made it hard to precisely document and assess the corresponding costs.

Unexpected efforts

We already mentioned that we faced different project properties that led to unexpected efforts. Besides these cost factors, we also experienced that features that appear to behave highly similar (e.g., menus of the Apo-Games), can differ heavily in their actual implementation. Arguably, this was the result of the original developers gaining more knowledge and re-implementing such features, without updating the variants from which the code was cloned. Consequently, the variants diverge during their co-evolution [Schultheiß et al., 2020; Strüber et al., 2019]. For this reason, we re-designed several features and implemented them from scratch instead of re-engineering them, which reduced redundancies and the codebase, but caused higher efforts.

Variability mechanisms

Finally, we want to summarize our experiences regarding variability mechanisms that we already sketched before. Regarding annotations, a regularly mentioned con is the obfuscation of source code [Apel et al., 2013a; Fenske et al., 2020; Medeiros et al., 2015; Schulze et al., 2013; Siegmund et al., 2012]. We experienced exactly the same problem, which we could tackle to some extent by refactoring the annotations into more disciplined forms. For composition, we also faced the known problem of comprehending related code that is separated into different modules [Krüger, 2018b; Krüger et al., 2019b; Siegmund et al., 2012]. This challenges the re-engineering, maintenance, and extension of the platform. Another problem for composition (i.e., feature-oriented programming) was the need for a basically complete configurator tool and its related artifacts (i.e., feature model). Without this, it is not possible to test the platform, but since features may be identified, located, or changed later on, this constantly required considerable rework. Moreover, re-engineering fine-grained features (e.g., in an individual `case` of a `switch`) towards composition, is challenging to impossible without major refactoring. In summary, we can only re-iterate our experience that annotations seem to be the more suitable and less expensive variability mechanism for a re-engineering project.

RO-E$_9$: Lessons Learned

From all of our cases, we elicited:

- *Confirming experiences regarding different benefits of platform engineering, namely an increased variant portfolio, improved code quality, facilitated variability management, a smaller codebase, and the removal of dead code.*
- *Experiences regarding known, but also novel, challenges with respect to missing tool support, deciding about features, intertwined activities, unexpected efforts, and variability mechanisms.*

3.3.5 Threats to Validity

Threats to validity

Our individual case studies and multi-case study design have partly opposing pros and cons regarding threats to validity [Bass et al., 2018; Leonard-Barton, 1990]. In the following, we briefly summarize the most important internal and external threats.

Internal Validity

Verification with original developers

The main threat to the internal validity of most of our cases (except Case 3) is that we could not verify our results with the original developer. While we have been careful while analyzing our subject systems and data, we cannot ensure that other researchers and particularly the original developers would obtain identical results. We mitigated this threat by several means. First, we always tested that the platforms we re-engineered could be executed, ensuring their correct behavior. Second, for Case 3 the original developer actually performed the re-engineering and we achieved similar results, improving the confidence in our findings. Finally, we used the multi-case design particularly to mitigate this threat, and used a reference study to improve our confidence in Case 1.

Data collection

Another threat to the internal validity is the granularity of the data we elicited. Moreover, most activities we identified are intertwined, which challenges precise cost measurements and their assignment to specific activities or cost factors. As we discussed in Section 3.1.1, this threatens our results and other researchers may find a different set of data more important to elicit. However, we built our selection of data on our previous insights and on discussions among the participating developers to mitigate this threat and elicit reasonable data. In addition, we defined a logging template that others may use or refine to verify our data.

External Validity

Subject systems

While a multi-case study design improves the confidence that our results are transferable, we still face threats to the external validity. Most prominently, we had no access to suitable benchmarking datasets of other researchers, and thus relied on cloned variants we contributed to the research community as well as one preprocessor-based variant-rich system. As a result, our findings may not be fully transferable to other real-world variant-rich systems, which may be larger, more complex, or have a higher degree of variance. Considering the missing datasets, we argue that the ones we contributed are highly valuable, particularly since they they stem from practice. With respect to the Apo-Games, we remark that open-source games exhibit similar development patterns and properties as other software [Businge et al., 2018]. Since our results revealed important problems that will only be more challenging for larger systems, we argue that our multi-case study provides valuable insights for practitioners and researchers despite this potential threat.

Tool selection

In each of our cases, we relied on a number of different tools, usually depending on the performing developers' preferences. While we often relied on tools that are established for platform engineering (e.g., FeatureIDE), we also had to develop our own tools to manage technologies not supported by established ones. Moreover, we experimented with various tools designed to support the re-engineering of variant-rich systems. Unfortunately, the involved developers usually considered these tools less helpful, if they could execute them at all. So, we may have relied on tooling that is neither established in practice nor reflects the most recent advancements in research. While this threatens the external validity, we argue that we relied on existing tools as far as possible and only extended these if absolutely necessary; essentially resembling how an actual organization could conduct a re-engineering project.

3.4 Summary

Chapter summary

In this chapter, we investigated the economics of software reuse. For this purpose, we first discussed how an organization can estimate the economics of (re-)engineering a variant-rich system, building upon cost models for software product-line engineering to elicit cost factors and discuss economic consequences. Building upon these insights, we then reported an extensive empirical study with concrete data on the economics of devolving variants with either reuse strategy. Finally, we reported a multi-case study to shed light in the economics and challenges of re-engineering variant-rich systems into a systematically managed platform.

Summarizing contributions

Overall, the contributions in this chapter provide guidance for practitioners to decide for, plan, and reason about (re-)engineering a variant-rich system. Since these are typically highly complex, large, and long-living systems, any decision related to a variant-rich system has long-term impact on an organization's structure and practices. We provide reliable experiences and empirical data that can help an organization to better understand the economics and impact of its decisions — ideally improving the confidence in such a strategical decision. For researchers, we highlight new directions to improve our understanding of (re-)engineering variant-rich systems and developing corresponding tools. Since these directions are based on real-world experiences, addressing them could immediately impact and support software-engineering practice. Abstractly, our results suggest the following core finding:

RO-E: Economics

Implementing some form of platform engineering is the economically most promising strategy for an organization to develop a variant-rich system.

Connection to other research objectives

Several themes appeared repeatedly throughout this chapter. While we cannot investigate all of them, we focused on a few that seemed to be particularly important for (re-)engineering variant-rich systems. First, we found that knowledge about a system is constantly mentioned as a key cost factor, not only in existing cost models and research, but also by our interviewees and multi-case study developers. Since knowledge seems to be one of the most important cost factors (e.g., during feature identification and location) and has rarely been investigated, we continue our research on this property in Chapter 4 (**RO-K**). Another important cost factor that is closely related to knowledge, is the used variability mechanism. In Chapter 5, we investigate to what extent feature traceability based on variability-mechanisms and similar techniques can facilitate the (re-)engineering of variant-rich systems by directly encoding feature knowledge into the source code (**RO-T**). Finally, we elicited various activities and described their relations to the properties we investigated. We sketched several processes and argued that a new process model as well as a detailed understanding of helpful practices are required, which we analyze in Chapter 6 by synthesizing from all of our findings (**RO-P**).

4. The Knowledge Problem

This chapter builds on publications at EASE [Krüger et al., 2020e], ESEC/FSE [Krüger, 2019a], ICSE [Krüger et al., 2018e], ICSME [Krüger and Hebig, 2020], SPLC [Krüger et al., 2017a], VaMoS [Krüger et al., 2018b], Empirical Software Engineering [Nielebock et al., 2019], and the Journal of Systems and Software [Krüger et al., 2019c].

Chapter structure

While investigating the economics of software reuse (cf. Chapter 3), we found that developers' knowledge (e.g., feature locations) is among the most important cost factors regarding the (re-)engineering of variant-rich systems. Compared to other cost factors with strong impact (e.g., delta, bugs, quality), the impact of developers' knowledge is more complex, challenging to understand, and gained less attention in research [Krüger, 2019a; Parnin and Rugaber, 2012] — which is why we investigate the importance of knowledge in this chapter (**RO-K**). First, we study what knowledge developers consider important about their system, and how well they can recall this knowledge (Section 4.1). Second, we investigate how developers forget the source code they worked on (Section 4.2), which is particularly important with respect to re-engineering and feature location [Krüger et al., 2019d,e]. Finally, we report our experiences and empirical studies regarding how developers can recover knowledge that is relevant for re-engineering (Section 4.3). The contributions in this chapter allow researchers and practitioners to better understand developers' knowledge needs during re-engineering projects. This is helpful to understand what information to document in what form, how to identify experts for a system, and from where to recover missing information. Moreover, our results indicate various open research opportunities regarding developers' knowledge in general, and in the context of re-engineering projects more specifically.

Conceptual framework of knowledge

We display a more detailed overview of our conceptual framework regarding knowledge in Figure 4.1. A *project* involves numerous pieces of *information* that are concerned with different *properties* related to the *systems* (e.g., what features are implemented), *processes* (e.g., how to fix bugs), and *developers* (e.g., who is responsible for what) involved in the project. To execute their activities, developers must have the necessary information stored in their *memory*, thus establishing their knowledge regarding the project. However, a developer's memory decays over time (i.e., they forget information) [Krüger et al., 2018e; Parnin, 2010; Parnin and Rugaber, 2012], which is why *documentation* of any form may be used to record and recover required information. Our other research objectives require knowledge to perform them correctly (also, traces record knowledge), and a better knowledge base can considerably facilitate them.

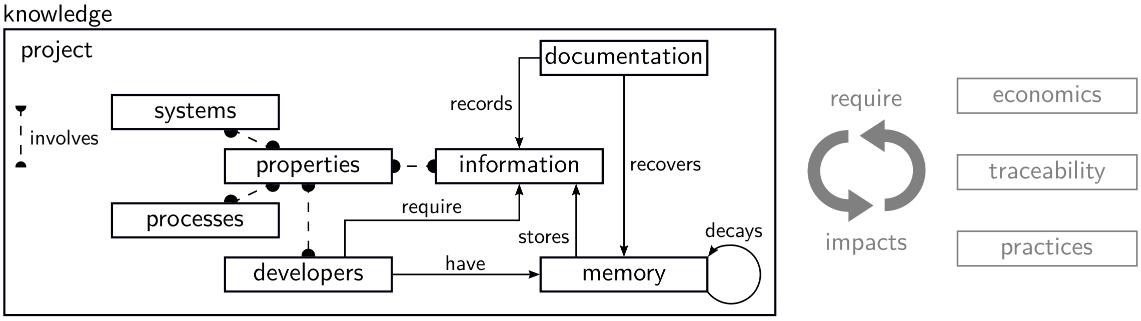


Figure 4.1: Details of the knowledge objective in our conceptual framework (cf. Figure 1.1).

4.1 Information Needs

Knowledge and re-engineering

While (re-)engineering any software system, developers must understand the involved assets (e.g., locating features in code) and their properties (e.g., dependencies between features). Consequently, a developer's knowledge impacts most activities and cost factors in software (re-)engineering, such as development efforts, software quality, and bug proneness [Anvik et al., 2006; Krüger and Berger, 2020b; LaToza and Myers, 2010b; Rus and Lindvall, 2002; Standish, 1984; Tiarks, 2011; von Mayrhauser and Vans, 1995]. Since the re-engineering of a variant-rich system involves a number of variants, usually developed by multiple developers over a longer period of time, it is important to understand which developers are still knowledgeable regarding which assets and properties. Understanding developers' remaining knowledge as well as their information needs, helps organizations to assign tasks, plan preparatory analyses, and manage their re-engineering projects.

Section contributions

In this section, we analyze developers' information needs based on a two-fold study [Krüger and Hebig, 2020]: We used a systematic literature review to identify related work, based on which we designed an interview survey we conducted with 17 developers. In detail, we are concerned with the following sub-objectives of **RO-K**:

RO-K$_1$ *Provide an overview of empirical studies related to developers' information needs.*

At first, we reviewed the related work on developers' information needs. Precisely, we focused on empirical studies that investigated what information developers asked for during their tasks, indicating that the corresponding knowledge was important, but not in their memory anymore. We used the results to obtain a basic understanding of information needs and scope our interview survey.

RO-K$_2$ *Investigate what knowledge developers consider important to remember.*

In our survey, we first analyzed what types of knowledge developers consider worth remembering. This is important to understand for what knowledge an organization needs to record or recover information to what extent to support (re-)engineering projects. For example, experts may remember the architecture of variants, which is why this would be important to record to facilitate the onboarding of novices and to not lose tacit knowledge.

RO-K$_3$ *Analyze the reliability of developers' memory.*

Building upon the previous sub-objective, we investigated to what extent developers' memory can be trusted regarding different types of knowledge. Our insights improve our understanding of how memory decay may affect different pieces of information, and thus support organizations and researchers in defining policies for recording and techniques for recovering relevant information. For instance,

> developers may reliably recall the features of a variant, but not their locations—which is why they could be a reliable information source for constructing a feature model, while feature location requires more effort.

Our interview guide, the results of our systematic literature review, and the anonymized responses to our interview survey are available in an evaluated open-access repository.[13] In the following, we describe the design of our systematic literature review and interview survey in Section 4.1.1 and Section 4.1.2, respectively. We then discuss threats to the validity of this study in Section 4.1.6. In Section 4.1.3, we present and discuss the results of our systematic literature review. Finally, we analyze the findings of our interview survey with respect to our last two sub-objectives in Section 4.1.4 and Section 4.1.5.

4.1.1 Eliciting Data with a Systematic Literature Review

Design adaptations

The goals of our systematic literature review were to (1) summarize empirical findings on developers' information needs; (2) ground our interview survey in empirical evidence; and (3) establish a dataset to which we could compare our results. As a result, we did not require a full-fledged systematic literature review, since we summarized and classified existing experiences [Kitchenham et al., 2015]. For this reason, we adapted our methodology as follows:

- We employed a snowballing search, starting from a set of relevant publications we knew. As a consequence, we may have missed publications that an automatic search could have identified. We decided to employ this adaptation to avoid the problems of automated searches [Kitchenham et al., 2015; Krüger et al., 2020c; Shakeel et al., 2018], and argue that we used an appropriate starting set for our snowballing.
- Instead of performing a quality assessment, we trusted the review process (IC_2). This is an established adaptation [Brereton et al., 2007], particularly since we intend to classify existing findings and use them to scope our own study [Kitchenham et al., 2015].
- Since they are not relevant for our study, we do not report the typical statistics that are part of a systematic literature review.

Using these established adaptations, we facilitated the conduct of our systematic literature review, without compromising its results with respect to our goals.

Search Strategy

Snowballing search

We started our snowballing search [Wohlin, 2014] with a set of five publications (asterisked in Table 4.1). To be as complete as possible, we employed backwards and forwards snowballing to identify further publications, without limiting the number of iterations. So, if we identified another relevant publication, we employed snowballing on that publication, too. We used Google Scholar for forwards snowballing (last updated on February 11, 2020).

Selection Criteria

Inclusion criteria

We focused on empirical studies that reported concrete questions developers had during their tasks, indicating that apparently relevant knowledge was missing. For this purpose, we defined the following inclusion criteria (IC):

IC_1 The publication is written in English.

IC_2 The publication has been peer reviewed.

[13] https://doi.org/10.5281/zenodo.3972404

IC_3 The publication reports an empirical study.

IC_4 The publication analyzes development and/or maintenance questions.

IC_5 The publication does one or both of the following:

(a) It identifies (ID) concrete questions that developers ask.
(b) It rates the importance (RI) and/or difficulty (RD) of questions.

Using these criteria, we ensured the quality of the included publications (IC_1, IC_2), and that they reported empirical findings (IC_3). We excluded publications that focus on questions on a highly specific topic that is not relevant for all (re-)engineering projects and activities, for instance, questions on API usages [Duala-Ekoko and Robillard, 2012], bug reports [Breu et al., 2009], code reviews [Pascarella et al., 2018], or concurrent programming [Pinto et al., 2015]. Finally, we only included publications with a systematic sample of concrete questions (IC_5), excluding publications that only exemplify questions, such as the one by Letovsky [1987].

Data Extraction and Synthesis

Data extraction

For each publication, we extracted its bibliographic data, all questions reported, and existing classifications proposed by the authors. Additionally, we extracted the number of participants, the questions involved, the research method, the scope (i.e., ID, RI, or RD according to IC_5), and additional comments (e.g., regarding availability of data, see Table 4.1) for each individual study reported in a publication. For analyzing our data, we relied on open-card sorting [Zimmermann, 2016]. Following this method, we started with identifying themes based on the classifications defined in the publications, aiming to unify 81 distinct classes of 420 questions. Then, we used the unified themes to reclassify all 465 questions (including those not previously classified) based on their texts. Using this reclassification, we checked the coverage of our unified classes and obtained a better understanding of the questions' contexts as well as relations. In the end, we synthesized existing rankings of questions according to their scope and our reclassification (i.e., architecture, meta, code). If a ranking in one publication was not normalized, we did so ourselves, for instance, the question ranked second out of 21 has a normalized ranking of 0.95. We averaged the normalized rankings of all questions in a theme.

Elicited Data

Elicited data

In the end, we included the 14 publications, comprising 17 individual studies, we display in Table 4.1. We can see that they involve varying numbers of participants, depending on the research methods and scopes. Nine of the involved studies identified questions, four rated the importance of questions, and two rated the difficulty of answering questions. Altogether, the publications include 465 unique questions and 81 classes. Unfortunately, three publications provide only a subset of all questions investigated, and their corresponding websites are not available anymore.

4.1.2 Eliciting Data with an Interview Survey

Motivation

The questions we identified through our systematic literature review indicate what knowledge developers require during their tasks, and thus suggest what information is important to record or recover for a re-engineering project. However, the importance of information also depends on a developer's specific task, their expertise, and their existing knowledge about a system. To better understand what knowledge developers consider important to remember, we conducted a qualitative interview survey [Wohlin et al., 2012]. For this purpose, we built upon the results of our systematic literature review and research on forgetting [Averell and

Table 4.1: Overview of the 14 publications on information needs. We started our snowballing search based on the asterisked publications.

reference	venue	#p	#q	research method	scope
Erdem et al. [1998]	ASE	—	60	newsgroup analysis	IQ
Sillito et al. [2006]*	FSE	25	44	observational study	IQ
Ko et al. [2007]	ICSE	17	21	observational study	IQ
		42	21	survey	RI
Sillito et al. [2008]	TSE	*<no relevant changes to Sillito et al. [2006]>*			
Fritz and Murphy [2010]*	ICSE	11	46 (78)[a]	interview survey	IQ
LaToza and Myers [2010a]*	PLATEAU	179	94	survey	IQ
LaToza and Myers [2010b]	ICSE	460	12	survey	RD
Tao et al. [2012]	FSE	33	8 (24)[b]	survey	IQ
		180	15	survey	RI
		180	15	survey	RD
Kubelka et al. [2014]	PLATEAU	6	7[c]	think-aloud sessions	IQ
Novais et al. [2014]*	VEM	42	11	survey	RI
Smith et al. [2015]*	ESEC/FSE	10	78 (559)[d]	think-aloud sessions	IQ
Al-Nayeem et al. [2017]	ICST	194	37	survey	IQ
Sharma et al. [2017]	CHASE	27	25	survey	RI
Kubelka et al. [2019]	ICPC	*<no relevant changes to Kubelka et al. [2014]>*			

#p: number of **p**articipants – #q: number of **q**uestions
IQ: **I**dentify **Q**uestions – RI: **R**ate **I**mportance – RD: **R**ate **D**ifficulty
[a] lists 46 questions, website with all 78 questions not available; [b] lists 8 of 24 questions;
[c] 7 new questions, others from previous work [Ko et al., 2007; Sillito et al., 2008];
[d] lists 78 questions, website with all 559 questions not available

Heathcote, 2011; Cohen and Conway, 2007; Kang and Hahn, 2009; Krüger et al., 2018e; Moran, 2016; Parnin and Rugaber, 2012].

Reasoning for interview

Our interview survey represents an empirical study of psychological aspects, namely human cognition. Such an empirical study faces a variety of potential threats due to humans' individual characteristics [Feldt et al., 2010; Krüger et al., 2018e; Siegmund and Schumann, 2015; Stacy and MacMillan, 1995], which are hard or even impossible to control or isolate. For instance, developers memory about a system (with which we are concerned) must first be established before we can investigate it, while we must also try to avoid biases that could be caused by giving away the purpose of our survey. To tackle such biases, we decided to ask our interviewees questions about a system they worked on before. Moreover, we decided to rely on an interview survey to obtain qualitative in-depth insights, limit faulty answers, and reduce the drop out rate (particularly since each interview took 1 to 2.5 hours) [Wohlin et al., 2012]. In summary, we decided to *conduct an interview survey* in which we asked *questions on information needs* about an *interviewee's system*. So, we report a descriptive empirical study, intending to understand a phenomenon (importance of knowledge and reliability of memory), similar to the publications we identified in our systematic literature review.

Interview Guide

General structure

In Table 4.2, we show an overview of our semi-structured interview guide. We use the identifiers throughout this section to refer to the questions we discuss. For each question, we show one example reference from our systematic literature review that comprised that or a similar question. In total, we defined 27 questions across five sections. During the interviews, we briefly introduced each section to the interviewees.

Self-assessment section

In the first section, we asked our interviewees to perform a self-assessment of the knowledge they still had with respect to their system. To obtain a more detailed understanding, we asked for individual assessments regarding their overall (OS_1), architectural (OS_2), meta (OS_3), and code (OS_4) knowledge. During our survey, we aimed to see whether our interviewees' self-assessments would change after reflecting about their system, which is why we repeated these questions after each of the next three sections. We remark that the analysis of self-assessments is a supportive means for another part of this dissertation, but it is not in its focus and we will only briefly summarize our insight in this regard.

Knowledge sections

In the next three sections, we adopted questions we identified during our systematic literature review to ask our interviewees about knowledge related to our unified themes. Consequently, we asked questions relating to the architecture, meta, and code knowledge of their systems. Note that we asked $C_{1–6}$ for three individual files, so each interview comprised three instances of each of these questions. We designed our interviews to involve a broader range of knowledge on different levels of detail, for instance, $C_{1–3}$ focus on more abstract code knowledge, while $C_{4–6}$ focus on code details. To improve our confidence in the responses' correctness, we involved several questions that we could verify against the actual system (e.g., A_6, M_6, C_6).

Importance section

In the last section, we asked our interviewees to think about our sub-objectives. We asked them what knowledge they consider important to remember by intuition (IK_1), by rating our unified themes (IK_2), and by rating each question we asked (IK_3). Finally, we asked them to elaborate on how they reflected about their knowledge (IK_4) and for any additional remark (IK_5). After this section, we evaluated the correctness (explained shortly) of their answers with each interviewee, who were now allowed to look at their system.

Design decisions

Our interview survey focused on developers' memory, which is why we did not allow our interviewees to investigate their code during the interview (which would initiate program comprehension) or analyze any other documentation. This also avoided biases with respect to questions related to concrete files or methods of the system, which the interviewer selected before the conduct. Similarly, the order and wording of our questions may be problematic for studying developers' memory. We decided for this setup and ordering of questions, particularly asking about the importance of knowledge at the end (but before evaluating correctness), to mitigate any confirmation biases. Namely, our interviewees may had focused on answering those questions correctly they considered important at the beginning to justify their decision. Finally, we investigated systems developed in a version-control system, allowing us to measure times, edits, and other pieces of information to evaluate some questions. However, several questions relate to more tacit and rarely documented knowledge, which is why we relied on our interviewees' knowledge as experts to evaluate those.

Conduct

Interview preparation

At least one interviewer conducted each interview with exactly one interviewee at a time. We explained to our interviewees that the interview was on program comprehension, without revealing our actual goals. Before conducting the interview, we asked each interviewee to allow us to access their version-control system (e.g., in advance via a link or by bringing their computer) to prepare our guide (e.g., selecting files and methods, analyzing times and edits). With respect to the selected files and methods, we aimed to involve such with different properties in terms of, for example, their size, last change, parameters, or positioning in the files system. After this preparation, we conducted the actual interview.

Table 4.2: Structure of our interview guide on knowledge needs.

id	example source	questions & answers (A)
section: overall self-assessment		
<asked 4 times: at the beginning and after each of the following three sections>		
OS_1	—	How well do you still know your system?
OS_2	—	How well do you still know the architecture of your system?
OS_3	—	How well do you know your code of the system?
OS_4	—	How well do you know the file *<name>*?
		$\mathbf{A}_{\mathbf{OS1\text{–}4}}$: rating from 0 to 100 %
section: architecture		
A_1	LaToza and Myers [2010a]	Can you draw a simple architecture of your system?
		$\mathbf{A}_{\mathbf{A1}}$: a drawn model *<updated after each other section>*
A_2	Smith et al. [2015]	Is a database functionality implemented in your system?
A_3	Sillito et al. [2006]	Is a user interface implemented in your system?
		$\mathbf{A}_{\mathbf{A2\text{–}3}}$: ○ yes: *<file>* ○ no
A_4	LaToza and Myers [2010b]	Can you name a file that acts as the main controller of your system?
		$\mathbf{A}_{\mathbf{A4}}$: ○ yes: *<file>* ○ yes: *<functionality>* ○ no
A_5	LaToza and Myers [2010a]	On which other functionalities does the file *<file>* rely?
		$\mathbf{A}_{\mathbf{A5}}$: open text
A_6	LaToza and Myers [2010a]	Can you exemplify a file/functionality you implemented using a library?
		$\mathbf{A}_{\mathbf{A6}}$: ○ yes: *<file>* ○ yes: *<functionality>* ○ no
section: meta-knowledge		
M_1	Novais et al. [2014]	When in the project life-cycle has the file *<file>* last been changed?
		$\mathbf{A}_{\mathbf{M1}}$: open text
M_2	LaToza and Myers [2010a]	Can you exemplify a file which has recently been changed and the reason why (e.g., last 2-3 commits)?
		$\mathbf{A}_{\mathbf{M2}}$: ○ yes: *<file>* *<reason>* ○ yes: *<file>* ○ no
M_3	Fritz and Murphy [2010]	Can you point out an old file that has especially rarely/often been changed?
		$\mathbf{A}_{\mathbf{M3}}$: ○ yes: *<file>* ○ no
M_4	Fritz and Murphy [2010]	How old is this file in the project life-cycle and how often has it been changed since the creation?
M_5	LaToza and Myers [2010a]	Who is the owner of file *<file>*?
M_6	LaToza and Myers [2010a]	How big is the file *<file>*?
		$\mathbf{A}_{\mathbf{M4\text{–}6}}$: open text
section: code comprehension		
<for three> files: a) *<file>*; b) *<file>*; c) *<file>*		
C_1	LaToza and Myers [2010a]	What is the intent of the code in the file?
		$\mathbf{A}_{\mathbf{C1}}$ *<per file>*: open text
C_2	LaToza and Myers [2010a]	Is there a code smell in the code of the file?
		$\mathbf{A}_{\mathbf{C2}}$ *<per file>*: ○ yes: *<smell>* ○ yes ○ no
C_3	Sillito et al. [2006]	Which data (in data object or database) is modified by the file?
		$\mathbf{A}_{\mathbf{C3}}$ *<per file>*: open text
<for three> methods: a) *<from file a>*; b) *<from file b>*; c) *<from file c>*		
C_4	LaToza and Myers [2010a]	Which parameters does the following method need?
C_5	Kubelka et al. [2014]	What type of data is returned by this method?
C_6	Sillito et al. [2006]	Which errors/exceptions can the method throw?
		$\mathbf{A}_{\mathbf{C4\text{–}6}}$ *<per method>*: open text
section: importance of knowledge		
IK_1	—	Which part of your system do you consider important?
		$\mathbf{A}_{\mathbf{IK1}}$: open text
IK_2	—	Which type of the previously investigated types of knowledge do you consider important?
		$\mathbf{A}_{\mathbf{IK2}}$: ○ architecture ○ meta ○ code
IK_3	—	Which of the previous questions do you consider important or irrelevant when talking about familiarity?
		$\mathbf{A}_{\mathbf{IK3}}$ *<(per A_i, M_i, C_i)>*: ○ irrelevant ○ half/half ○ important
IK_4	—	What do you consider/reflect about when making a self-assessment of your familiarity?
IK_5	—	Do you have additional remarks?
		$\mathbf{A}_{\mathbf{IK4\text{–}5}}$: open text

Table 4.3: Overview of our interviews on knowledge needs in order of conduct.

#	**area**	**domain**	**programming languages**	**loc**	**#d**
1	academia	document parser	Java	<10k	2
2	academia	Model editor	Java	<10k	3
3	academia	security analysis	Java	<10k	1
4	academia	machine learning	Python	<10k	4
5	academia	static code analysis	Java	<10k	1
6	industry	web services	JavaScript, PHP	10k–100k	2
7	industry	web services	PHP	>100k	1
8	academia	development environment	Java	>100k	6
9	academia	databases	C++	>100k	3
10	academia	static code analysis	Java	<10k	1
11	industry	android app	Java	10k–100k	1
12	industry	enterprise resource planning	C#	>100k	6
13	academia	static code analysis	Java	<10k	1
14	academia	web services	Ruby	<10k	1
15	open-source	geometry processing	Rust	<10k	1
16	industry	static code analysis	OCAML	<10k	2
17	open-source	traceability	Java	<10k	5

#d: number of active developers

Number of interviews

Instead of stopping at a fixed number of interviews, we relied on saturation as a reasonable stop criterion for qualitative research [Wohlin et al., 2012]. Namely, we stopped after we found that the last three interviews we conducted had no significant impact on the average responses anymore. This led to a total of 17 interviews, which is also a comparable sample size to similar studies we identified in our systematic literature review (i.e. interview, observational, and think-aloud studies in Table 4.2). Moreover, the results between our systematic literature review and interview survey that are concerned with the same questions are comparable, improving our confidence that the number of interviews is reasonable.

Interviewees

Interviewees

Following recommendations of Wohlin et al. [2012], we focused on including interviewees based on differences rather than similarity, aiming to derive insights from a diverse sample. For this purpose, we invited recent and former collaborators form various countries (e.g., Germany, Sweden, France). All of them had between five and ten years of programming experiences and have been active programmers of their system. Five of our interviewees were working full-time in industry and four had worked in industry before. Three interviewees were female. In Table 4.3, we show an overview of each system, with the area referring to the domain of the system, not the interviewee. We can see that the systems span a variety of domains and programming languages. Also, they have been developed for three months to more than ten years by one to six regular developers. Interestingly, most of the systems had no dedicated documentation, and thus relied heavily on the developers' knowledge. One limitation of our interview survey is that most systems have been relatively small with respect ot their size and the number of regular developers (even though one system had over 50 contributors over time). As a consequence, we can generalize our observations mainly for smaller systems, even though we interviewed a diverse sample of developers.

Correctness Evaluation

Procedure

To evaluate the correctness of answers, we re-iterated through all questions with each interviewee, this time also investigating the system and its version-control data. This procedure had two major benefits. First, feasibility: Many of our questions can be answered solely or at least more easily by our interviewee, who had the respective knowledge and could access the system. Second, uncertainty: For other questions, we had to evaluate the correctness of answers compared to the interviewee's understanding of the system, since we cannot assume that there is a single correct answer that is valid to every developer of a system. For instance, the presence of code smells may be up for debate between developers, depending on what they consider to represent a code smell.

Rating scheme

We evaluated questions $A_{2–6}$ and $M_{1–6}$ as correct (1 point), partially correct (0.5 points), or incorrect (0 points). Identically, we rated the three instances for $C_{1–6}$ individually and averaged the points for each question. For instance, if a code smell was correctly described for one file (1 point) and incorrect for the other two files (0 points each), the interviewee's score for C_2 was 0.33. Note that we had five cases in which we could not ask or properly correct each question for all files, due to unsuitable methods (two cases) and lost commit histories during repository migrations (three cases).

Architectural model

Rating the correctness of the architectural model (A_1) was a special case. Due to the subjective perspective of what an architectural model should comprise, we allowed our interviewees to update the model during the interview. In the end, we rated the correctness of the final model based on three questions:

1. Did the interviewee consider the final model as correct after looking at their system?
2. Was the model system-specific? For example, this was not the case for a generic model-view-controller architectural model without any further system specifics.
3. Did the interviewee refine or correct the model? We defined refinements as additions (e.g., of components) made to enrich the model. In contrast, corrections refer to the interviewee removing (crossing out) or substituting elements of the model.

If the interviewee and interviewer agreed that an architectural model was correct, system-specific, and received refinements only (but no corrections), we assigned 1 point. For models that were not system-specific, we assigned 0 points. In a single case, we assigned 0.5 points for an architecture that was not system-specific, but included annotations that described specific technologies used within the system. We also assigned 0.5 points if the architectural model was system-specific, but received corrections.

4.1.3 RO-K_1: Studies on Developers' Information Needs

Information needs

In the following, we describe and discuss our results of analyzing the publications we show in Table 4.1. We display the themes and according classifications we extracted from the publications in Figure 4.2. In Figure 4.3, we summarize the rankings we synthesized from those presented in the publications.

Results

Themes from classifications

On the left side of Figure 4.2, we show the seven themes we synthesized from the authors' classifications, involving 420 questions and 81 classes. During this phase, we decided to discard the questions of Erdem et al. [1998] during our more detailed analysis, since they are too *general* (e.g., "What does it do?") and can be asked for anything. In contrast, we found that the themes *testing*, program *comprehension*, and *other* subsume questions that relate to the remaining three themes.

Reclassification

As a consequence, we decided to reclassify each individual question using these three themes (note that we use the abbreviations throughout this section):

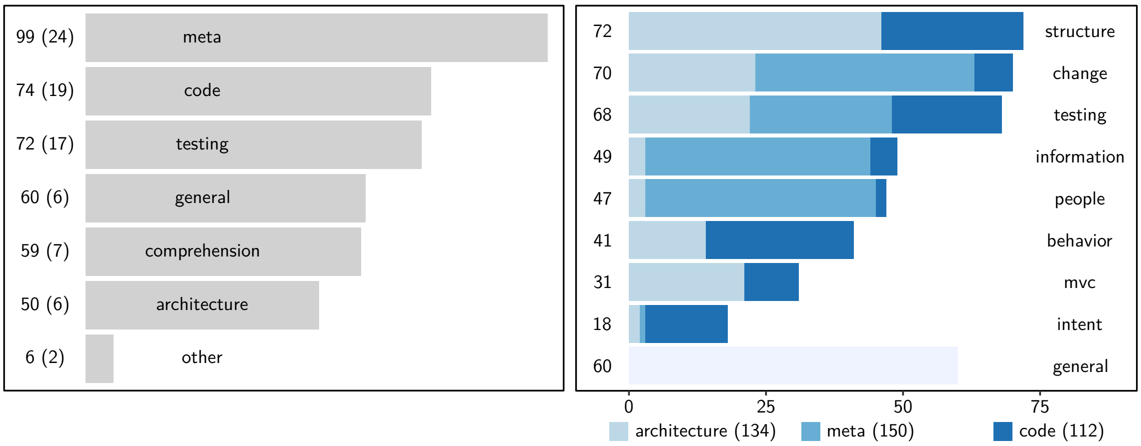


Figure 4.2: Overview of the knowledge themes we identified from the publications in Table 4.1. The themes on the left are based on the classifications defined in the publications, with the numbers indicating the number of associated questions (and classes). The themes on the right are based on our reclassification.

A *Architecture* questions focus on the structure of a system, such as (feature) dependencies, APIs, or architectural patterns (e.g., "Who can call this?" [Smith et al., 2015], "[Which] API has changed?" [Fritz and Murphy, 2010]).

M *Meta* questions focus on the context of a system, such as ownership, (variant) evolution, or developer roles (e.g., "How has it changed over time?" [LaToza and Myers, 2010a], "Who owns a test case?" [Fritz and Murphy, 2010]).

C *Code* questions focus on the implementation of a system, such as code smells, bugs, or feature locations (e.g., "What are the arguments to this function?" [Sillito et al., 2006], "How big is this code?" [LaToza and Myers, 2010a]).

These themes represent a suitable abstraction, but each question may belong to multiple themes, for instance, involving the evolution (*meta*) of a class (*code*). For simplicity, we reclassified each question to the theme we considered predominant, and derived sub-themes for a more detailed understanding. We display our final reclassification (including sub-themes) of all 456 questions — adding the 36 that were not classified by the authors — on the right side of Figure 4.2. Again, we selected the predominant sub-theme for each question, for instance, "Who has made changes to [a] defect?" [Fritz and Murphy, 2010] relates mainly to *people*, but also to *testing* and *change*.

Rankings Finally, we synthesized the rankings of difficulty and importance of questions reported in the publications (cf. Figure 4.3). For this purpose, we used our three main themes and merged the relative ranking (i.e., a value between 0 and 1) of each question in these publications. We found only two examples for the difficulty of answering meta questions, which is why we cannot judge this ranking confidently. However, for the remaining themes, and particularly the importance of questions, we identified several ranked questions (numbers below the box-plots) we could build on.

Discussion

Themes Not surprisingly, most questions for architecture and code relate to a system's *structure*, *behavior*, the mode-view-controller (*mvc*) pattern, and *testing*. Code questions also relate heavily to the *intent* that should be implemented in the source code of a system. Meta questions relate mostly to *testing*, *change*, *information*, and *people*. Our reclassification is

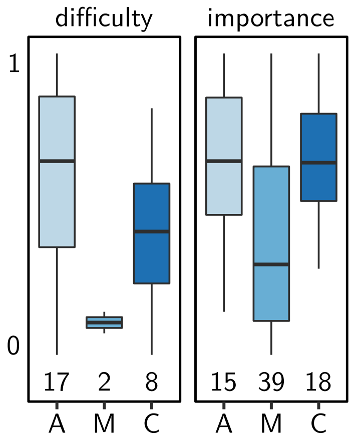


Figure 4.3: Synthesized rankings ([0,1]) of architectural (A), meta (M), and code (C) questions (numbers below each box-plot) based on the publications in Table 4.1.

similar to the one we synthesized from the authors' classifications (e.g., a similar number of questions for testing and meta knowledge). We can see that most questions relate to meta knowledge, while the fewest relate to code knowledge — potentially because a developer can analyze the code to recover knowledge during program comprehension. In contrast, architectural and meta knowledge are often not directly accessible or not explicitly recorded, which could be the reason for more questions in this regard.

Difficulty and importance

Considering Figure 4.3, we can see that the difficulty and importance of answering questions seem to be related, which has also been found by Tao et al. [2012]. Also, meta questions are the largest theme, but they seem less important than questions of the other two themes — which is not caused by their sheer number. For example, the seven (of 21) meta questions ranked in the study of Ko et al. [2007] are all among the lowest nine, even though they appeared as frequently as the questions from the other two themes. These insights indicate that questions on meta information may appear frequently, but can be recovered more easily (e.g., from version-control data) or are simply not as important to know.

RO-K$_1$: Studies on Developers' Information Needs

We reviewed existing studies on developers' information needs and learned:

- *Architecture, meta, and code are general themes for classifying questions.*
- *The difficulty and importance of the questions in a theme seem to relate.*
- *How often questions of a theme occur seems unrelated to the theme's importance.*
- *Developers ask fewer questions about source code.*
- *Meta questions seem less important compared to questions of the other themes.*

4.1.4 RO-K$_2$: The Importance of Knowledge

Importance of knowledge

We analyzed what knowledge developers consider important to memorize about their system based on the answers to questions IK$_{1–3}$. In Figure 4.4, we summarize our interviewees' perception of whether it is important to memorize information of a knowledge theme. We provide the more detailed summary of our interviewees' perception of the importance of each individual question, and their correctness in answering these, in Figure 4.5.

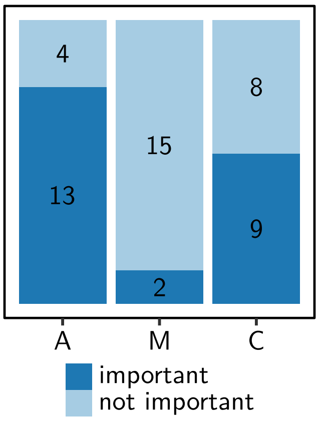


Figure 4.4: Summary of our interviewees' perceived importance of knowledge themes (IK$_2$; A: architecture, M: meta; C: code).

Results

Open perception

Since IK$_1$ was an open-text question, we employed open coding [Seaman, 1999; Shull et al., 2008] to extract 35 codes from the answers. Using this methodology, we aimed to understand what knowledge our interviewees perceive important without focusing on predefined themes. We found codes for architecture in seven answers, and six more codes for closely related themes, namely dependencies, APIs, extension mechanisms, and own extensions — all capturing the structure and variability (i.e., extensions) of a system. Another regular theme is connected to understanding a system's behavior by analyzing its intent (3 codes) during program comprehension (1 code). Further individual codes were related to bug locations, a system's main controller, domain-specific knowledge, and code conventions.

Perception of themes

Then, we asked each interviewee to rate the importance of remembering knowledge with respect to our high-level themes (IK$_2$). As we can see in Figure 4.4, most interviewees considered architectural knowledge as the most important to remember. While only two of our interviewees considered meta knowledge as important, roughly half of them perceived code knowledge as important. Interestingly, the overall perception aligns well with the results of our systematic literature review (cf. Figure 4.3) and the interviewees' answers to IK$_1$. In all cases, architecture was the most prominent theme, followed by code and meta. We remark that we expected meta knowledge to be perceived less important, due to the smaller systems for which we interviewed developers. However, the alignment to the other publications improves our confidence that our findings are useful beyond such systems.

Perception of architecture questions

We can see the same pattern for the low-level ratings of the importance of remembering knowledge with respect to each individual question (IK$_3$). With the exception of A$_6$ (knowing files that rely on libraries), more than 50 % of our interviewees considered all architectural questions important. Particularly, most interviewees agreed on the importance of knowing the architectural model (A$_1$) and the main controller of the system (A$_4$). The other two concepts of the model-view-controller pattern, user interface (A$_3$) and data storage (A$_2$), were perceived similarly important to know. In most cases when interviewees did not perceive this knowledge important, their system did not implement these concepts. Slightly more than half of our interviewees considered it important to know the functionalities and dependencies a file has (A$_5$).

Perception of meta questions

Regarding meta knowledge, only knowing recently changed files (M$_2$) and file owners (M$_5$) was considered important by more than half of our interviewees. Roughly a third of our interviewees perceived it important to know which older files face rare or frequent

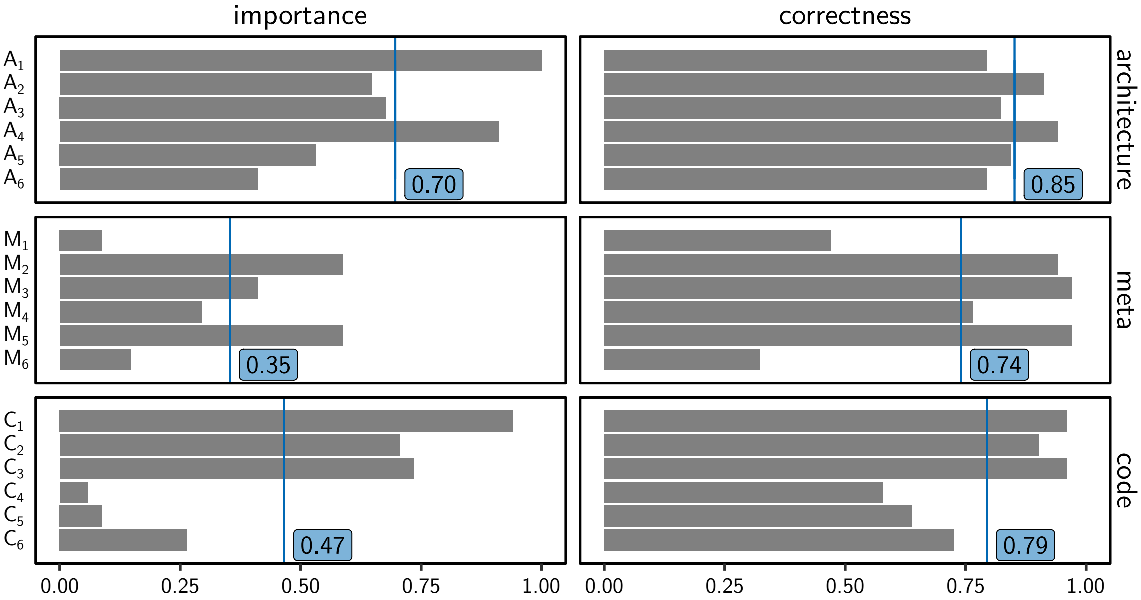


Figure 4.5: Summary of our interviewees' perceived importance (IK_3) and their correctness for each question. The blue lines illustrate the mean value for each knowledge theme.

change (M_3, M_4). Knowing the size of a file (M_6) or when it was last changed (M_1) was perceived unimportant by most interviewees. Interestingly, more than two of our interviewees considered it important to know the answers to questions $M_{2–6}$, even though only two considered meta knowledge important. This contradicts our expectations for smaller systems, and may indicate again that our findings are relevant for larger systems, too.

Perception of code questions

We can see in Figure 4.5 that code questions represent two groups regarding whether our interviewees considered them important to recall. On the one hand, over 70 % of our interviewees perceived it important to know the intent of a file (C_1), what data is manipulated in a file (C_3), and whether a file comprises code smells (C_2). On the other hand, few of our interviewees perceived it important to know implementation details of a method ($C_{4–6}$). However, our question relating to exceptions (C_6), and thus software quality as well as testing, was considered comparably important.

Discussion

Architectural knowledge

The results of our systematic literature review and interview survey indicate that architectural knowledge is important to remember, but the corresponding questions are perceived difficult to answer in existing studies. This suggests that developers may aim to memorize architectural knowledge, which then becomes tacit knowledge and decays over time. Our findings imply that involving experts of one or more variants arguably facilitates re-engineering a variant-rich system. Moreover, it is important to record architectural knowledge (e.g., platform architecture model, feature model) to support new developers that are less knowledgeable. Such measures can reduce the costs of (re-)engineering projects, improve traceability, and facilitate development practices by tackling particularly the knowledge needs of developers that have not worked on the cloned variants or platform before.

Meta knowledge

Our results indicate that meta knowledge is the least important theme, but we found interesting discrepancies between individual questions. While the publications we reviewed suggest a similar tendency, the perceived importance may be biased by (1) the smaller number of developers our interviewees' collaborated with and (2) the direct availability of meta information in their version-control systems. However, the questions highlighted as more important are also highly relevant for re-engineering projects: Knowing the owner of a file can help to identify experts, while knowing stable or constantly evolving code helps to scope the platform and limit investments, for instance, by focusing on established features first or identifying features that should be re-implemented from scratch.

Code knowledge

Finally, our results indicate that code knowledge ranks between architectural and meta knowledge. Interestingly, our analysis revealed that this may be caused by different abstraction levels of code knowledge. It seems that developers consider it important to know the intent or flaws of their code, while they are less interested in code details that may be too detailed and easily recoverable. In the context of re-engineering projects, this suggests that developers can be a reliable source to understand the general purpose and potential quality problems of code. However, developers do not memorize code details, which again highlights the challenge of feature location and the value of establishing feature traceability to facilitate the recovery of detailed code knowledge. Moreover, this indicates that developers consider it more important to memorize the problem space of a variant-rich system than its solution space.

RO-K$_2$: The Importance of Knowledge

We learned that developers consider it most important to memorize abstract knowledge, namely a system's architecture and the intent of the code. In contrast, meta and detailed code knowledge are perceived less important to remember, arguably because they are encoded in, and easier to recover from, the code or version-control system.

4.1.5 RO-K$_3$: Reliability of Developers' Memory

Correctness

On the right side of Figure 4.5, we display the average correctness of the answers our interviewees gave to each question. Overall, our interviewees performed quite well for all questions, resulting in an average correctness of 80 %. In the following, we discuss the correctness for each theme and with respect to the perceived importance of the questions.

Results

Correctness for themes

We can see in Figure 4.5 that at least 79 % of our interviewees answered each architectural question correctly. Particularly a system's main control file (A_4) and its data storage (A_2) could be named correctly by most interviewees. For meta-knowledge, half of our questions (M_2, M_3, M_5) were answered correctly considerably more often (80 %) than the others. Interestingly, only two questions could be answered correctly by fewer than half of our interviewees, and both are related to meta-knowledge (M_1, M_6). Finally, we can again find the previously identified pattern with respect to code questions. Questions on a higher level of abstraction ($C_{1–3}$), such as a file's intent, could be answered correctly by most interviewees. In contrast, our interviewees' correctness regarding questions about code details ($C_{4–6}$) ranges from 55 % to 75 %.

High- and low-level code questions

Seeing the reappearing pattern for code knowledge, we investigated whether it is a meaningful observation. For this purpose, we display our interviewees' correctness for both levels of abstraction and their combination in Figure 4.6, also involving the days since each interviewees' last edit to the corresponding file. We remark again that we had to remove five entries for this analysis (cf. Section 4.1.2), resulting in 46 paired data points. Still, we can see that our interviewees could correctly answer more abstract (i.e., high-level) questions more often — independently of the time that passed and the consequent memory decay since their last edit of the corresponding code. To understand whether our observation is significant, we used hypothesis testing and the R statistics environment [R Core Team, 2018–2020]. First, we tested whether our data is independent of the time that passed using

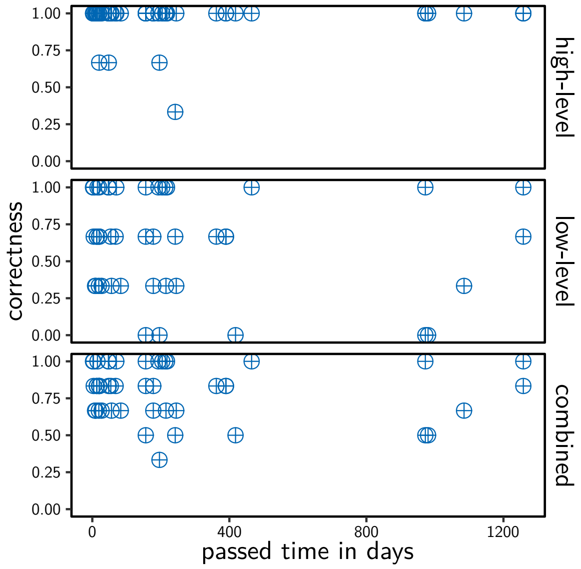


Figure 4.6: Comparison of our interviewees' correctness regarding code questions, separated by level of abstraction (high-level: C_{1-3}, low-level: C_{4-6}, combined: C_{1-6}).

Kendall's τ [Kendall, 1938; Sen, 1968], since it allows to test non-normal distributions (i.e., the correctness). We found neither a relevant ($-0.136 < \tau < 0.005$) nor a significant ($p > 0.05$) correlation between time and correctness for any group. However, for the low-level questions, we identified a negative tendency (-0.136). Since this indicates that the differences mainly depend on the level of abstraction, we tested this hypothesis using the Wilcoxon signed-rank test [Wilcoxon, 1945] for non-normal distributions of paired data. Comparing the means of both distributions, the test indicated that the level of abstraction leads to significant differences in terms of correctness ($p < 0.001$). Thus, we continue under the assumption that developers are better at recalling more abstract code knowledge.

Importance and correctness

We already found that the difficulty and importance of answering a question seem to relate. Building on this insight, we now analyze whether the importance also relates to our interviewees' correctness when recalling an answer. In Figure 4.7, we display the relation between these two properties in a more intuitive and concise way than in Figure 4.5. We can see that the perceived importance and the correctness for each question seem to relate. Particularly, those questions perceived as unimportant were answered less often correctly (e.g., M_1). Also, our interviewees achieved above 75 % correctness for all questions that more than 50 % perceived important. Again, we tested this observation (i.e., developers memorize knowledge on questions they perceive important) using Kendall's τ. The test showed a significant, moderately positive correlation ($\tau = 0.508$, $p < 0.005$) between importance and correctness with a confidence interval of 0.95. Since we conducted a qualitative study, we had few data points for this test, which is why the statistical power is low (0.575).

Self-assessments

Finally, we briefly summarize our findings regarding the reliability of our interviewees' self-assessments. Interestingly, throughout all interviews only a single interviewee increased their score (+5 %) and eight kept it steady, leading to a decrease on average (-13.75 %). Existing guidelines and studies in empirical software engineering assume that self-assessments can be used to predict developers' knowledge and expertise [Feigenspan et al., 2012; Ko et al., 2015; Siegmund, 2012; Siegmund et al., 2014]. We used Kendall's τ to test this hypothesis on our data, which showed no significant correlation, but a small positive tendency (initial self-assessment $\tau = 0.176$, final self-assessment $\tau = 0.032$). Since, we have a small sample size (17 data points), we could only show a strong correlation with enough statistical power

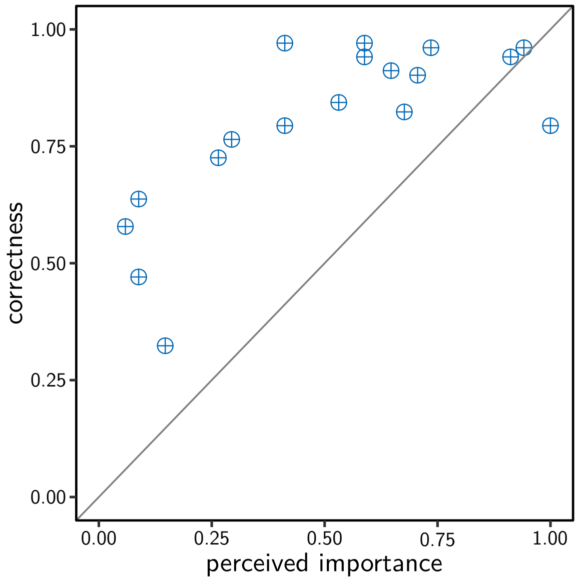


Figure 4.7: Average correctness and perceived importance for each question (data points).

(80 %). Still, we found a positive tendency that aligns to the hypothesis raised and evaluated in previous works, on which we built for other studies in this dissertation.

Discussion

Importance versus correctness

Overall, our interviewees performed well when answering questions from their memory. They performed particularly well for the knowledge they considered important — which they seem to intend to memorize, but may not record explicitly. Considering re-engineering projects, these findings have several implications. For example, the involved developers (e.g., experts, novices) require individual support to improve their knowledge (e.g., techniques for architecture recovery). However, aligning to the cost factors we identified before, our findings suggest to involve experts of the existing variants to guide the planning and conduct of re-engineering projects. Due to the variations in the perceived importance and actual correctness for different types of knowledge, an organization must understand which knowledge is relevant and how to obtain it (e.g., considering meta knowledge).

Abstractions versus details

We previously found an interesting difference between code details and code abstractions with respect to their perceived importance. Considering correctness, we identified a negative tendency between code details and the time that passed since developers' last edit to the respective code — indicating that developers are better at remembering code abstractions. Reflecting also on our previous findings, this insight has direct implications. As a concrete example, experts could be a reliable information source to support feature identification, which mainly requires abstract knowledge of the code (i.e., features in the domain space). However, for the more costly feature location (i.e., assets in the solution space), they arguably face similar problems as other developers (e.g., novices). Consequently, understanding whether developers have detailed code knowledge, implementing feature traceability, and providing additional tools to recover information have immediate impact on the economics and success of a re-engineering project.

RO-K$_3$: Reliability of Developers' Memory

Our results suggest that developers can reliably answer questions about their systems from memory, which means that they can support the planning of a re-engineering project. Still, since they are better at remembering abstract knowledge and do not seem *to record the corresponding information, an organization should implement recording strategies (e.g., feature traces) to guide novices and use reverse-engineering techniques.*

4.1.6 Threats to Validity

Threats to validity

As already mentioned, an empirical study on human memory involves various threats to validity, since it is impossible to account for all human characteristics. In the following, we briefly summarize some of the most relevant threats.

Internal Validity

Background factors

Our interview survey was concerned with human characteristics (i.e., developers' memory, knowledge, opinions) that are influenced by numerous background factors we can hardly control, such as age, motivation, or a subject's individual memory performance. Similarly, the questions we used may not be ideal for our interview survey, could be misunderstood, or were in an inappropriate order. We mitigated such threats by building upon empirical evidence to select questions, conducting the interviews face-to-face to clarify confusions, pondering unavoidable biases against each other, and testing our interview guide beforehand.

External Validity

Generalizing beyond smaller systems

Our interview survey was qualitative, had a small sample size, and involved mainly smaller systems. This threatens the external validity of our results. We mitigated such threats by interviewing a diverse sample of developers, using saturation as stop criterion, and comparing our findings against existing publications. While our results seem to be transferable to larger systems, we have to be careful with overgeneralizing them beyond smaller systems.

Context of information needs

Another threat to our interview survey is that we considered knowledge in general. However, a developer's information needs and the knowledge they consider important may depend, for instance, on their concrete task (e.g., adding a feature versus fixing a bug) or other context properties. Consequently, our results may change if we analyze information needs in different contexts — which we intend to do in future work.

Conclusion Validity

Methodology

We conducted a systematic literature review to identify questions for constructing our interview survey. Since we reclassified all questions into themes, other researches may achieve different results. We mitigated this threat to the conclusion validity by also analyzing the individual questions, comparing our results to the publications we identified, and cross-checking our results. In addition, we followed established guidelines for our research methods [Kitchenham et al., 2015; Wohlin, 2014; Wohlin et al., 2012; Zimmermann, 2016] to ensure that we obtained reliable results. Finally, all of our data is publicly available to allow other researchers to evaluate and replicate our study.

4.2 Memory Decay

Memory decay

In the previous section, we found that developers can reliably recall different types of knowledge depending on various factors, and other studies show similar findings in other settings [Fritz et al., 2007, 2010; Kang and Hahn, 2009]. To understand which developers can be a reliable information source or can perform a task more efficiently, several techniques have been proposed to identify experts for a specific part of a system [McDonald and Ackerman, 2000; Minto and Murphy, 2007; Mockus and Herbsleb, 2002; Oliveira et al., 2019; Schuler and Zimmermann, 2008]. While some of these works are concerned with developers' knowledge about code, we are not aware of detailed analyses of how memory decay impacts developers' code knowledge, and thus the program comprehension required during feature location. Consequently, understanding developers' memory decay regarding source code is critical in re-engineering projects (cf. Chapter 3), for instance, to decide who has the most expertise to locate and re-engineer a feature.

Section contributions To tackle this problem, we [Krüger et al., 2018e] investigated whether we can use the forgetting curve of Ebbinghaus [1885] (explained shortly) in software engineering. We focus on detailed code knowledge, since we found that it seems to be harder to recall, and on identifying factors that impact developers' memory. For this purpose, we define the following three sub-objectives of **RO-K**:

RO-K$_4$ *Analyze the impact of repetition, ownership, and tracking on developers' memory.*

Numerous factors impact a person's memory, such as individual characteristics or properties of the artifact that is remembered. Considering research on learning, forgetting, and expertise identification in software engineering, we focus on three factors for software (re-)engineering: (1) how often a developer worked on the code (*repetition*); (2) how much of the code a developer implemented themself (*ownership*); and (3) how extensively a developer tracks others' changes to their code (*tracking*). Having empirical evidence on the impact of these factors helps organizations and researchers to understand how expert developers can be identified—which is often done based on educated guesses instead of empirical evidence.

RO-K$_5$ *Understand developers' memory strength.*

To apply Ebbinghaus' forgetting curve, we have to understand developers' memory strength. We use our data and findings regarding the impact of the previous factors to approximate how well developers can memorize their code. Our findings help researchers to understand and estimate to what extent developers' memory is reliable after a period of time.

RO-K$_6$ *Explore the applicability of Ebbinghaus' forgetting curve in software engineering.*

Finally, we explore whether we can apply Ebbinghaus' forgetting curve to software engineering. We compare different memory strengths to account for the previous factors and discuss the results. The findings help researchers measure developers' memory decay and adapt forgetting curves more precisely to software engineering, which can guide organizations in better understanding which developers are still experts for re-engineering a certain piece of code.

All anonymized survey responses are available in an open-access repository.[14] Next, we describe the methodology of our online survey in Section 4.2.1 and the threats to its validity in Section 4.2.5. In Section 4.2.2, we discuss the impact of the three factors we analyzed on developers' memory. Then, we discuss developers' memory strength (cf. Section 4.2.3) to finally apply Ebbinghaus' forgetting curve to our data (cf. Section 4.2.4).

4.2.1 Eliciting Data with an Online Survey

Methodology To address our research objectives, we constructed an online survey. In the following, we first introduce Ebbinghaus' forgetting curve before detailing our survey setup and participants.

[14] https://bitbucket.org/Jacob_Krueger/icse-2018-data

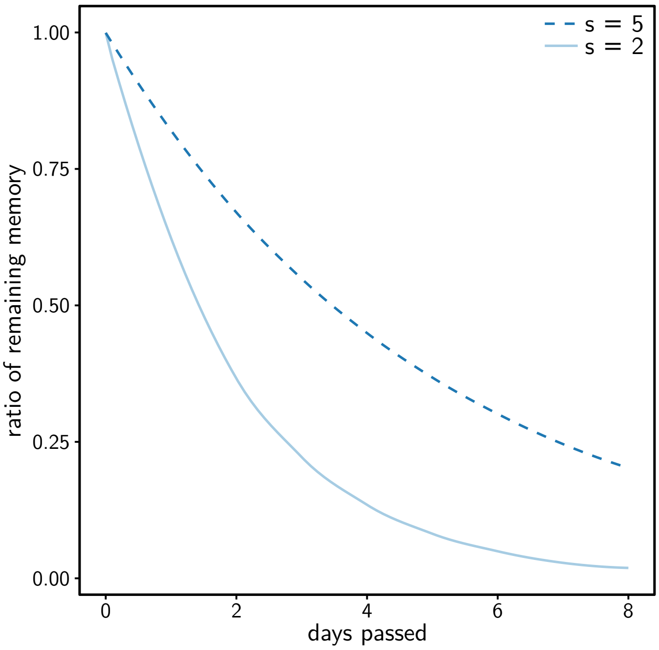


Figure 4.8: Examples of forgetting curves for memory strengths (s) of 2 and 5.

Ebbinghaus' Forgetting Curve

Forgetting curves

A developer's memory is not in a consistent state: They can gain or recover knowledge (e.g., during program comprehension, by reading documentation), but also forget it (i.e., memory decay). Consequently, a developer becomes less knowledgeable about their system over time with respect to the parts they do not work on or those someone else changes. The process of forgetting is well-investigated in psychology, resulting in different forgetting models and curves that are intended to approximate how humans forget an artifact [Averell and Heathcote, 2011; Jaber and Sikström, 2004a,b; Murre and Dros, 2015; Nembhard and Osothsilp, 2001]. For our study, we rely on the forgetting curve proposed by Ebbinghaus [1885]. While it is an older forgetting curve, it has been replicated in recent studies, performs similar to other forgetting curves, and thus is established in psychology [Averell and Heathcote, 2011; Murre and Dros, 2015].

Ebbinghaus' forgetting curve

Ebbinghaus' forgetting curve describes an exponential decay of a subject's memory, as we show in Equation 4.1.

$$R = e^{-\frac{t}{s}} \tag{4.1}$$

We can see that the memory retention rate (R) depends on a subject's individual memory strength (s) and the time (t) in days that passed between studying the artifact and the memory test. In Figure 4.8, we exemplify forgetting curves for two different memory strengths. We can see that a higher memory strength (s = 5, dashed dark blue line) leads to a slower retention rate, which means that a subject's memory lasts longer. For instance, with that memory strength, a subject loses 46 % of its memory regarding the artifact within three days. The memory strength is individual for every subject and depends on several factors, such as the artifact or learning effects.

Survey Setup

Survey introduction

In our online survey, we first provided a short introduction of the term *familiarity*, which we used to represent the remaining knowledge a developer has, as follows:

Table 4.4: Projects from which we invited participants to our survey.

project	language	developers		
		invited	responded	included
aframe	JavaScript	43	5	4
angular.js	JavaScript	75	8	7
astropy	Python	41	13	7
ember.js	JavaScript	75	5	3
FeatureIDE	Java	10	4	4
ipython	Python	33	3	3
odoo	Python	135	15	10
react	JavaScript	153	4	4
serverless	JavaScript	89	12	11
sympy	Python	68	9	7
overall		722	78	60

> Software familiarity – generally known as a result of study or experience. If familiar, you know: The purpose of a file, its usage across the project, and its structure or programming patterns.

As motivated before, this definition focuses on code knowledge, instead of architectural or meta knowledge. Furthermore, we asked each participant to insert their GitHub username or mail address to avoid multiple responses from the same participant and to allow us to identify their changes. Then, each participant had to state one file in their project they worked with. We asked them not to inspect that file.

Survey questions

We asked a series of questions, of which the following are relevant for this study:

Q_1 *How well do you know the content of the file?*

We first asked our participants to perform a self-assessment of their remaining knowledge with respect to the file they specified, which is a reasonable method considering our previous findings and existing guidelines [Feigenspan et al., 2012; Ko et al., 2015; Siegmund, 2012; Siegmund et al., 2014]. For this assessment, we defined a Likert-scale ranging from 1 (i.e., barely knowing the purpose) to 9 (i.e., knowing purpose, usage, and structure). Note that we assumed no participant would have complete or no knowledge at all, which is why we excluded 0 and 10 from the scale.

Q_2 *After how many days do you only remember the structure and purpose of a file, but have forgotten the details?*

Second, we asked each participant to estimate after how many days they would have a remaining knowledge of 5 on our Likert-scale (i.e., the purpose and use of the file). We used the answers to this question as a sanity check, particularly for **RO-K$_5$**.

Q_3 *How well do you track changes other developers make on your files?*

With this question, we aimed to capture the tracking behavior of our participants (**RO-K$_4$**). In contrast to the values for repetition and ownership, we could not extract this data from version-control histories. Again, we used a Likert-scale with a range from 0 (i.e., do not track at all) to 10 (i.e., follow every change).

Q_4 *How many lines of code does the file contain?*

We used this question to check whether our participants did actually remember the correct file. Consequently, we excluded responses with high error rates (explained shortly).

Q_5 *When was the last date you edited the file?*

Finally, we asked our participants to state when they last edited their specified file. We again excluded responses that indicated that a participant remembered a wrong file or was not motivated.

The two check questions represent rather strict criteria, considering that we found that developers performed worse at recalling this particular knowledge (i.e., M_1 and M_6 in Table 4.2).

Participants

Considered projects

Since we planned to extract version-control data to confirm our check questions (i.e., Q_4, Q_5), we invited 722 developers with public contact data from the ten open-source projects we display in Table 4.4. Aiming to involve different development strategies and users, we selected these projects from those trending on GitHub (at the end of 2016) that exhibited different properties, such as domains, team sizes, and programming languages. Moreover, we considered research projects, for instance, astropy and FeatureIDE, since they usually lead to a higher response rate [Dey, 1997]. At the beginning of 2017, we sent a mail with the survey link to each developer who actively worked on the projects in 2016.

Quality Assurance

Exclusion criteria

We assessed the quality of all responses by employing the following three exclusion criteria:

EC_1 We excluded all response for which the participant specified to have worked on a file, but did actually not commit to it. So, we excluded four responses for which we could not extract the data we needed to address our sub-objectives.

EC_2 We excluded all responses in which the participant specified a file they lastly worked on more than one year ago (before 2016), even though we asked them to consider only that time period. So, we excluded nine responses, since we were not confident that the participant would have actual memory remaining.

EC_3 We excluded every response for which the answers to our two check questions deviated by more than 100 % (75 % for the lower bound with respect to the lines of code) from the actual value. So, we excluded nine questions based on the lines of code (Q_4) and five for the date since the last edit (Q_5).

We can see in Table 4.4 that we excluded 18 responses in total, some fulfilling multiple of our exclusion criteria. In the end, we considered 60 responses as valid, which are a suitable data basis for our analysis.

Data Extraction and Analysis

Extracted version-control data

To address our sub-objectives and perform our quality assurance, we extracted data from each participant's GitHub project. First, we extracted how often a participant committed to their specified file. Note that we considered each day with commits only once, to avoid that a series of small commits on a single day would skew our results. Second, we computed the ratio of code ownership by summing up each line last edited by the participant (identified through `git blame`) and relating the sum to the total file size. So, we also extracted the size of the specified file in its most recent form (when we received the survey response) to analyze our first check question (Q_4). Finally, we extracted the number of days since the last time a participant committed to their specified file, which we needed for our second check question (Q_5).

Table 4.5: Overview of the results for our significance tests for each factor, showing Spearman's ρ, Kendall's τ, and the corresponding significance values.

factor	ρ	**p-value**	τ	**p-value**
file size	0.162	0.218	0.11	0.236
repetition	0.671	<0.001	0.546	<0.001
ownership	0.553	<0.001	0.42	<0.001
tracking	0.036	0.788	0.023	0.81

Statistical tests

For our analysis, we focused on describing and discussing observations we derived from our data. We used statistical tests solely as a supportive means to understand whether our observations are relevant and to determine effect sizes — limiting problems of overly interpreting statistical significance [Amrhein et al., 2019; Baker, 2016; Wasserstein and Lazar, 2016; Wasserstein et al., 2019]. To test our hypotheses with respect to what factors impact developers' memory, we used Spearman's ρ and Kendall's τ, which are both used to assess monotonic rank correlations between two variables without requiring normal distribution. Both tests provide an effect size between -1 and 1, indicating a negative or positive correlation, respectively. We used the R statistics environment [R Core Team, 2018–2020] for all tests (including the corresponding significance tests with a confidence interval of 0.95). For a complete overview, we summarize the results of all tests in Table 4.5.

4.2.2 RO-K$_4$: Impact of Factors on Developers' Memory

Survey data

In the following, we analyze how the three factors we considered impact developers' memory. We display the corresponding data in Table 4.6. For a more concise overview, we group all responses by the remaining memory the participants stated to have. Note that the number of participants specifies how often the combination of memory and commits occurred. As a consequence, the time that passed since the last commit represents the average for the participants of each combination. We received four to ten responses for each level of memory, which is not an ideal (i.e., equal) distribution — but still suitable for our analysis.

Results

Memory decay

In Figure 4.9, we show how our participants' remaining memory relates to the days since their last commit. Every circle represents one participant (cf. Table 4.6), and the sizes represent the number of commits. Moreover, we display two curves: a solid dark blue one

Table 4.6: Overview of our participants' self-assessed memory (m), the number of their commits (#c), and the average time in days since their last commit (Δd). Note that the number of participants (#p) describes how often the combination of memory and commits occurred, which is why we averaged the time.

m	1	2			3				4						
#c	1	1	2	4	1	2	3	6	1	2	7				
#p	4	8	1	1	6	2	1	1	3	1	1				
Δd	206.3	146.6	317	86	70.8	169	60	184	52	159	41				
m	5			6					7						
#c	1	2	9	1	2	4	5	21	1	3	4	8	27		
#p	2	1	1	1	1	1	1	1	1	2	1	1	1		
Δd	58	114	25	25	23	28	23	44	10	183	43	91	100		
m	8									9					
#c	1	2	5	6	7	9	11	15	27	3	4	16	35	37	43
#p	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Δd	9	15	38	96	115	30	299	137	114	41	234	55	43	34	151

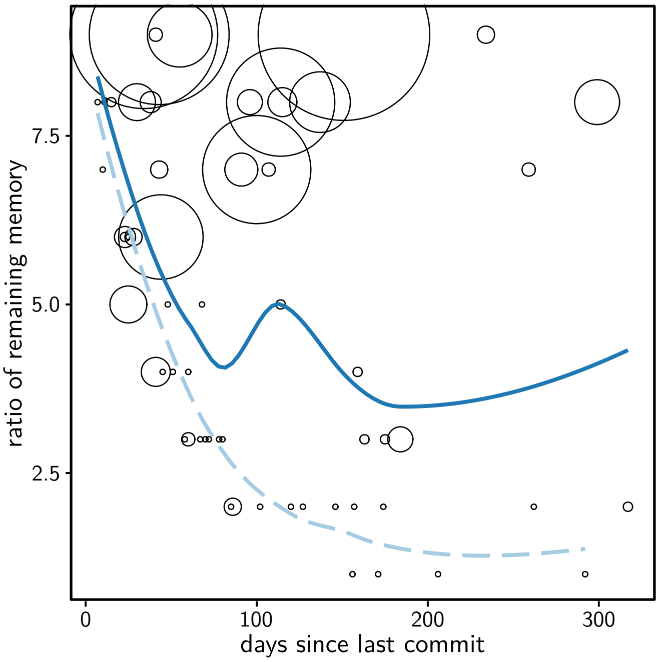


Figure 4.9: Comparing our participants' stated memory to the days since their last commit and the number of commits (represented by the size of each circle). The solid dark blue curve averages all 60 values. The dashed light blue curve shows the average for the 27 responses that involve only a single commit.

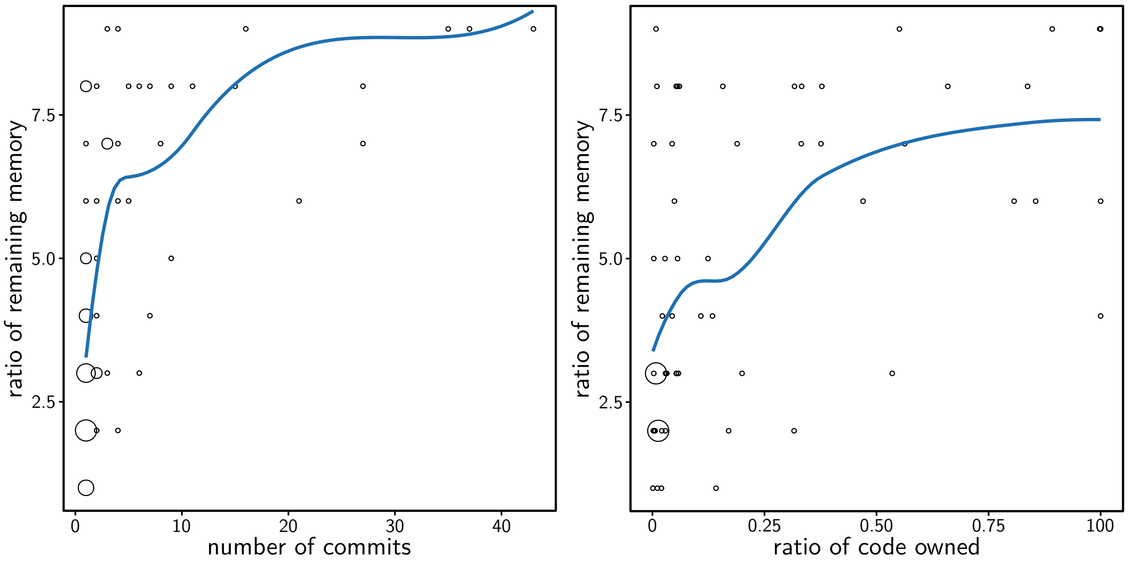
that averages all 60 responses, and a dashed light blue one that averages the 27 responses that involve only one commit. Assuming that all developers have the same memory strength and that no other factors would impact their memory, the first curve should resemble the one of Ebbinghaus. As we can see, the curve first decreases as expected, but after around 100 days it suddenly rises — due to several responses with higher memory levels. This observation could indicate two possibilities. First, Ebbinghaus' forgetting curve may be unsuitable for software engineering. Second, other factors besides the time that passed impact developers' memory. Since the second curve we show excludes response with multiple commits (i.e., repetitions) and fits Ebbinghaus' curve well, we favor the second possibility.

File size

Before we investigated the factors we were actually interested in, we performed an additional sanity check. Namely, we tested whether the file size itself showed any impact on a participant's memory. This should not be the case, since we asked for a Likert-scale assessment representing to what ratio (as defined by Ebbinghaus' forgetting curve) a participant could remember a file. Our statistical tests indicated no significant correlations ($p > 0.2$) and only very weak effect sizes ($\rho = 0.16$ and $\tau = 0.11$). These results confirm our assumption that there is no correlation between file size and the stated memory.

Repetition

We already observed in Figure 4.9 that repetition seems to impact developers' memory, resulting in the rise of the forgetting curve after around 100 days. Moreover, most responses by participants with more than five commits are above a level of 5, while single-commit responses are mostly below that level. In Figure 4.10a, we directly compare repetitions to the participants' remaining memory. We can see that all responses (sizes of circles) below the memory level of 5 involve ten or fewer commits. On average (blue curve), our data shows that a higher number of commits seems to lead to an increased memory level. As we can see in Table 4.5, our statistical tests support this observation, revealing a highly significant ($p < 0.001$) correlation between both factors. Since the tests indicate moderate



(a) Impact of repetition.

(b) Impact of ownership.

Figure 4.10: Impact of repetition (left) and ownership (right) on our participants' memory. The blue curves represent averages, while the size of a circle represents how often that combination occurred.

to strong positive effect sizes ($\rho = 0.67$, $\tau = 0.55$), we reject the null hypothesis that repetition does not impact memory. Instead, we continue in favor of our observation that repetition in terms of commits to a file positively impacts a developers' memory.

Ownership

Second, we investigated whether a participant's memory is impacted by how much code in the specified file they implemented themself. We display the direct comparison of both factors in Figure 4.10b. As we can see, the average (blue curve) behaves similar as for repetition, but on a lower memory level. This observation implies that ownership also positively impacts developers' memory, which is a reasonable assumption considering that they implemented that code (which usually requires mental effort, except for copy-pasted code). The null hypothesis for our statistical tests is that ownership and memory are not correlated. As for repetition, our results (cf. Table 4.5) indicate a highly significant correlation ($p < 0.001$), which is why we reject that hypothesis in favor of our observation. Both tests reveal moderate positive effect sizes ($\rho = 0.55$, $\tau = 0.42$).

Tracking

Finally, we asked our participants to assess their tracking behavior, indicating whether they try to analyze and understand changes others implement in their code. Surprisingly, many of our participants stated a value above 5, and the ratio of responses above and below that threshold is almost equal. We do not visualize the results, since we observed no indication of any relation between tracking and memory. Our tests support this null hypothesis, showing neither a correlation between both factors ($p > 0.78$) nor any meaningful effect sizes ($\rho = 0.04$, $\tau = 0.02$). So, we continue with the assumption that tracking does not impact developer's memory. However, this observation may be caused by how we phrased the question and by what developers consider as tracking.

Discussion

Code memory

Overall, our results indicate that repetitions have the strongest impact on a developer's perceived memory, followed by the ratio of code ownership. This is intuitively reasonable, since both factors indicate that the developer analyzed the code in more detail — and we improve the empirical evidence for this intuition. Interestingly, the number of commits seems to outweigh time to some extent, suggesting that the repeated effort that is spent for implementing and maintaining code refreshes and strengthens a developer's memory. For re-engineering projects, our findings suggest that expertise identification systems that build upon these factors (e.g., by Fritz et al. [2010]) are suitable to find developers that are likely to perform well, for instance, during feature location. However, aligning to our previous findings, we showed that developers cannot remember such details forever, and thus recording the corresponding information (e.g., with feature traces) is key for facilitating any (re-)engineering project.

RO-K$_4$: Impact of Factors on Developers' Memory

By analyzing our survey data, we found indicators that developers' memory is:

- *Moderately to strongly positively correlated to repetitions.*
- *Moderately positively correlated to code ownership.*
- *Not correlated to tracking behavior.*

4.2.3 RO-K$_5$: Developers' Memory Strength

Computing memory strength

To better understand how developers forget the code they worked on, we computed their memory strength (s). For this purpose, we transposed Ebbinghaus' forgetting curve (cf. Equation 4.1) into Equation 4.2:

$$s = -\frac{t}{ln(R)} \tag{4.2}$$

This equation allows us to estimate how fast a developer's memory fades based on the time (t) we extracted from the version-control data and their stated memory (R). We decided to compute three distributions of memory strengths:

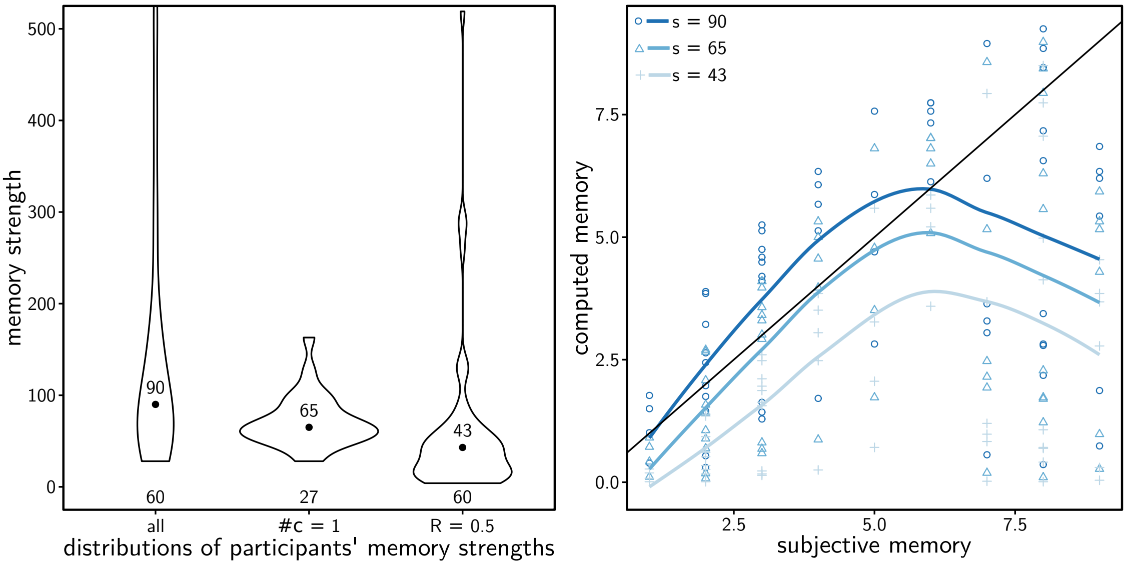
1. We used the memory levels stated by all 60 participants (*all*), which includes the biasing factors we identified. For this reason, we assumed that the median memory strength in this distribution should be higher than it actually is.
2. We used the memory levels stated by the 27 participants that committed once to their file (*#c = 1*). Since this removed the strongest bias we identified (repetition), we assumed that the median memory strength in this distribution should be lower and closer to the real one.
3. We used the responses to our second question, in which the participants should assess after how much time they forgot half of their knowledge (*R = 0.5*). Since this question is challenging to answer, we assumed that the median memory strength could be way off — but be a useful sanity check.

Overall, we assumed that the second distribution would represent the most reliable estimation of our participants' median memory strength.

Results

Distributions of memory strengths

In Figure 4.11a, we display the three distributions as violin plots. We also show the medians (dots) as well as the number of data points (below each violin plot) in each distribution. Note that we used the median in our analysis to better cope with the few large outliers in our data. As we can see, the distributions behave similar to what we assumed. We computed the highest median memory strength (90), and also the largest derivation, when considering the raw responses of all participants. When we considered only single commit



(a) Computed memory strengths (left: all 60 participants; middle: 27 participants with one commit; right: estimation question).

(b) Subjective versus computed memory.

Figure 4.11: Memory strengths (left) and their alignment to the responses (right).

responses, the median was considerably lower (65) and the range of derivation smaller. Interestingly, the median was even lower (43) when we computed memory strengths based on our second question, even though the overall derivation was higher. We remark that our participants' approximations indicated that they forget half of the knowledge about their file after 40 days on average and 30 days in the median.

Discussion

Developers' memory strength

Interpreting our results is quite complex and abstract, since the memory strength is a highly individual factor and is hard to translate directly into practice. However, our results again highlight that different factors impact developers' memory, and we have multiple candidates for employing Ebbinghaus' forgetting curve in software engineering. We argue that the median memory strength of 65 we found in the second distribution is potentially most reasonable. It is based on a smaller sample (resulting in less deviation), but in contrast to the first distribution it is not skewed by repetitions. Moreover, while the median of the third distribution could be a better approximation (covering all participants), we argue that this was a challenging estimate — highlighted by large outliers. However, the average days our participants stated for this estimate is 40, which is close to what the memory strength of 65 would suggest (45 days until half the knowledge is forgotten). So, we argue that 65 is a good estimation for developers' average memory strength. While we need replications, this finding has direct implications for a re-engineering project: If particularly old variants shall be re-engineered, it is more challenging to involve experts. As a result, it may be more suitable to initiate an incremental re-engineering project that starts with integrating and extending more recent variants. To this end, our findings can help to define thresholds for expertise-identification systems that support organization during their decisions.

RO-K$_5$: Developers' Memory Strength

We approximated a median memory strength of 65 with respect to our participants, indicating that they forget half of their knowledge about code within 45 days.

4.2.4 RO-K$_6$: The Forgetting Curve in Software Engineering

Computing memory decay

Finally, we analyzed to what extent we can employ Ebbinghaus' forgetting curve to software engineering. Our previous findings provide supportive evidence that different factors used in expertise-identification systems are reasonable to find experts that are more efficient while re-engineering a variant-rich system. Still, the techniques we are aware of do not involve systematic analyses of developers' memory decay, which could be incorporated by adapting forgetting curves based on empirical findings.

Results

Subjective and computed memory

We used the memory strengths we identified to compute our participants' remaining memory based on Ebbinghaus' forgetting curve. In Figure 4.11b, we compare the stated memory to the computed one for each memory strength. Ideally, one of the resulting plots would align to the black diagonal, which would mean that both values are identical for each participant. As we can see, none of the plots does, which is not surprising, since the forgetting curve does not account for other factors than time. For this reason, all plots start to drop at a subjective memory level of roughly 6, for which we found that repetition can outweigh time. However, particularly for a memory strength of 65, our computed plot is close to the diagonal below the memory level of 6. This underpins that this memory strength seems to be a reasonable approximation for software developers, as long as other factors do not intervene.

Discussion

Measuring forgetting

Overall, the results of our study indicate that the forgetting curve of Ebbinghaus can be useful in software (re-)engineering. However, it does not consider some factors, for instance, repetitions and code ownership, that are highly important in the context of developing variants. For this reason, Ebbinghaus' forgetting curve seems to perform well if developers (or anyone else) would not modify their code later on. To better understand which developers are actual experts of a variant, we must adapt the forgetting curve to consider additional factors. Still, our results show how important it is for an organization to record information, and support its developers in recovering detailed code knowledge — which is forgotten rather fast, and is also expensive to recover. Our findings are rather fundamental and ask for further research to incorporate them into expertise-identification systems that support organizations plan their re-engineering projects; among other activities.

RO-K$_6$: Ebbinghaus' The Forgetting Curve in Software Engineering

Ebbinghaus' forgetting curve is not ideal to measure how developers forget the details of their code, which is impacted by additional factors that are not incorporated.

4.2.5 Threats To Validity

Threats to validity

Due to the intersection of software engineering and psychology, there are several threats to the validity of our study that are hard to overcome. In this section, we detail the most challenging ones we are aware of.

Construct Validity

Terminology of the survey

The phrasing and terminology of our survey questions may have confused participants. This could be particularly problematic, since non-native English speakers may have participated in our survey. We mitigated this threat by using our control questions to exclude responses that indicated misunderstandings (e.g., a participant stating a file they did not commit to in the specified time period).

Internal Validity

Additional factors

The internal validity of our study is threatened by potentially unknown factors. For instance, we did not consider the following factors that may be relevant:

- Performing code reviews and testing a system, while not actually committing changes, arguably impacts a developer's memory.
- Different development processes and strategies (e.g., agile methods) could impact how and what code developers remember.
- The degree of reuse within one or across multiple variants could impact developers' memory, since the same code occurs multiple times.
- According to our findings and the existing literature, the features, effort spent, and importance of information impact how well developers memorize details of their system.

While this threatens our results, we intentionally limited our analysis to a prominent forgetting curve and a few factors that have been shown to be relevant in existing research.

Forgetting curves

We considered only Ebbinghaus' forgetting curve. However, other forgetting curves may be more suitable to understand how developers memorize their system, and involve other relevant factors. We relied on Ebbinghaus' forgetting curve, because it is simplistic and has been replicated in several studies [Averell and Heathcote, 2011; Murre and Dros, 2015]. Since other forgetting curves perform similar, we argue that this is a minor threat to the internal validity of our study. Still, the previous studies have not been performed in the area of software engineering, and thus the findings may not be comparable.

External Validity

Background factors

A developer's background, such as their age, education, gender, motivation, or simply their individual memory strength, may impact their performance [Hars and Ou, 2002; Hertel et al., 2003; Stănciulescu et al., 2015]. However, psychological and medical research suggests that the memory performance remains stable until middle age [Nilsson, 2003] and that gender mainly impacts episodic memory [Herlitz et al., 1997], which is not relevant for our study. Regarding education and motivation, we assume that these factors are comparably homogeneous for our participants. Unfortunately, we could not control for these factors, and thus they remain a threat to the external validity of our study.

Conclusion Validity

Responses

A main threat to our study is that we relied on developers' self-assessments. Similarly, we have a rather small number of responses, which could lead to statistical errors during our analysis. As we described previously, self-assessments seem to be reliable, and we used check questions to improve our confidence in the reliability of the responses, and thus our statistical analysis. We used two different tests, Spearman's ρ and Kendall's τ to perform our hypotheses tests. The former is less strict, allowing us to easier identify potential correlations, while the latter is more strict, improving our confidence that a correlation occurred not only by chance [Fredricks and Nelsen, 2007; Hauke and Kossowski, 2011]. Neither requires a normal distribution or a linear correlation.

Replication

Despite the threats we described, any researcher can replicate our study using our questions, and can compare their findings to our results. Still, depending on the participants and the analyzed factors, the results may vary. This is the case for most empirical studies, and we require replication studies to confirm and extend our findings.

4.3 Information Sources

Studying information sources

In the previous sections, we established that there can be severe knowledge problems when an organization starts a re-engineering project, causing additional costs (cf. Chapter 3). Within this section, we explore what information sources can help to recover relevant information, even though explicit documentation may not exist. For this purpose, we [Krüger et al., 2018b, 2019c] conducted a multi-case study in which we identified as well as located features and their facets in two open-source systems (i.e., Marlin and Bitcoin-wallet). We aimed to understand how to recover feature locations (enriched by other feature facets), since it is one of the most challenging and costly activities in a re-engineering project (cf. Section 3.1.3). Marlin and Bitcoin-wallet exhibit various properties of variant-rich systems, for instance, different reuse strategies and variability mechanisms (i.e., C preprocessor, runtime parameters). In addition, both systems are hosted on GitHub, which provides several additional information sources we can explore, for example, issues and pull requests.

Section contributions

With our study, we aimed to tackle the following three sub-objectives of **RO-K**:

RO-K$_7$ *Explore what information sources support feature location to what extent.*

First, we systematically explored what information sources are available for the two systems, and used them to recover feature locations. We focused particularly on differences between optional and mandatory features: Optional features can often be traced based on the variability mechanism used, but mandatory features have the most potential for reuse and are harder to locate — particularly in cloned variants without any feature traces. The information sources we identified can guide organizations in their re-engineering projects, and researchers in improving feature-location techniques with additional inputs.

RO-K$_8$ *Understand search strategies for recovering feature locations.*

We manually analyzed the communities developing Marlin and Bitcoin-wallet as well as the systems themselves. Despite the different information sources, we found that we relied on similar search strategies to locate features, which we consolidated into patterns. The patterns can help researchers to improve feature-location techniques and guide practitioners in recovering the features of their systems.

RO-K$_9$ *Identify what information sources support specifying feature facets to what extent.*

Finally, we systematically investigated the information sources we identified to recover feature facets. To this end, we read the information recorded in a source and connected it to features and their facets. Our findings help researchers to better understand the properties of variant-rich systems, and support practitioners in understanding from what sources they can recover what information.

All of our results are available in an open-access repository, including feature fact sheets, feature models, and annotated source code.[15] Next, we describe our methodology in Section 4.3.1 and the corresponding threats to validity in Section 4.3.5. In Section 4.3.2, Section 4.3.3, and Section 4.3.4, we report and discuss our findings with respect to each of our sub-objectives, respectively.

4.3.1 Eliciting Data with a Multi-Case Study

Multi-case study design

We employed a multi-case study design [Bass et al., 2018; Leonard-Barton, 1990; Runeson et al., 2012] to address our sub-objectives, relying on two different cases. In the following,

[15] https://bitbucket.org/rhebig/jss2018/

we first provide a brief overview of feature facets before describing the details of our study design. Note that we separately performed a qualitative analysis of information sources in 25 development communities of Unix-like operating systems [Krüger et al., 2020e] and an experiment on the usefulness of source-code comments [Nielebock et al., 2019]. So, in both studies, we investigated information sources and recovery. We use the corresponding insights of both studies to expand our discussions, but do not report their methodologies or results in detail.

Feature Facets

Feature facets

By interviewing industrial practitioners, Berger et al. [2015] derived a list of feature facets that are relevant for describing, managing, and evolving a feature. We analyzed the following nine feature facets in our study:

Architectural responsibility describes a feature's connection to the system architecture, for example, it could be part of the application logic or the user interface.

Binding mode describes whether a feature can be re-bound (dynamic binding at the start of the variant or during runtime) or not (static binding before the variant is deployed).

Binding time describes at what point in time a feature is bound to a variant, for instance, at compile-time, load-time, or runtime (cf. Section 2.3.4).

Definition and approval describes how a feature has been defined and approved, for instance, by using workshops or analyzing related variant-rich systems.

Evolution describes how a feature changed over time, for instance, in terms of revisions from scoping it to rolling it out.

Nature describes whether a feature is a unit of functionality (mandatory) or a unit of variability (optional).

Quality and performance describes any non-functional properties of a feature, allowing developers to test these and check requirements.

Rationale describes the reasons for which a feature has been developed, such as customer requests or market analyses.

Responsbilitiy describes the developers who manage a feature.

We focused on these nine feature facets, because they capture the development and evolution processes of a variant-rich system — which are directly related to our research objectives.

Subject Systems

Marlin

As our first subject system, we selected the 3D-printer firmware Marlin.[16] Marlin is a common subject for researching variant-rich systems [Abal et al., 2018; Stănciulescu et al., 2015; Zhou et al., 2018], since it is a successful open-source system that exhibits three different representations of variability. First, the Marlin platform builds on the C preprocessor to implement variation points and makefiles to include or exclude whole files. Second, Marlin has been forked more than 14,000 times by various users that extend and adapt the codebase to their needs. Third, not all features in Marlin are annotated: some rely on runtime parameters to implement dynamic variability. For our study, we considered the mainline of Marlin (i.e., *Release Candidate 8*) from November 2011 until December 2016.

[16] https://marlinfw.org

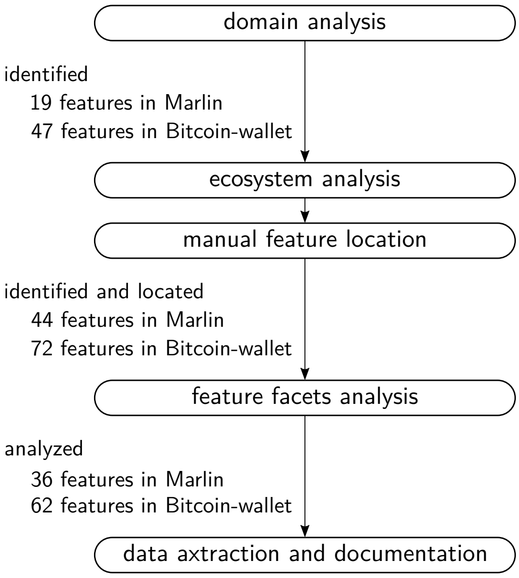


Figure 4.12: Overview of our methodology for recovering feature locations and facets.

Bitcoin-wallet

As our second subject system, we selected the Android app Bitcoin-wallet,[17] which uses runtime parameters to implement variability in its platform. Even though runtime parameters usually allow users to dynamically customize the app while it is running, we found that Bitcoin-wallet uses constants (the developers refer to *compile-time flags*) to control some features at compile-time — representing static variability. Moreover, Bitcoin-wallet has been forked more than 1,600 times. We used version 6.3 (committed on October 1, 2018) and the complete version history back to March 2011 for our study.

Study Design

Methodology overview

We employed the same methodology to analyze each subject system with minor adaptations to account for their different domains: embedded printer firmware and an Android app. In Figure 4.12, we display an overview of our methodology. We also show the number of features we analyzed in each relevant step.

Domain analysis

At first, we analyzed the domains of our two subject systems to identify an initial set of features. A distinct pair of researchers performed the domain analysis for each system. For Marlin, we actually constructed two 3D-printers, a Delta and a Cartesian printer, that use different mechanical parts to move their nozzle. Furthermore, we read manuals, installed the Marlin firmware, and tested configurations — helping us to identify mandatory and optional features. After our domain analysis, we constructed a first feature model with 13 mandatory and six optional features. Regarding Bitcoin-wallet, we installed the app on various devices and emulators to test it. Similar to playing the Apo-Games in our re-engineering cases (cf. Section 3.3), we explored the app's behavior by changing configuration options and observing its user interface. In the end, we agreed on a set of 30 mandatory and 17 optional features, which we organized in a feature model with nine additional abstract features.

Ecosystem analysis

Second, we analyzed how each subject system evolved, including the respective development process and community. The goal of this analysis was to identify additional information sources for feature locations and facets. For Marlin, we identified 18 core developers that lead the development. Moreover, we analyzed release logs, issues, discussions, pull requests, and commits to understand the extensive feature-development process. For Bitcoin-wallet,

[17] https://play.google.com/store/apps/details?id=de.schildbach.wallet

we identified a single core developer, with other developers opening issues, implementing small contributions, and providing feedback. In contrast to Marlin, we could not identify an established development process that incorporates the whole community.

Feature annotations

After improving our understanding of the two subject systems, we analyzed the codebase to annotate feature locations. If the features were not already marked by preprocessor directives (i.e., optional features in Marlin), we added embedded feature annotations [Abukwaik et al., 2018; Ji et al., 2015; Krüger et al., 2019b] for which we had tooling to collect metrics [Andam et al., 2017; Entekhabi et al., 2019]. Namely, we used the following annotations:

`//&begin[<feature name>]` and `//&end[<feature name>]` define that the code encapsulated by them belongs to the feature specified by its name.

`//&line[<feature name>]` defines that one line of code belongs to the specified feature.

By putting these annotations in comments, we avoided interference with the code or the C preprocessor. After this step, we refined our feature models to include newly identified features and their dependencies.

Manual feature location

Due to our previous experiences (cf. Chapter 3) and missing support for the newly identified information sources, we performed our feature location manually instead of relying on existing tools. For Marlin, we relied especially on the release log, which links features to pull requests and commits (representing implicit feature traces), as well as our domain knowledge of constructing the printers. We then performed a manual code review, starting from Marlin's main file by reading comments, G-Code documentation, and the code itself. In the end, we identified 44 features, of which we ignored one that had only some empty methods defined, but no actual implementation. For Bitcoin-wallet, we relied mainly on the Wiki and change log. Unfortunately, these information sources had no links to the source code. Consequently, we performed an extensive code review, starting from the configuration file. In the end, we identified and located 72 feature.

Feature facets analysis

After identifying and locating the features, we investigated their facets by iterating through all available information sources and identifying connected assets based on keywords. For Marlin, we considered 36 features, excluding eight that are (1) repetitions (e.g., unit transformations), (2) interactions and glue code (e.g., movement specifics for different hardware), or (3) small code portions included in other features (e.g., coordinates or radius to measure movements). Besides the sources we used for feature location, we identified additional ones, such as the contributor list and pull-request reviews — leading to a total of ten information sources for recovering feature facets. For Bitcoin-wallet, we used the same methodology for 62 features, excluding ten features that represent interchangeable options (e.g., how Bitcoins are displayed on the user interface). While we could use the same information sources as for Marlin, they had less connection to each other.

Data extraction and documentation

We designed *feature fact sheets* to record the following information (if available):

- A name for the feature.
- The feature's name in preprocessor directives and our annotations.
- A description of the feature's intent.
- The information sources used to identify and locate the feature.
- The search strategies applied for feature location.
- The version in which the feature was released.

- The feature's characteristics (i.e., lines of code, scattering degree, tangling degree).
- The pull request and the sources linked within it.
- The identified feature facets.
- The values of each facet.
- The information source used to recover each facet.

All of our feature fact sheets and all other data we collected are part of our repository.

Example

To provide a better intuition about our analysis methodology, we exemplify it for the Marlin feature `Homing`. This feature puts the extruder of a printer in a stop position while nothing is printed. We first identified this feature while constructing and testing the printers, and located its source code based on G-Codes (which control the mechanical parts), comments, and keywords (i.e., *home*) in the source code. Our analysis showed that the feature is mandatory. Moreover, the rationale for `Homing` is derived from the *technical environment*. Through the G-Code documentation, we found that the feature is part of the *application logic* in terms of its architectural responsibility. From the commit messages, we identified that the feature is an essential requirement for any 3D-printer, indicating that the definition and approval stems from a *market analysis*. With respect to the binding time and mode, we recovered from the source code and commits that `Homing` is bound at *implementation time*, but also comprises *dynamic* variability to react to the reason why homing is needed (e.g., using a different position for cleaning). Finally, the commit changes indicated that the features is managed by *platform developers* and has been *rolled out in release 1.1.2*.

Development culture and processes

Note that our analysis allowed us to obtain a detailed understanding of how features and variants in Marlin and Bitcoin-wallet are developed. Most interestingly, the open-source community of Marlin employs a development process that is highly similar to the one we elicited for Axis (cf. Section 3.2.3) — incorporating clone & own and platform engineering. In contrast, Bitcoin-wallet is driven by a single developer, and thus has fewer contributors that participate in any form of a systematic process. These insights improve our confidence that both subject systems can help us obtain complementary results. However, we omit the results of this part of our study, since we incorporate them into our research on processes and practices in Chapter 6.

4.3.2 RO-K$_7$: Information Sources for Feature Location

Information sources

At first, we elicited information sources that could help us locate features. Next, we describe those sources that proved to be most helpful. In Figure 4.13, we display the number of features in Marlin we could locate through each of these information sources.

Results

Feature location in Marlin

For Marlin, we identified and located 43 features (31 optional, 12 mandatory). The most helpful information sources have been the release log (connecting features to pull requests, commits, and code), `#ifdef` annotations, G-Codes, our domain knowledge, and the actual code review. Code reviews, `#ifdef` annotations (for optional features), and domain knowledge are well-known information sources. Similarly to other studies on manual feature location (cf. Section 3.1.3), we found that keywords in the source code can be highly helpful. However, in some cases, we required additional domain knowledge to improve our searches, indicating that syntax-based feature location requires a deeper understanding of the domain to be effective. G-Codes are a domain-specific information source and control the hardware of 3D-printers. Their close connection to the hardware made it easier to

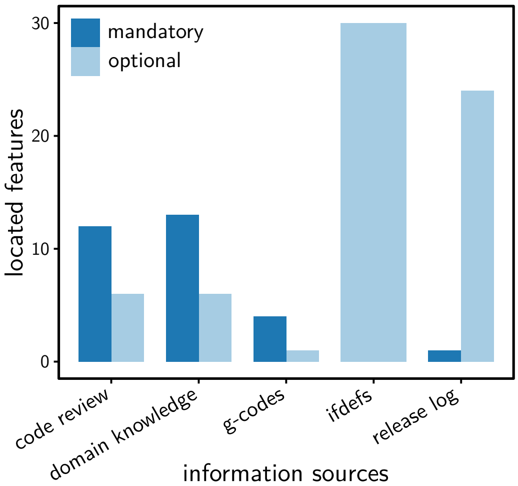


Figure 4.13: Overview of the information sources we used to locate features in Marlin.

understand the behavior of features involving G-Codes, and indicated also mandatory ones. Nonetheless, most interesting for modern software engineering and other systems is the release log and its connected information, which is automatically recorded to some extent.

Feature location in Bitcoin-wallet

Locating features in Bitcoin-wallet was more challenging, since the project and source code comprised fewer information sources. Namely, we could rely only on our domain knowledge and a code review, since we found no links between the code and the release log. We started our code review by identifying keywords in the file `Constants`, which defines the aforementioned compile-time flags. As for Marlin, we used a syntax-based keyword search to locate features. In the end, we identified and located 72 features (60 mandatory, 12 optional).

Other information sources

We identified several additional information sources connected to the software-hosting platform (e.g., issue trackers, Wiki pages, discussion forums) and the domain (e.g., app-store descriptions). Unfortunately, we experienced that these information sources have rarely been useful to locate features. In particular, they provided few to no links to the source code, and thus revealed no new feature locations. Still, these information sources helped to identify (or confirm) features, and recover feature facets.

Discussion

Information sources for feature location

Marlin's developers have adopted the notion of features being optional, which is a widely established notion in platform engineering. This made it easier to locate optional features through the release log (comprising almost only optional features) and `#ifdef` annotations. To identify and locate mandatory features, we had to rely heavily on our domain knowledge and code review. As for our re-engineering cases (cf. Section 3.3), we found that comments in the code were helpful. Even more promising seem domain-specific information sources (e.g., G-Codes), but they require domain knowledge to identify and utilize them. This indicates that feature location (techniques) can be improved by involving different types of documentation as information sources. In summary, we identified five complementary information sources that proved helpful to identify and locate features:

- Domain knowledge (e.g., constructing printers)
- Release log (e.g., pull requests, commits)
- Code review (e.g., comments, dependencies)
- Variability mechanisms (e.g., `#ifdef` annotations)

- Domain-specific sources (e.g., G-Codes)

Unfortunately, Bitcoin-wallet does not have such an extensive set of information sources for feature location. In particular, missing links between the release log and code, the dynamic variability, and a missing notion of features made it harder for us to locate features.

Quality of recovered information

Regarding the usefulness of source-code comments, our own experiment [Nielebock et al., 2019] with 277 developers and related studies (e.g., by Fluri et al. [2007]) indicate an important problem, particularly when considering experienced developers. We found that experienced developers mistrust comments, arguing that these are often poorly maintained and do not reflect the actual source code — which easily becomes a self-fulfilling prophecy. Such findings highlight one important aspect any organization has to consider, and that requires domain knowledge to tackle: Ensuring the quality of the information that is recorded and recovered; especially if it can co-evolve from the code or the actual intentions. Since software-hosting platforms automatically store data on the evolution of a variant-rich system, they can be a helpful means to tackle such quality problems to some extent.

RO-K$_7$: Information Sources for Feature Location

We found that various information sources exist in the context of modern software-engineering practices that can help an organization to recover knowledge for a re-engineering project, and researchers in designing new feature-location techniques.

4.3.3 RO-K$_8$: Search Patterns for Feature Location

Search patterns

We can abstract our search strategies for feature locations into two patterns: we either analyzed the release log or the source code. In this section, we describe both patterns and discuss how to employ them.

Results

Search through release log

Using Marlin's release log for feature location had various advantages. Namely, it lists optional features and links them to other artifacts, such as commits. Our main effort was to browse through such artifacts and use them to locate features in the code. Even though this reduced the need for reviewing and comprehending the source code, we faced new problems:

- The release log, pull requests, and commits involve natural language, which may result in ambiguities or language barriers.
- The release log covered only the latest releases, which is why we could locate older features only through the source code.
- The release log contained mostly optional features (only one feature was mandatory), making this search strategy hardly useful for mandatory features.

These problems are mainly decision dependent (e.g., to only list optional features), and thus can be solved to some extent by employing a different policy. Still, the release log with its linked artifacts was a well-documented, excellent information source for identifying and locating features. Each feature in the release log was linked to six pull requests at most, which tremendously reduced the effort considering that Marlin had 4,000 pull requests. Overall, we inspected 38 pull requests, 100 linked commits, and used the code diffs to locate 24 optional and one mandatory feature.

Search through source code

To tackle the problems of the release log for Marlin, and the fact that no links existed for Bitcoin-wallet, we used systematic code reviews. For Marlin, we needed around 25 hours in total. We started from the file `Marlin_Main.cpp` and found that most of Marlin's mandatory features are, at least partly, located in that file. Using our domain knowledge, we located features and continued with the remaining files, leading to six optional and 12 mandatory features. Note that this involved also new features we did not know before. While reviewing files, we relied heavily on `#ifdef` annotations and G-Codes as prominent seeds for feature locations. After exploiting these two information sources, we systematically reviewed the remaining source code, using keywords, comments, method calls, and dependencies to locate further feature code. For Bitcoin-wallet, we had fewer information sources, and thus relied heavily on the keywords defined in `Constants.java` and `Configuration.java` to locate features. We analyzed our initial feature locations by inspecting variables and the locations these were used in, which required approximately 20 hours of effort. Note that we focused only on the Java code of Bitcoin-wallet. We did not locate source code for six features:

- The feature `BlockExplorer` had three options, namely `Blockchain`, `Blockcypher`, and `Blocktrail`. We found that these options are controlled by a single parameter that is provided by the user interface, which is in XML.
- The feature `Localization` changes the language of Bitcoin-wallet and is defined in a different configuration file, which is automatically processed by Android.
- The features `Cloud Storage`, `Email`, and `Webpage file download` — representing different methods for trading Bitcoins — were apparently not explicitly implemented, but the by-product of other methods.

We experienced that the different design decisions, platforms, and variability mechanisms can facilitate, or hamper, feature location. Importantly, this search strategy is similar to those identified in prior studies (cf. Section 3.1.3), and resembles a combination of information-retrieval as well as exploration-based search patterns [Wang et al., 2011, 2013].

Discussion

Adapting search strategies

Our results indicate that search strategies must be adapted to the different information sources. While this can require some effort, we experienced that additional sources facilitated feature location overall. We also experienced that domain knowledge (e.g., to understand information sources) and code review (e.g., to actually locate features based on commit diffs) are always required. Finally, our results clearly show the importance of establishing traceability in a variant-rich system: The release log of Marlin essentially comprised traces to other artifacts, which facilitated feature location. Still, if both systems also traced features explicitly, we could have saved 45 hours of manual code review to recover feature locations. This clearly shows the benefits of feature traceability for (re-)engineering variant-rich systems.

RO-K$_8$: Search Patterns for Feature Location

We identified two different search patterns in the context of software-hosting platforms: using the release log and using code review. While the former relies heavily on established traces, the latter is always required to some extent.

4.3.4 RO-K$_9$: Information Sources for Feature Facets

Recovered facets

Finally, we report and discuss our results of recovering feature facets. In Table 4.7 we display the feature facets and their values we recovered for Marlin and Bitcoin-wallet. We show an excerpt of which feature facets we recovered from which information source in Figure 4.14.

Table 4.7: Overview of the number of features for which we recovered specific facets.

feature facet	**value**	**#features**	
		Marlin	**Bitcoin-wallet**
architectural responsibility	application logic	28	39
	infrastructure level task	0	3
	user interface	14	19
binding mode	dynamic	3	18
	static	34	43
binding time	compile time	35	2
	configuration time	0	2
	design time	0	39
	implementation	7	0
	link time	3	0
	runtime	0	18
definition and approval	competitors	5	0
	customer requests	25	16
	market analysis	19	9
evolution	rolled out	35	61
nature	configuration/calibration parameter	0	20
	unit of functionality	12	34
	unit of variability	31	7
quality and performance	accessibility/visibility	0	1
	accuracy (or precision)	0	2
	availability	0	1
	clone avoidance	2	0
	code optimization	7	0
	cost	0	2
	performance	0	1
	privacy	0	1
	recoverability	1	0
	reliability	4	1
	resource consumption	5	0
	response time	2	4
	safety	3	0
	security	0	9
	size	0	1
	usability	1	0
rationale	aspects of the technical environment	12	3
	business reasons – customer requests	18	15
	business reasons – market demand	0	6
	social aspects – usage context	7	27
	social aspects – user needs	0	9
responsibility	application developer	7	0
	platform developer	30	60

Results

Values and sources

We can see in Table 4.7 that most facets of both systems comprise multiple values. For instance, the rationale for features in Marlin originates from the users' context, customer requirements, and the technical environment (i.e., hardware). The sole exception is the facet evolution, because we analyzed only the main branches in which all features are rolled out. Furthermore, we can see that we used far more sources to obtain the values for feature facets (cf. Figure 4.14) compared to feature locations (cf. Figure 4.13). This is reasonable, since the values of feature facets are far more diverse and often include information that is

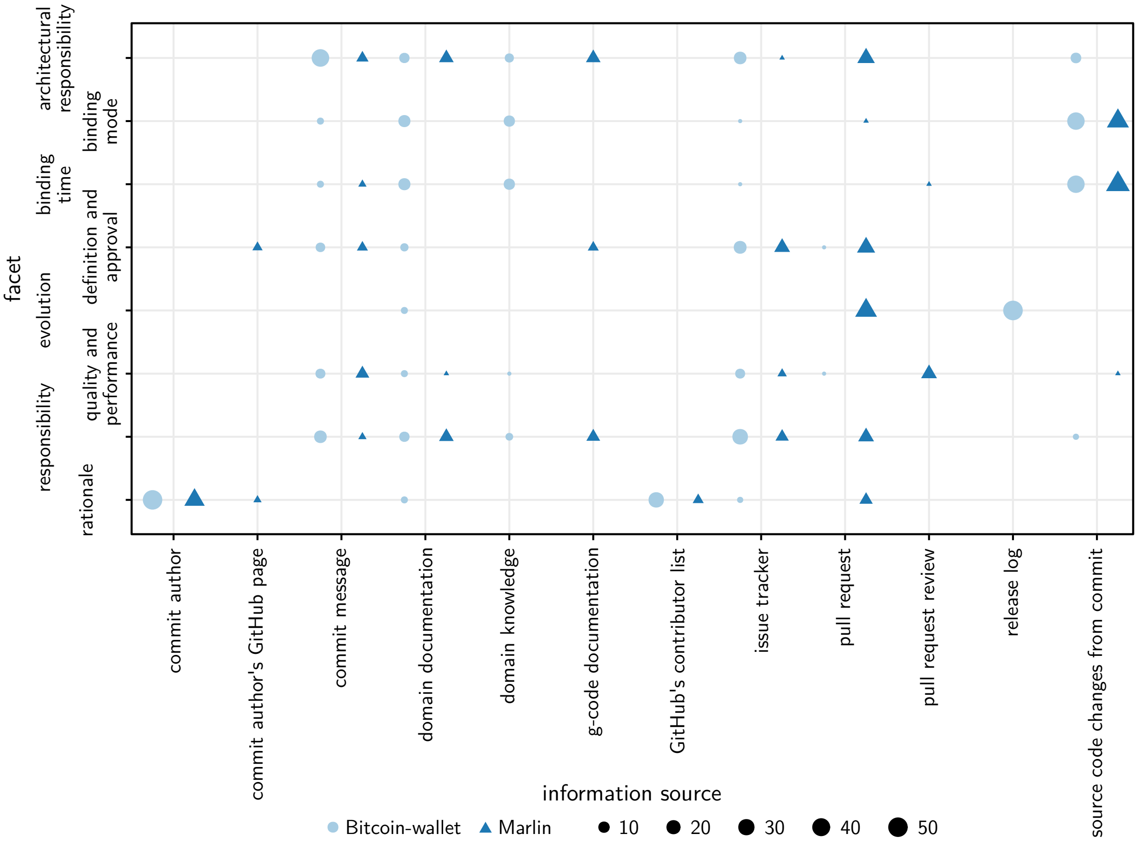


Figure 4.14: Excerpt of the feature facets identified and the information sources used for Marlin and Bitcoin-wallet.

not visible in the source code. Moreover, we can see similarities and differences between the information sources we used for each system. For instance, we used commit messages to recover several facets (e.g., rationale, definition and approval) and source code changes from commits as main source for binding time and binding mode. Other information sources are aligned to specific facets, for example, GitHub's contributor list or the commit author helped only to recover the facet responsibility and the pull requests (Marlin) or the release log (Bitcoin-wallet) helped only with the facet evolution. This is caused by the different development cultures (e.g., linked release log) between both systems, which is why domain documentation (e.g., Wiki pages, readme files) was a richer information source for Bitcoin-wallet. So, information sources for feature facets can vary strongly between systems.

Individual feature facets

In the following, we summarize our core findings for each feature facet:

Architectural Responsibility: For Marlin, we found only two values: application logic or user interface. This seems reasonable, since 3D-printers comprise no other architectural components, such as a database. Interestingly, Bitcoin-wallet exhibits similar results, with only three features belonging to an additional component, the infrastructure. We found no information source that was particularly useful to recover this facet. Consequently, we relied on commit messages, domain documentation, G-Code documentation, issue trackers, pull requests, and sometimes the source code.

Binding Mode and Binding Time: Marlin relies on the C preprocessor, and thus static binding at implementation time (mandatory features) or build time (optional features). Interestingly, we found some features that relied on dynamic variability that could be changed at link time. In contrast, Bitcoin-wallet is highly different, with most features being bound either at design time or dynamically at runtime. We relied mainly on the code itself to recover these two feature facets, since we had to understand how variability mechanisms were implemented (e.g., compile-time flags). For this reason, pull requests, domain documentation, domain knowledge, and commit messages served as supportive means, but were rather limited in their usefulness.

Definition and Approval: For both systems, we found that features mainly originate from customer requests or market analyses (e.g., use cases, hardware) and, in the case of Marlin, are refined during community discussions. We recovered this facet mainly from commit messages and issue trackers (as well as the linked pull requests in Marlin). For some features, we had to dig into additional information sources, namely the commit author's GitHub pages, domain documentation, and G-Code documentation.

Evolution: As we already exemplified, all features we recovered had the value rolled out, and we relied on the release log and pull requests to recover this information. Still, we think it is interesting that we could not recover this information from any other source, except for domain documentation that resembled the release log for Bitcoin-wallet. We remark again that this is arguably caused by our focus on the systems' main branches.

Nature: We extensively discussed the information sources for this facet in Section 4.3.2, which is why we do not display it in Figure 4.14. For Marlin, we found that most features were units of variability, while few provided mandatory functionality needed to use 3D-printers. Interestingly, for Bitcoin-wallet, we found the opposite: far more features were units of functionality and few could actually be disabled. However, in contrast to Marlin, Bitcoin-wallet also comprised several configuration parameters that allow users to customize a feature's behavior at runtime.

Quality and Performance: Marlin employs a rather strict quality assurance with testing branches and code reviews. Moreover, we can see in Table 4.7 that the community values several non-functional properties, such as code optimization, reliability, and safety. In contrast, Bitcoin-wallet has a stronger focus on security, which is reasonable due to its financial domain. Not surprisingly, we could capture non-functional properties only for a subset of all features, utilizing commit messages, domain documentation, issue trackers, pull request reviews, and source code changes. Also, we found that such requirements are rarely made explicit, but often discussed in pull-request reviews.

Rationale: We can see that, for both systems, customer requests are of major interest. However, due their different domains, the second-most features in Marlin are concerned with the technical environment (e.g., hardware), whereas in Bitcoin-wallet the usage context (e.g., reacting to emergency situations) is far more important. We found no centralized information source from which we could recover this facet for either system, such as a requirements database. As a result, we relied on commit messages, domain documentation, G-Code documentation, issue trackers, and pull requests — but none of the information sources stands out.

Responsibility: We distinguished between two roles a developer could have: either a platform developer who works on core features or an application developer who works on their own feature [Ghanam et al., 2012; Holmström Olsson et al., 2012]. Unfortunately, GitHub's contributor list was not particularly helpful, since it only shows involvement — but does not define roles. For this reason, we relied especially on the commit author as information source, and considered also contributors' GitHub pages, issue trackers, as well as pull requests.

In summary, we could recover values for several, but not all, features and their facets from various information sources.

Discussion

Dynamic and static variability

We want to highlight two feature facets that show why it is important to recover knowledge about a system before a re-engineering project, and why to use different information sources for confirmation. Namely, for binding mode and binding time, we found that both systems use a specific variability mechanism that should clearly define these two facets (i.e., C preprocessor for Marlin, runtime parameters for Bitcoin-wallet). However, both systems also implement other variability. For instance, Marlin uses runtime parameters to react to specific use cases while a 3D-printer is running. Similarly, in Bitcoin-wallet the developers use compile-time flags to make runtime parameters constants. So, both systems combine static and dynamic variability, which we did only recover based on our detailed analysis of multiple information sources. Arguably, such hidden and mixed combination of static and dynamic variability exists in other systems, too — and represents an important piece of information to define a platform architecture during a re-engineering project.

Use of information sources

We found that several information sources can be helpful to recover feature facets. Unfortunately, many of those information sources exist only in modern software-hosting platforms, and their usefulness heavily depends on how a community uses them. In this direction, we [Krüger et al., 2020e] analyzed the community websites of 25 Unix-like distributions to understand what information they provide on their development processes. Aligning to our findings in this section, we found that the communities used different strategies to record information. Unfortunately, even those communities that recorded information usually did this in an abstract manner that did not help to understand their actual processes, or scattered the information across various websites. Consequently, if proper documentation is missing, an organization must rely on other information sources to understand the variants it wants to re-engineer. Our results can guide an organization to investigate the right information sources, and help researchers develop new tools.

RO-K$_9$: Information Sources for Feature Facets

We experienced that several information sources are helpful to recover feature facets to a varying extent. Particularly, tracing and recording information on modern software-hosting platforms seems promising to facilitate documenting and recovering knowledge.

4.3.5 Threats to Validity

Threats to validity

Next, we discuss the threats to the validity of our multi-case study. Note that these threats are mainly related to the fact that we performed an extensive, manual exploration — which we employed due to the limited reliability of existing tools, for instance, for feature location (cf. Section 3.3.4) [Ji et al., 2015; Razzaq et al., 2018; Wilde et al., 2003].

Internal Validity

Analysts

The most relevant internal threat is that we recovered features and their facets ourselves, which may lead to different results compared to the original developers. We mitigated this threat by involving two different analysts in each case who became domain experts, for instance, by constructing 3D-printers. Moreover, we analyzed the domains, systems, and communities by reading the available documentation (e.g., for G-Codes) and analyzing meta-data (e.g., in the version-control systems). Finally, we cross-checked the results of each team among the analysts.

External Validity

Subject systems

The external validity of our study is threatened by the two subject systems we considered, which may differ from other systems. While analyzing more subject systems would help tackle this threat, this is not feasible for a manual, exploratory analysis. To mitigate this threat nonetheless, we considered particularly Marlin as a substantial system, with research indicating that such embedded open-source C/C++ systems exhibit similar properties as industrial ones [Hunsen et al., 2016]. Similarly, Bitcoin-wallet should be representative, since Android apps also share common characteristics [Businge et al., 2018].

GitHub

Another threat is the software-hosting platform on which both subject systems are developed: GitHub. Other platforms may use different techniques and tools, for instance, as their version-control system, for integrating cloned variants or for recording information. However, GitHub is one of the largest of such platforms and widely used by open-source, academic, and industrial developers alike. Also, while the type and structure of the available data and information source may vary, their underlying concepts and ideas are usually similar. As a consequence, we argue that our results remain useful for other systems.

Conclusion Validity

Evolution of systems

Other researchers who replicate our study may obtain varying results. In particular, our subject systems continue to evolve, resulting in changed, new, or removed features as well as changes in their facets. Moreover, since features are a highly abstract concept, others may not agree on the same features, and thus their facets. Aiming to mitigate this threat, we provide a detailed overview of our methodology, and include all of our data with the annotated codebases in our repository to allow for comparisons.

4.4 Summary

Chapter summary

In this chapter, we analyzed the knowledge needs of software developers and how they can recover this knowledge. While our findings are applicable to many software-engineering activities, they are particularly interesting when conducting a re-engineering project of a variant-rich system; considering that such projects usually involve several long-living variants. To this end, we first analyzed what knowledge developers consider important and can recall from their memory. Since we found that particularly code details are problematic, we then provided an understanding of how developers forget their source code. Building on these insights, we studied how an organization can recover relevant knowledge for re-engineering a variant-rich system.

Summarizing contributions

Our contributions in this chapter are more fundamental, helping researchers understand how developers' memory works and what they perceive important. The results are helpful to design new tools for measuring expertise, locating features, and recovering feature facets—and our results in Chapter 3 showed that such tools could considerably reduce the costs of re-engineering a variant-rich system. For practitioners, we provided insights into what information is important to record, for instance, for saving tacit knowledge and making it available to novices. Moreover, we showed which information sources developers and organizations can exploit to recover their knowledge of a variant-rich system, particularly to facilitate its re-engineering towards a platform. Abstractly, our results suggest the following core finding:

RO-K: Knowledge

Recording detailed feature knowledge and tracing features to the code is important for (re-)engineering a variant-rich system to avoid expensive knowledge recovery.

Connection to other research objectives

Our findings in this chapter align to our results on the economics of variant-rich systems in Chapter 3 (**RO-E**). Namely, several of the lessons we learned from our five re-engineering cases reappeared, and our experiences confirmed that knowledge has the indicated major impact on economics. Second, since memory decays and we showed the costs of recovering missing knowledge, this underpins the importance of establishing (feature) traceability, which we study in Chapter 5 (**RO-T**). In particular, we found that Marlin's release log, which traces to various assets, was a tremendous help — but since feature traces were mostly missing, we still spent 45 hours on manual code reviews alone. Finally, developers' knowledge is also concerned with development processes and practices, for instance, in the form of meta-knowledge about processes or considering that developers forget while a project progresses. Moreover, we recovered development processes and practices that we incorporate and discuss in Chapter 6 (**RO-P**).

5. Feature Traceability

This chapter builds on publications at ESEC/FSE [Krüger et al., 2019b] FOSD [Krüger et al., 2016b], ICSE [Krüger, 2017; Mukelabai et al., 2018a], ICSME [Fenske et al., 2020], SAC [Krüger, 2018b; Krüger et al., 2018c], SANER [Krüger et al., 2021], SPLC [Ludwig et al., 2019], VaMoS [Krieter et al., 2018a; Ludwig et al., 2020], Empirical Software Engineering [Nielebock et al., 2019], and Software: Practice and Experience [Krüger et al., 2018d].

Chapter structure

While studying the economics (cf. Chapter 3) of re-engineering variant-rich systems and the cost factor knowledge (cf. Chapter 4), we found that establishing feature traceability in source code can tremendously help to comprehend, manage, and evolve a variant-rich system. For instance, our results indicate that feature traces reduce the costs of feature location, and enable developers to recover the knowledge they require to re-engineer or maintain a feature faster. In this chapter, we investigate different techniques for tracing features in source code in more detail (**RO-T**). First, we provide an overview of the related work to connect the results of previous studies with our own experiences, which we detailed in the previous chapters and extend with additional studies (Section 5.1). Second, we report an experiment in which we compared developers' performance with respect to virtual and physical feature traces (Section 5.2). Finally, since our results indicate that virtual traces seem more reasonable, we analyze the differences of using these only for tracing compared to using them also for configuring (Section 5.3). The contributions in this chapter help practitioners to understand the pros and cons of different traceability techniques and their relations to variability mechanisms, which are two closely related concepts in the context of variant-rich systems [Apel et al., 2013a; Vale et al., 2017]. As such, our findings can guide an organization when introducing feature traceability. For researchers, we highlight research opportunities, such as conducting empirical studies to measure the usefulness of feature traces for specific activities or advancing tool support to manage, maintain, and analyze feature traces.

Conceptual framework of traceability

We display a more detailed overview of our conceptual framework regarding feature traceability in Figure 5.1. The *system* within a *project* comprises various assets, most prominently the *source code*. Moreover, the source code implements the system's *features* at certain *locations*. While developing, maintaining, or re-engineering the system, *developers* change the assets, for which they have to locate and comprehend the source code that implements the relevant features. For this purpose, feature *traces* (e.g., annotations) can be used to make feature locations explicit in the source code (or other assets), providing explicit information about

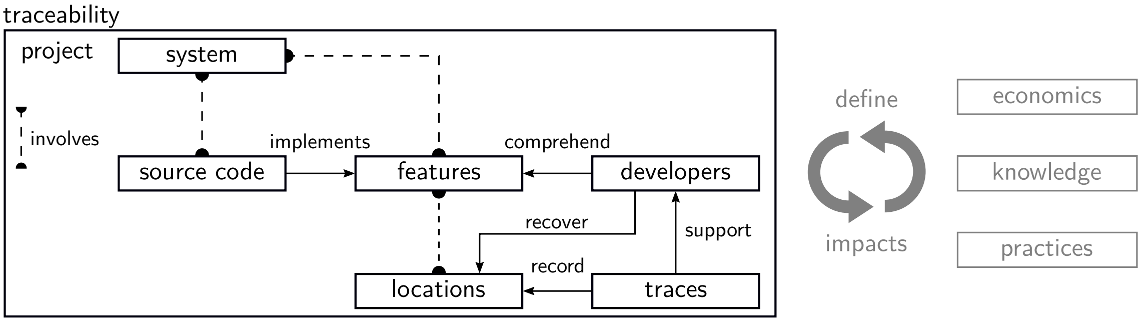


Figure 5.1: Details of the traceability objective in our conceptual framework (cf. Figure 1.1).

the locations. Essentially, feature traceability refers to techniques that provide a mapping between the problem space (e.g., features defined in the feature model) and solution space (e.g., feature locations in the source code) [Apel et al., 2013a] This mapping impacts our other research objectives (e.g., providing knowledge), based on which a suitable feature-traceability technique should be defined (e.g., depending on its support for envisioned practices).

5.1 Feature Traces

Feature traces Traceability in software engineering covers various dimensions (e.g., variants, evolution) and artifacts (e.g., requirements, tests, models) [Charalampidou et al., 2021; Cleland-Huang et al., 2014; Nair et al., 2013; Torkar et al., 2012; Vale et al., 2017]. We focus on feature-traceability in the source code (including variability mechanisms), since features are the primary concern of interest for developing and evolving variant-rich systems [Apel et al., 2013a]. Feature traces help developers identify and locate features (e.g., using automated tool support [Andam et al., 2017; Entekhabi et al., 2019]), reducing the costs of feature location and knowledge recovery. An organization may consider different techniques to establish feature traces [Antoniol et al., 2017], but it is often unclear what technique is favorable for what purpose and situation [Vale et al., 2017].

Section contributions In this section, we discuss the properties, pros, and cons of feature-traceability techniques by synthesizing from multiple studies [Fenske et al., 2020; Krieter et al., 2018a; Krüger, 2018b; Krüger et al., 2016b, 2018c, 2019b; Ludwig et al., 2019; Mukelabai et al., 2018a]. For this purpose, we defined the following three sub-objectives of **RO-T**:

RO-T$_1$ *Compare the dimensions of feature traceability.*

There are various techniques to trace features in a variant-rich system, such as embedded annotations, feature modules, external databases, or variability mechanisms. We build on the dimensions of variability defined by Apel et al. [2013a] (cf. Section 2.3.4) to derive similar dimensions for feature traceability (i.e., technology, representation, and usage). These dimensions help understand and compare the pros and cons of individual feature-traceability techniques, and thus can guide organizations when deciding which to adopt.

RO-T$_2$ *Collect existing empirical evidence on the usefulness of different feature traces.*

Besides the dimensions, we are particularly concerned with the impact of feature traces on program comprehension. For this purpose, we collect existing experiments that investigate the impact of different or “optimized” (e.g., improved discipline of annotations [Liebig et al., 2011]) feature traces on developers. This overview helps practitioners and researchers to understand how different feature traces may impair or benefit developers’ tasks (e.g., feature location).

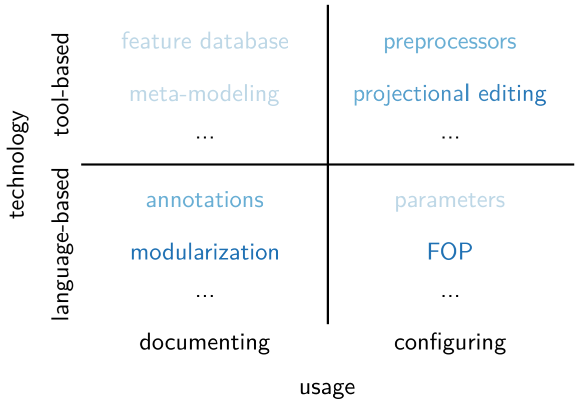


Figure 5.2: Example techniques and their alignment to the dimensions of traceability. The colors highlight the dimension representation: no, virtual, and physical.

RO-T$_3$ *Discuss experiences and open challenges related to feature traces.*

Finally, we connect the dimensions and related work to our results in the previous chapters. Moreover, we extend our discussion based on findings we obtained in further studies with developers and by quantitatively analyzing variant-rich systems. Our results help practitioners understand the perceptions and problems of feature traces in a (re-)engineering project, and motivate our remaining sub-objectives in this chapter.

In Section 5.1.1, we describe the different dimensions of feature traces that impact their use in practice. We continue with an overview of the related work on program comprehension in the context of feature traceability in Section 5.1.2. Within Section 5.1.3, we summarize our additional studies and connect them to our previous findings to motivate and scope our remaining sub-objectives in this chapter.

5.1.1 RO-T$_1$: Dimensions of Feature Traceability

Dimensions of feature traces

Next, we define three dimensions of feature-traceability techniques (i.e., technology, representation, usage) that an organization has to consider when scoping its reuse strategy. For this purpose, we synthesize our insights from developing tools to support the management of variant-rich systems [Krieter et al., 2018a; Krüger et al., 2016b; Ludwig et al., 2020; Mukelabai et al., 2018a]. Moreover, we reflected on existing classifications of variability mechanisms and traceability techniques [Gacek and Anastasopoules, 2001; Kästner and Apel, 2008; Svahnberg et al., 2005; Vale et al., 2017], particularly considering the dimensions defined by Apel et al. [2013a]. In Figure 5.2, we exemplify techniques for feature traceability classified based on the three dimensions. Note that the different levels of each dimension serve only as a basic classification and cannot be clearly distinguished for every traceability technique: some mix or integrate different levels intentionally.

Quality criteria

After introducing the dimensions, we discuss how different feature-traceability techniques can impact three quality criteria [Apel et al., 2013a] during a (re-)engineering project. Precisely, we are concerned with quality criteria that relate to our economics (**RO-E**) and knowledge (**RO-K**) objectives: adoption effort, granularity, and program comprehension. We build upon the dimensions and quality criteria to define the scope of our studies in the remainder of this chapter.

Dimensions

Dimension: technology

The technology dimension adopts the homonymous dimension for variability mechanisms defined by Apel et al. [2013a] as follows:

Language-based techniques rely on the programming language of the variant-rich system to implement feature traces. Consequently, all traces are made explicit in the source code itself, and thus are directly available to the developer. Typical examples are runtime parameters, embedded annotations in comments (cf. Section 4.3.1), and components, which all use only capabilities available in most programming languages. An extension to the programming language is only necessary, if a corresponding variability mechanism is used for tracing, such as feature-oriented programming [Prehofer, 1997].

Tool-based techniques require external tools to implement feature traces. On the one hand, this may involve completely external tools, for example, feature databases. Such techniques are often heavyweight, must be maintained in parallel, and add another level of abstraction that prevents developers from easily locating features — particularly their scattered and tangled (i.e., interacting) code. On the other hand, some techniques rely on tools that are well-integrated into development processes, for instance, the C preprocessor. Moreover, we reproted in Section 4.3 how modern software-hosting platforms allow developers to establish feature traces by linking pull requests to the issue tracker and release log, allowing them to recover the code later [Krüger et al., 2019c].

As we can see, even though we can adopt the variability dimension of Apel et al. [2013a], its levels cover far more techniques that partly exhibit completely different properties.

Dimension: usage

The usage dimension reflects that feature traces may be implemented based on variability mechanisms, distinguishing how the traces can be used:

Documenting techniques aim to only trace features and ideally their locations. Prime examples are feature databases and meta-modeling (e.g., variability models), which both document the features of a system. However, these examples rely on external artifacts, and thus do not necessarily trace locations directly in the source code, for which developers require other techniques, such as embedded annotations. None of these techniques is used to configure a variant-rich system on its own, even though some (e.g., variability models, annotations) may be used by techniques that can.

Configuring techniques are based on variability mechanisms, and their primary goal is to allow developers to configure a variant-rich system. Due to this goal, configuring techniques implement traces in the source code itself — potentially using additional means to manage the traced features (e.g., variability models). For the same reason, most of such techniques (e.g., preprocessors, feature-oriented programming) support only the tracing of optional features.

While we distinguish between these two levels, we can see that their strict separation is challenging for some techniques that can be used for either [Vale et al., 2017]. However, adopting any of such technique has a primary goal that is in either level. For example, feature models may be used for documenting only, or as a supportive means for feature-oriented programming — defining how features of a variant-rich system can be configured.

Dimension: representation

Again, the representation dimension adopts the homonymous dimension defined by Apel et al. [2013a], essentially referring to separation of concerns [Apel and Kästner, 2009b; Kästner and Apel, 2013; Parnas, 1972; Tarr et al., 1999]. However, we introduce a new level and change the naming to reflect on other feature-traceability techniques than variability mechanisms:

No representation means that a technique does not implement explicit feature traces in the source code. For instance, feature databases document the features of a variant-rich system in an external database, and may not trace to the source code or any other assets. Similarly, while runtime parameters are explicit in the source code, they can be used across features and do not mark the start or end of a feature location. Consequently, such techniques do not immediately benefit a developer when inspecting the source code. Instead, developers have to manually inspect the traces and link them to features.

Virtual representation means that features are part of the same codebase and traced by marking their locations. The most widely used techniques for virtual representations build upon annotations (e.g., embedded annotations, preprocessors) to mark the start and end of feature locations. One other solution are background colors to visually highlight the code belonging to a feature [Feigenspan et al., 2011, 2013]. While such feature traces are immediately available to developers, they add further (potentially co-evolving) constructs to the code that must also be maintained.

Physical representation means that features are separated from the codebase. Typical techniques are, for instance, modularization, plug-ins, components, or feature-oriented programming. Ideally, developers can easily locate all code belonging to a feature in its respective files, but scattered and tangled feature code challenge such ideal cases.

Several researchers explored how to combine, or migrate between, virtual and physical representations [Benduhn et al., 2016; Kästner and Apel, 2008; Kästner et al., 2009; Krüger et al., 2016b, 2018d; Ludwig et al., 2020]. Such combinations allow developers to use the representation most suitable for a certain feature, but usually require multiple tools that complicate the development process. To still achieve the benefits of such combinations, researchers recently started to explore the use of projectional editing for engineering variant-rich system [Behringer, 2017; Behringer and Fey, 2016; Behringer et al., 2017; Mukelabai et al., 2018a; Walkingshaw and Ostermann, 2014]. Projectional editing uses an internal structure that allows developers to flexibly switch between different representations.

Quality Criteria

Adoption effort

(Re-)engineering a platform requires planning (cf. Chapter 3), independently of the employed techniques. Considering feature traceability, an organization must assess the costs of implementing feature traces with a specific technique versus the benefits that technique promises. To this end, an organization's core decision is based on its needs in the dimension usage. Namely, the organization must decide whether it wants to implement feature traceability for configuring, documenting, or both. For instance, to enable configuring, an organization has to adopt the corresponding tools and processes. Afterwards, the organization has to select a technology by also considering the dimension representation. For example, simple language-based (e.g., runtime parameters, embedded annotations) or tool-based (e.g., preprocessors) techniques for virtual representations are well-known and require little additional effort. In contrast, more advanced techniques (e.g., feature-oriented programming) may be more helpful to structure large features (e.g., physical representation), but require additional training and adapted processes.

Granularity

Granularity defines on what level of detail features can be traced or configured, and represents the connection between adoption effort and program comprehension. Some techniques (e.g., preprocessors) allow a fine granularity (e.g., down to single characters), while others are far more coarse grained (e.g., modularization into components). For an organization, it is important to understand on what level of granularity its feature traceability shall be implemented, defining particularly the dimension representation. As a concrete example, techniques for virtual representations (e.g., annotations) allow fine-grained traces on statement level or below (we discuss the problems of fine-grained annotations shortly). In contrast, techniques for physical representations could be used on statement level (e.g., a class with a single line of code), but usually comprise larger code fragments to avoid an explosion in code size (e.g., method calls) and in the number of files. Either decision again depends on the usage of traces, and defines also the techniques that can be used; thus impacting the adoption effort (i.e., adding annotations) and program comprehension (i.e., comprehending fine-grained annotations).

Program comprehension

We previously identified that program comprehension is a key cost factor while (re-)engineering variant-rich systems (e.g., during feature location), which can ideally be facilitated by feature traceability. Since program comprehension is a complex cognitive problem, all three dimensions are of equal importance. For instance, preprocessors are external tools that developers need to understand first, and even though they require little adoption effort, they are often argued to obfuscate the source code with annotations that make it harder to comprehend how the configured code behaves — a situation referred to as "`#ifdef` hell" [Lohmann et al., 2006; Spencer and Collyer, 1992; Tartler et al., 2011]. Similarly, for feature traces based on physical representations, it can become challenging to comprehend interacting feature code that is scattered across different files. Consequently, an organization must define a strategy to properly use and maintain feature traces in a way that supports program comprehension (e.g., considering granularity), whether they are part of the deployment process (i.e., configuring) or not (i.e., documenting). Otherwise, introducing feature traceability will not pay off.

RO-T$_1$: Dimensions of Feature Traceability

We defined three dimensions of feature traceability (i.e., technology, representation, and usage) that help an organization select a suitable technique, and discussed how they impact adoption effort, granularity, and program comprehension.

5.1.2 RO-T$_2$: Empirical Studies on Feature Traces

Related research areas

Features have become a fundamental concept in software engineering, not only for implementing variability in a platform [Apel et al., 2013a; Clements and Northrop, 2001], but to communicate, document, and structure systems [Berger et al., 2015; Krüger et al., 2019c]. Consequently, there is an extensive body of research on automated [Dit et al., 2013; Razzaq et al., 2018; Rubin and Chechik, 2013b] and manual feature location [Krüger et al., 2019a; Wang et al., 2013], as well as on techniques related to feature traceability [Charalampidou et al., 2021; Vale et al., 2017] from different research areas. For instance, requirements traceability is a closely related research area concerned with recovering traces between requirements and the source code [Nair et al., 2013; Torkar et al., 2012]. Empirical studies indicate that such traces facilitate developers' tasks by linking to relevant code locations [Egyed et al., 2010; Jaber et al., 2013; Mäder and Egyed, 2014; Rempel and Mäder, 2017]. Unfortunately, most of these techniques are based on heavyweight external tools (e.g., databases), have no direct representation in the source code, and are concerned with another abstraction than features. Since our research is focused on feature traceability in the source code, we do not include such techniques from related areas.

Focus of this section

In the following, we [Fenske et al., 2020; Krüger et al., 2019b] provide an overview of existing empirical studies that investigated the impact of different feature-traceability techniques. For this purpose, we first define and briefly summarize different types of existing studies. Afterwards, we focus on experiments with human subjects, which aim to understand how feature traces impact developers — and thus are the ones relevant for our research objectives. We provide and overview of such experiments and their properties in Table 5.1, comparing them also to our own experiments that we report in this chapter.

Scope of Existing Studies

Study types

Early research related to feature traceability and variability mechanisms was based on a theoretical point of view and educated assumptions, for instance, to guide the decomposition of assets into modules [Parnas, 1972; Tarr et al., 1999], discuss the pros and cons of the C preprocessor [Favre, 1996, 1997; Spencer and Collyer, 1992], or argue on the benefits of reusing features [Frakes and Terry, 1996; Standish, 1984]. Consequently, such early works rarely involve empirical evidence (and anecdotal evidence at best). In contrast, in this dissertation, we are primarily concerned with establishing such evidence for the re-engineering of variant-rich systems. We can distinguish between four types of studies that aim to improve our understanding of feature traceability and collect confirmatory or refuting evidence (note that we exemplify studies for the C preprocessor, which is analyzed most extensively):

Descriptive studies reason on the impact of a technique through qualitative data based on, for instance, developers' perceptions [Krüger, 2018b; Krüger et al., 2018c; Medeiros et al., 2015], case studies [Spencer and Collyer, 1992], or qualitative code analyses [Abal et al., 2018; Ernst et al., 2002; Fenske et al., 2015; Muniz et al., 2018].

Measurement studies quantify how a technique is used based on software metrics, and use the resulting quantitative data to reason on the technique's impact [Ernst et al., 2002; Fenske et al., 2015; Krüger et al., 2018b; Liebig et al., 2010, 2011; Ludwig et al., 2019; Medeiros et al., 2013; Queiroz et al., 2017].

Correlational studies investigate whether specific software metrics correlate with another property of interest [Fenske et al., 2017b; Hunsen et al., 2016], for example, fault proneness [Ferreira et al., 2016].

Experimental studies manipulate one aspect of how a technique is used and analyze how that manipulation impacts an outcome of interest (cf. Table 5.1).

Not surprisingly, many researchers combine different types of studies to improve the confidence in their findings.

Properties of experiments

Since we found that feature traceability can facilitate (re-)engineering projects by supporting developers, we focus on experimental studies with human subjects that aim to understand how program comprehension is impacted by feature traces in the source code. In Table 5.1, we summarize these experiments, including the number of participants (i.e., novices, professionals), the manipulated dimension or aspect (e.g., representation), and the measurements used. If the authors of an experiment did not report their own classification, we assigned student developers on all levels to the novices and industrial, GitHub, or post-doc developers to the professionals. Moreover, we distinguish between four types of measurements:

- comprehension tasks (C), such as "How many variants of this code are possible?";
- maintenance tasks (M), such as locating a bug or suggesting how to fix it;
- subjective opinions (S), such as "How do you rate the code's readability?"; and
- memory performance (MP), such as measuring the participants' memory decay.

During comprehension and maintenance tasks, experimenters can measure the response time (t), correctness (c), or both (c+t) of their participants.

Related Studies

Focus on representation

Based on our knowledge, we identified ten publications that report 14 experiments. We can see in Table 5.1 that six experiments are concerned with the representation of feature traces.

Table 5.1: Related experiments on feature traceability with human subjects. At the bottom, we compare our experiments in this chapter to the related work.

experiment	**#participants**		**manipulated dimension or aspect**	**measurements**
	nov.	**prof.**		
Le et al. [2011]	25	6	representation (color/annotation)	C_c, C_t, S
Siegmund et al. [2012]	8	0	representation (composition/annotation)	M_{c+t}
Feigenspan et al. [2013]	43	0	representation (color/annotation)	C_{c+t}, M_{c+t}, S
	20	0	representation (color/annotation)	S
	14	0	representation (color/annotation)	C_{c+t}, M_{c+t}, S
Schulze et al. [2013]	19	0	annotation granularity	C_{c+t}, M_{c+t}
Medeiros et al. [2015]	0	202	annotation granularity	S
Melo et al. [2016]	63	6	#features	M_{c+t}
Malaquias et al. [2017]	0	99	annotation granularity	S
	64	0	annotation granularity	M_{c+t}
Medeiros et al. [2018]	0	246	annotation granularity	S
	0	≤28	annotation granularity	S
Muniz et al. [2018]	0	110	bugs	M_c
Rodrigues Santos et al. [2019]	33	0	representation (composition/annotation)	C_{c+t}
Krüger et al. [2019b]	3	46	representation (no/composition/annotation)	C_{c+t}, S
Fenske et al. [2020]	0	521	annotation granularity/complexity	C_c, S
Krüger et al. [2021]	1	18	representation (no/composition/annotation)	MP

C_c, C_t: correctness/time for Comprehension tasks; M_c, M_t: correctness/time for Maintenance tasks; S: Subjective opinion; MP: Memory Performance ; nov.: novices; prof.: professionals

Namely, four controlled experiments [Feigenspan et al., 2013; Le et al., 2011] compare background colors to textual annotations in the context of the C preprocessor. The participants in all experiments preferred background colors and could solve their tasks faster. However, only in one experiment did background colors improve the participants' correctness [Le et al., 2011]. In two other experiments [Rodrigues Santos et al., 2019; Siegmund et al., 2012], the researchers compared virtual (annotations based on the C preprocessor) and physical (composition based on feature-oriented programming) representations. Interestingly, neither experiment revealed any differences between the two representations regarding the participants' performance. Since all of these experiments are concerned with platform engineering, they also focus on configuring and consequent technologies. Thus, it remains unclear what the impact of virtual and physical feature traces compared to no traces is, and whether using them for documenting instead of configuring may be more helpful for developers.

Focus on granularity

Six other experiments are concerned solely with the C-preprocessor and the impact of varying granularity (and consequent complexity) of its annotations [Malaquias et al., 2017; Medeiros et al., 2015, 2018; Schulze et al., 2013]. Unfortunately, four of these experiments are based on subjective opinions only. For instance, Medeiros et al. [2015, 2018] used online questionnaires in which GitHub developers should assess the quality of different code examples, showing that these favored disciplined annotations. Similarly, Malaquias et al. [2017] and Medeiros et al. [2018] refactored annotations in open-source projects towards more disciplined forms and submitted consequent pull requests — of which a majority was accepted. The other two experiments [Malaquias et al., 2017; Schulze et al., 2013] involve students who performed maintenance and program-comprehension tasks on disciplined or undisciplined annotations. Interestingly, only the experiment of Malaquias et al. [2017] showed an improvement with respect to correctness and time. Unfortunately, all of these experiments focus on feature traces used for configuring, and provide rather weak evidence regarding the impact of granularity on adoption effort and program comprehension. More precisely, the evidence is mainly subjective, based on different samples of participants, and involves varying code examples

Other experiments

Finally, two experiments are not concerned with our dimensions of feature traceability. Melo et al. [2016] found that a higher degree of variability (i.e., more features) reduced their participants' speed and accuracy during bug finding. Closely related, Muniz et al. [2018] showed that even professional developers have problems to locate bugs in configurable source code. These experiments are complementary to our research objectives.

RO-T$_2$: Empirical Studies on Feature Traces

Existing experiments are primarily concerned with studying the dimension representation, but focus only on configuring and rely on subsequent technologies. Another set of experiments studies the quality criterion granularity explicitly, while several experiments measure program comprehension (i.e., correctness) and effort (i.e., time).

5.1.3 RO-T$_3$: Experiences and Challenges

Scoping new experiments

Existing experiments focus mainly on the dimension representation, but usually compare only virtual and physical representations based on variability mechanisms. We are not aware of experiments that aim to understand the impact of these representations compared to no traces, or the differences of using traces for configuring and documenting. In the following, we build on our experiences of re-engineering variant-rich systems (cf. Section 3.3) and three additional empirical studies to motivate why we conducted new experiments on those two dimensions. As additional studies, we (1) analyzed StackOverflow posts to understand how developers perceive configurable, physical representations [Krüger, 2018b]; (2) conducted a developer survey to understand how developers perceive physical compared to virtual representations [Krüger et al., 2018c]; and (3) performed a measurement study to elicit how configurable, virtual representations may hide information [Ludwig et al., 2019]. Note that we heavily summarize the setups and results of these studies to provide a concise overview.

StackOverflow Analysis

StackOverflow analysis

To obtain a first impression of how developers perceive configurable, physical representations, we analyzed StackOverflow posts. Community-question-answering systems, such as StackOverflow, connect a large international community that contributes to a specific knowledge base (i.e., software development), and thus can be a helpful means to elicit the perceptions of practitioners and unveil practical problems [Barua et al., 2014; Krüger et al., 2017c]. We searched for all questions that comprised the following string:

```
“aspect-oriented programming”
```

We investigated aspect-oriented programming [Kiczales et al., 1997], since it is arguably the most established variability mechanism for fine-grained physical representations in practice (e.g., in the Spring Framework) [Rashid et al., 2010]. For our analysis, we included from all returned questions (1,734) those with an accepted answer (306) that were concerned with the technique's pros and cons (197). We manually analyzed the text of each selected post (i.e., questions and answers) to understand the subjective perceptions of StackOverflow users.

Developer Survey

Developer survey design

Second, we conducted a developer survey in which we asked developers to assess the impact of modularizing (physical representation) C-preprocessor annotations (virtual representation). For this purpose, we designed an online questionnaire that comprised several code examples,

Table 5.2: Overview of the 19 subject systems we included in our measurement study. The last two columns show how many features with how many annotations that comprised a feature expression (i.e., a preprocessor macro) we analyzed.

system	version	year	domain	since	LOC (C)	#features	#annotations
Apache	8.1	2017	web server	1995	153,357	86	1,000
CPython	3.7.1rc1	2018	program interpreter	1989	426,942	686	4,295
Emacs	26.1	2018	text editor	1985	330,196	680	2,327
GIMP	2.9.8	2018	image editor	1996	761,314	90	1,996
Git	2.19.0	2018	version control system	2005	206,239	65	821
glibc	2.9	2018	programming library	1987	818,176	409	5,217
ImageMagick	7.0.8-12	2018	programming library	1987	342,797	5	993
libxml2	2.7.2	2018	programming library	1999	169,761	117	2,360
Lighttpd	1.4.50	2018	web server	2003	49,693	173	450
Linux kernel	4.10.4	2017	operating system	1991	14,746,931	11,011	36,082
MySQL	8.0.12	2018	database system	1995	153,157	355	4,901
OpenLDAP	2.4.46	2018	network service	1998	287,066	347	1,377
PHP	7.3.0rc2	2018	Program interpreter	1985	894,426	1,162	5,977
PostgreSQL	10.1	2017	Database system	1995	790,282	387	2,585
Sendmail	8.12.11	2018	E-mail server	1983	85,639	24	1,223
Subversion	1.10.2	2018	Version control system	2000	967,225	39	1,008
Sylpheed	3.6.0	2018	E-mail client	2000	117,980	75	417
Vim	8.1	2018	Text editor	2000	343,228	1,378	2,570
Xfig	3.2.7a	2018	Graphics editor	1985	109,341	29	193

an assessment of the participants' background, and five short questions. At the beginning, we asked each participant for what purpose they used the C preprocessor and on what level of granularity. Then, all participants should assess how modularizing features could impact their program comprehension, maintenance efforts, and overall processes. We prepared answering options for most questions based on previous studies, and allowed participants to submit additional comments to most questions.

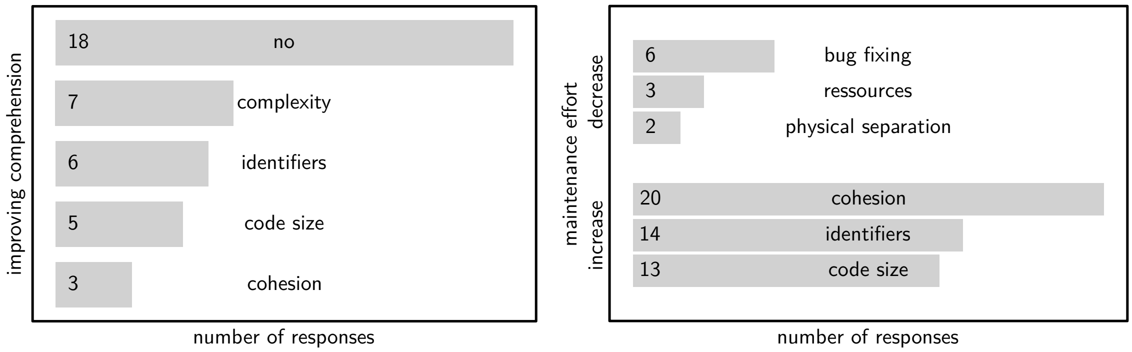
Survey participants

We recruited developers that were experienced with using the C preprocessor in real-world projects. For this purpose, we invited developers from Google newsgroups, XING, and a German software-development mailing list. In the end, we received 35 responses, of which we excluded one, since the participant stated to have no experiences with C or C++. Most participants stated to have an academic degree in computer science, they had been programming for approximately 19.8 years, and had used C or C++ for around 13.9 years. Based on such background information, we argue that our participants were experienced developers who could make educated assessments on different representations of feature traces.

Measurement Study

Analyzed systems

Finally, we performed a measurement study to explore how C-preprocessor annotations may hide features from developers and challenge their program comprehension regarding configurability. We selected a set of 19 open-source systems from related studies [Liebig et al., 2010, 2011; Queiroz et al., 2017], excluding those that were not maintained anymore or that included syntactically malformed test files that we could not parse with our tools [Kuiter et al., 2018a; Ludwig et al., 2020]. We provide an overview of each system's properties in Table 5.2. As we can see, we cover long-living variant-rich systems from various domains and with varying sizes. However, we analyzed only a subset of all existing preprocessor annotations, namely those that are used for external features (e.g., those prefixed with `CONFIG_` in the Linux kernel). Internal features may be used only during development (e.g., for debugging or logging), which is why we did not consider them. While this may exclude relevant annotations, we still found numerous cases in which features were hidden.



(a) Program comprehension.

(b) Maintenance effort.

Figure 5.3: Survey participants' assessment of the impact of physical representation on program comprehension (left) and maintenance effort (reight).

Developers' Perceptions of Representations

Physical representation

While virtual representations of features are widely used in practice for configuring (e.g., the C preprocessor), they are often argued to obfuscate the source code and complicate program comprehension [Apel and Kästner, 2009a; Fenske and Schulze, 2015; Medeiros et al., 2015; Spencer and Collyer, 1992] — even though evidence for those assumptions is scarce [Fenske et al., 2017b, 2020]. Despite such perceived problems, developers rarely employ physical representations to trace and configure features. By analyzing StackOverflow, we found that developers who used aspect-oriented programming experienced:

Pros: The physical representation allows to reuse the same code at various locations (i.e., extending multiple methods with the same aspect) without the need for code clones. For this reason, aspects in particular are perceived as an ideal solution for orthogonal features (e.g., logging, security) that otherwise do not change any behavior.

Cons: Since features are physically separated from the codebase, several developers argue that the anti-pattern *action-at-a-distance* causes problems. More precisely, a virtual representation allows to see how configurations change the code, while the physical representation implements features in other (potentially unknown) locations that are injected at a later binding time (e.g., while compiling). Consequently, developers cannot directly comprehend how the code may behave in every possible configuration. Other developers argue that existing tools are not suitable to manage aspects, or favor to implement physical representations using object-orientation (e.g., modularization with runtime parameters).

These insights shed light into potential problems of physical representations, and align to other experiences reported for aspect-oriented programming in practice [Ali et al., 2010; Colyer and Clement, 2004; Hohenstein and Jäger, 2009; Lesiecki, 2006].

Virtual versus physical

In our second study, we aimed to understand the differences of both representations in more detail. To this end, we asked our participants to assess the impact that a physical representation (as an addition to a virtual one) could have. We summarize their responses regarding program comprehension and effort in Figure 5.3. Note that the numbers can add up to more than 100 %, since we allowed to select multiple answers. Regarding program comprehension, we can see that 18 (52.9 %) of our participants perceived no value in representing features physically. One specified reason aligns to the results of our StackOverflow analysis:

> *"C and C++ already have a clean separation of code at function level, adding new separation layers just makes the code less manageable."*

```
#ifdef A
 // ...
#else
 // ...
#endif
```

(a) #else

```
#ifndef A
 // ...
#else
 // ...
#endif
```

(b) #ifndef

```
#if !defined(A)
 // ...
#elif !defined(B)
 // ...
#endif
```

(c) #if !defined

Figure 5.4: Examples for configurable feature annotations that can hide information.

Other developers considered a physical representation a helpful means depending on the code complexity, number of identifiers defined at other locations, the size of the separated code, and the cohesion of features. Seeing these responses, it is not surprising that only 11 participants thought that their maintenance effort could decrease, for example, by facilitating bug fixing, resource allocation, and physically separating features. Most of our participants stated that the effort would increase, mainly due to the issues that could impair program comprehension. Despite this negative perception, most of our participants saw useful application scenarios for physical representations, particularly for analyzing the source code (18).

Insights on technology

While we were not concerned with the dimension technology in our survey, its close relation to the other dimensions resulted in a few insights based on our participants' explanations. For instance, one participant explained:

> *"It's my opinion that comprehensibility of source code is inversely related to the number of tools required to build it, and to the number of source files which need to be read to understand a particular functional unit. That said, #ifdef-hell kills comprehensibility as well. This is a toss-up for me."*

Clearly, this response indicates that a fine-grained physical representation is not useful, but also that introducing additional technologies to establish feature traceability can be problematic. Such insights support our argument that an organization should carefully define its needs with respect to feature traceability at the beginning of a (re-)engineering project to establish the required technologies.

Virtual Representations and Configuring

Corner cases

The C preprocessor adds complexity to the source code, since it defines locations that are removed for certain feature configurations. During our measurement study, we identified and explored the use of specific constructs of C-preprocessor annotations that can impair feature traceability by hiding information from the developer. In the following, we focus only on such *corner cases*, for which we show examples in Figure 5.4. Essentially, we can distinguish between two types of corner cases:

#else annotations hide information because it is unclear what the #else implies in terms of feature locations. For instance, in Figure 5.4a, it is clear that the first block (between #ifdef A and #else) traces the location of feature A. However, the meaning of the second block after the #else is unclear: Does this code belong to another feature that is not traced explicitly, is it base code that must be replaced if feature A is executed, or is it glue code required if feature A is not selected? Such information is hidden and may exist only as tacit domain knowledge of system experts.

Negating annotations, such as #ifndef A in Figure 5.4c or #if !defined(A) in Figure 5.4b, are the opposite of defining a feature location. They actually refer to the absence of a feature, and thus the code is somehow related to the feature, but does

Table 5.3: Comparison of the identified corner cases (CC) to all configurable annotations we measured in our subject systems.

	feature annotations			**lines of feature code**		
system	**all**	**CC**	**%**	**all**	**CC**	**%**
Apache	278	37	13.31	4,192	327	7.8
CPython	1,320	296	22.42	22,828	2,803	12.28
Emacs	2,701	690	25.55	82,209	14,781	17.98
GIMP	256	52	20.31	4,161	586	14.08
Git	121	34	28.1	1,649	410	24.86
glibc	1,167	366	31.36	13,787	4,381	31.78
ImageMagick	6	4	66.67	63	33	52.38
libxml2	427	75	17.56	4,119	735	17.84
Lighttpd	473	95	20.08	6,650	1,037	15.59
Linux kernel	49,771	11,987	24.08	1,148,508	132,400	11.53
MySQL	1,237	378	30.56	12,698	3,272	25.77
OpenLDAP	1,008	236	23.41	22,667	2,565	11.32
PHP	3,474	589	16.95	131,262	6,137	4.68
PostgreSQL	1,371	420	30.63	25,222	3,767	14.94
Sendmail	49	17	34.69	623	187	30.02
Subversion	381	209	54.86	2,025	811	40.05
Sylpheed	559	61	10.91	14,665	345	2.35
Vim	10,713	1,232	11.5	306,255	14,513	4.74
Xfig	66	10	15.15	704	234	33.24

not actually implement it. Still, many scientific analyses and tools assume that these annotations correctly locate a feature, even though the precise meaning is unclear. For instance, the code could again be base code that must be removed if feature `A` is selected, or imply another feature that is not traced explicitly.

While simple corner cases may not seem problematic, their interactions and tacit implications (e.g., feature dependencies) impair developers' program comprehension. Note that we focus on these corner cases to limit the extent of our analysis, but the C preprocessor allows complex regular expressions in several of its annotations — which impairs feature traceability.

Prevalence of corner cases

In Table 5.3, we display how many of the feature annotations we inspected are related to our corner cases, and how much lines of code these impact. For instance, we analyzed 49,771 configurable feature annotations in the Linux kernel, of which 11,987 (24.08 %) relate to our corner cases — impacting 132,400 lines of code. Without going into details, we found that every system comprises `#else` annotations, with mean and median values of around 17.5 %. Negations are hard to separate, since they can be defined in any C-preprocessor annotation that allows regular expressions (e.g., as in Figure 5.4c). However, `#ifndef` annotations alone occurred frequently (mean of 5.91 %). Overall, we can see in Table 5.3 that corner cases are extensively used by developers to configure features. Unfortunately, these annotations do not allow developers to easily trace and locate features.

Implications for Experiments

Limitations of previous experiments

Our studies indicate that feature traceability is mainly impacted by the dimensions representation and usage, with technology being a consequent decision. Unfortunately, the existing experiments we surveyed on these two dimensions have severe limitations, since they focus on (1) comparing virtual and physical representations to each other, but not to no feature traces; and (2) configurable feature traces and their granularity, only. Moreover, almost all experiments that measure program comprehension or maintenance performance involve only novice participants — which is a valid simplification for many settings, but can still threaten the external validity [Falessi et al., 2017; Runeson, 2003]. By reflecting on our findings and the related work (cf. Section 5.1.2), we derived two open research problems that we explore in the following:

- Understanding the impact of virtually and physically represented feature traces compared to no traces at all (Section 5.2).
- Understanding the impact of using annotations for tracing or configuring (Section 5.3).

Furthermore, we focused on involving experienced practitioners to improve the external validity of our experiments. So, besides extending on previous studies (cf. Section 5.1.2), we also aimed to collect actual evidence for existing hypotheses and our participants' opinions that we described in this section. Such evidence helps organizations select a suitable traceability technique based on reliable empirical data, not only educated guesses.

RO-T$_3$: Experiences and Challenges

Based on our experiences, related experiments, and developers' perceptions, most pros and cons of feature-traceability techniques seem to be related to the dimensions representation (e.g., action-at-a-distance) and usage (e.g., configuring corner cases).

Threats to Validity

Misunderstandings

Regarding our first two studies, we analyzed (e.g., StackOverflow posts, answers to open-ended survey questions) and used (i.e., our survey questions) natural language. This may result in misunderstandings that threaten the construct validity of our results. We mitigated such threats by using check questions in our survey and open-card sorting [Zimmermann, 2016] to agree on the interpretation of our data.

Data elicitation

Our StackOverflow analysis and developer survey are based on subjective opinions and experiences, which do not necessarily reflect on actual facts (i.e., experimentally measured effects may conflict a developer's opinion). We mitigated this threat by involving experienced developers and by using our results mainly to define experiments to elicit actual evidence. Similarly, the internal validity of our measurement study is threatened by our self-developed analysis tools. However, we tested our tools extensively, and a comparison to established tools (i.e., TypeChef [Kästner et al., 2011; Kenner et al., 2010] and SuperC [Gazzillo and Grimm, 2012]) showed that ours perform similar or better [Kuiter et al., 2018a]. Thus, while not fully avoidable, we argue that we mitigated this threat as far as possible.

Generalizability

The major threat regarding the external validity of our studies is the data collection. We used StackOverflow to obtain insights from a diverse sample of software developers, intending to increase the generalizability of our results [Krüger et al., 2017c]. In contrast, our survey involved only 34 C and C++ developers (22 from Germany), whose experiences may not be fully transferable to others. However, both studies indicate similar insights related to physical representations, which increases our confidence in the results. Finally, we analyzed how the C-preprocessor is used in 19 open-source systems, which may not be representative of other variant-rich systems. Since the C preprocessor is widely established in practice and used similarly in open-source and industrial systems [Hunsen et al., 2016], we argue that this threat is limited.

Reliability

The results of our studies depend on the analyzed StackOverflow posts, survey participants, and subject systems — and other researchers may derive different findings, particularly if they vary these samples. However, this is a threat to any empirical study, and we provide all methodological details in the original publications to allow other researchers to replicate each study. Moreover, we publish our tooling for analyzing annotations with extensive documentation in an evaluated open-access repository.[18]

5.2 Virtual and Physical Representations

Experiment on representation

Based on our previous findings, we now investigate how different representations of feature traces impact developers' program comprehension. To this end, we compare virtual and physical representations to no traces. Arguably, either representation of traces should benefit program comprehension, since they guide developers by making feature locations explicit. However, as we showed, existing experiments compare only virtual and physical representations in the context of configurability [Feigenspan et al., 2013; Le et al., 2011; Rodrigues Santos et al., 2019; Siegmund et al., 2012]. With our experiment [Krüger et al., 2019b], we investigated the more fundamental question whether such feature traces are helpful at all for developers — without adding complexity that arises from configuring a variant-rich system. Moreover, we [Krüger et al., 2021] studied the impact of different feature representations on developers' memory, building on our findings in Chapter 4.

Section contributions

In detail, we address the following three sub-objectives of **RO-T** in this section:

RO-T$_4$ *Study the impact of feature representations on program comprehension.*

First, we investigated whether a virtual or physical representation of feature traces impacts our participants' program comprehension. To this end, we compared how effectively (i.e., in terms of correct solutions) and efficiently (i.e., the time needed) our participants could perform six different tasks. The results provide empirical evidence that can help organizations understand how different representations of feature traces can facilitate developers' tasks.

RO-T$_5$ *Investigate developers' perceptions regarding feature representations.*

Similar to other studies, we also elicited our participants' perceptions regarding the different representations to understand whether they confirmed the pros and cons we discussed before. Moreover, we could use the same set of participants to compare our measured quantitative data to subjective qualitative perceptions. So, the results can help organizations and researchers provide tooling that is customized to developers' needs, perceptions, and analysis strategies.

RO-T$_6$ *Analyze the impact of feature representations on developers' memory.*

Finally, we aimed to understand how the different representations of feature traces impact developers' memory. Previously (cf. Chapter 4), we found that developers could apparently recall more abstract concepts (e.g., feature) better than details, which we aim to detail in this study. As a consequence, we connect feature traceability back to developers' knowledge, providing additional insights that can help researchers and organizations while scoping tools, reverse-engineering techniques, and strategies for recording knowledge.

We published our tooling and the anonymous results related to our first two sub-objectives in an evaluated open-access repository.[19] Our artifacts for the third sub-objective are also publicly available.[20] Next, we present the methodology of our experiment in Section 5.2.1

[18] https://bitbucket.org/ldwxlnx/splc2019data/src/master/

[19] https://doi.org/10.5281/zenodo.3264974

[20] https://doi.org/10.5281/zenodo.4417629

Table 5.4: Survey questions we used to measure programming experience in our experiment on the representation of feature traces.

id	question	answers
SQ_1	How do you estimate your programming experience?	1 (very inexperienced) – 10 (very experienced)
SQ_2	How experienced are you with the Java programming language?	1 (very inexperienced) – 10 (very experienced)
SQ_3	For how many years have you been programming?	○ <2; ○ 2-5; ○ 6-10; ○ 11+
SQ_4	For how many years have you been programming for larger software projects (e.g., in companies)?	○ <2; ○ 2-5; ○ 6-10; ○ 11+
SQ_5	What is your highest degree of education that is related to programming?	multiple choice (optional text)

and its consequent threats in Section 5.2.5. In Section 5.2.2, Section 5.2.3, and Section 5.2.4, we report and discuss the results of each sub-objective individually.

5.2.1 Eliciting Data with an Experiment

Online experiment

To investigate our sub-objectives, we designed an online experiment that involved the representation of feature traces as *independent variable* (i.e., no, virtual, physical). Additionally, we controlled for our participants' programming experience, which is why we considered it as an independent instead of a confounding variable [Siegmund and Schumann, 2015]. For **RO-T$_4$**, we measured our participants' correctness and their time in solving six tasks as *dependent variables*. Similarly, for **RO-T$_6$**, we measured our participants' correctness in answering questions about the investigated code after a certain period of time. Considering the results of previous studies, we defined the following *null-hypotheses* that we aimed to refute:

$H_{0\text{-}1}$ Different representations do not impact our participants' correctness.

$H_{0\text{-}2}$ Different representations do not impact our participants' efficiency.

We used hypothesis testing to perform pair-wise comparisons between representations, resulting in 18 tests for each hypothesis (i.e., three groups and six tasks). To correct for multiple hypothesis testing, we used the Holm-Bonferroni method [Holm, 1979]. While we employed different tests, we always relied on their implementation in the R statistics environment [R Core Team, 2018–2020]. Note that we again used hypothesis testing only to support our observations, but are not building on them to avoid misinterpretations [Amrhein et al., 2019; Baker, 2016; Wasserstein and Lazar, 2016; Wasserstein et al., 2019]. In the following, we describe the details of our experimental setup in more detail.

Subject System

Subject system

As subject system, we used a Java platform for content management on mobile devices that was originally implemented with aspect-oriented programming [Young, 2005] and later extended with feature annotations (based on the C preprocessor). This platform has been carefully designed using established coding practices, making it a well-suited subject system for our experiment [Rodrigues Santos et al., 2019; Sethi et al., 2009; Siegmund et al., 2012]. We selected one file from the platform (`MediaControler.java`) that implements ten features for storing and managing media files. To mitigate biases, we deleted all existing comments in the file. Moreover, we reduced the size to around 400 lines of code by removing library imports and a small SMS feature, limiting the effort and time required by our participants.

Representation of traces

Regarding the representation of feature traces, we refactored the file into three versions:

No representation means that we removed all preprocessor annotations to provide purely object-oriented code without any explicit feature traces.

Table 5.5: Experience values of our participants and their participation per iteration (I).

representation	experience values				participants			
	min	**median**	**mean**	**max**	**I_1**	**I_2**	**I_3**	**I_4**
no	6.20	7.60	7.61	9.20	16	7	3	2
virtual	5.60	7.00	6.96	8.80	18	7	6	4
physical	4.00	7.20	6.88	9.20	15	5	2	2
total	4.00	7.40	7.15	9.20	49	19	11	8

Virtual representation means that we replaced existing preprocessor annotations with embedded annotations (i.e., `//&begin[<feature name>]`, `//&end[<feature name>]`).

Physical representation means that we modularized all features into individual classes, also removing existing annotations.

As we can see, we refactored all configurability into pure traceability for each version. So, none of our participants required any additional knowledge about variability mechanisms, and we could focus on the impact of different representations on program comprehension.

Participants

Inviting participants

To improve the external validity of our experiment, we invited 144 software developers (i.e., extended personal networks) from various organizations and countries. After accepting, each participant had to answer the five questions regarding programming experience we show in Table 5.4. We selected these questions from existing guidelines [Siegmund et al., 2014], and derived our answer classifications for SQ_3 and SQ_4 from a StackOverflow user survey in which each class accounted for roughly 25 % of the participants.[21] To align these classes to the answers of SQ_1 and SQ_2, we mapped them to the same scale (i.e., 2, 4, 7, 9). For the educational degree, we considered only whether a participant obtained one (8) or not (3), since we cannot assess which degrees indicate a "better" developer. In the end, we derived an experience value by computing the average of all scales. We considered a developer as an expert if that value was above 5.5 — or as a novice, otherwise. Then, we randomly assigned each participant to one of our three code versions (ensuring equal ratios of novices and experts) and sent out the first iteration of our actual experiment. After two weeks of receiving a participant's answer to any iteration, we sent out the next iteration (explained shortly).

Responses

Not surprisingly, not all developers we invited participated in our experiment. In Table 5.5, we display an overview of our participants' experience values for each representation and their participation in each iteration. For the first iteration, we received 49 responses, mostly from Turkey (20), Germany (13), and the United States (7). We can see that the mean and median values are similar across all groups. Moreover, only three participants working on the physical representation did not achieve an expert rating in terms of experience. Still, at least 12 experts and 15 participants overall worked on each representation during the first iteration. So, while we found differences in the distributions of participants, we argue that these are small. Unfortunately, many participants dropped out during the remaining three iterations of our experiment. While we still received at least two responses for each representation in the last iteration, we have to be cautious with deriving results from such small samples.

Tasks & Questions

Iterations

Our experiment involved four iterations. In the first, we aimed to address **RO-T_4** and **RO-T_5** by showing our participants code on which they solved tasks and answered questions.

[21] https://insights.stackoverflow.com/survey/2016#developer-profile-experience

Then, we conducted three more iterations with at least two weeks pause between them. In these, we aimed to address **RO-T$_6$** by asking our participants questions about the code they worked on, but without showing it.

Tasks For the first iteration of our experiment, we defined six tasks that involve, but are not solely concerned with, feature location to avoid:

- that participants with virtually and physically represented features could quickly search for feature names to locate them and solve their tasks, and
- that our participants' performance would be impacted by learning effects.

So, instead of focusing on the simple task of locating a feature in the presence of feature traces, we researched the impact of such traces on tasks for which developers require actual program comprehension. For this purpose, we defined two sections, each with three tasks. First, we investigated the impact of feature traces on how developers comprehend features and their interactions. Since feature interactions mean that the behavior of different features is tangled, they can easily challenge program comprehension and bug fixing [Apel et al., 2013a]. We defined the following three tasks for this section:

1. From four feature pairs, select the one that involves an interaction in the source code.
2. Select the lines of code in which a specified feature pair interacts.
3. From four statements about a feature interaction, select the correct ones.

Second, we asked our participants to locate three bugs that we inserted into:

4. a feature (does not allow to capture photos);
5. a feature interaction (does increase the counter for videos incorrectly); and
6. the base code (does not allow to delete photos).

Such bugs resemble simple faults (e.g., copy-paste errors) similar to mutations [Jia and Harman, 2011]. We mitigated learning biases by involving different features into each task.

Survey questions At the end of the first iteration, we asked each participant to describe their own perception regarding the feature traces their code comprised. In Table 5.6, we display an overview of the six questions we asked. With EQ$_1$, we aimed to check whether our participants faced any misunderstandings, which could be detailed in EQ$_6$. Moreover, we used the responses to EQ$_5$ to identify whether participants were interrupted during the experiment. With the other three questions, we aimed to elicit qualitative data to address **RO-T$_5$**. Note that we had three different versions for EQ$_4$, depending on the representation a participant was working one.

Memory iterations To investigate developers' memory (**RO-T$_6$**), we sent mails in three more iterations, each two weeks after we received a participant's response to the previous iteration. In each iteration, we asked our participants to answer comprehension questions regarding their code, without showing it. For this purpose, we designed three multiple-choice questions on (available/correct answers in parentheses):

- the details of one feature and its interactions (4/3);
- the types of bugs in the code (5/2); and
- the causes for these bugs, for instance, feature interactions (4/1).

Table 5.6: Survey questions on our participants' perceptions on feature traceability.

id	question	answers
EQ_1	Did you have any problems in answering the survey, e.g., understanding the questions or concepts?	○ yes; ○ no
EQ_2	What was your strategy for comprehending the code in order to do the tasks?	free text
EQ_3	What have been your main problems or challenges during the tasks?	free text
EQ_4	<*no*> Do you think that a different code design concerning the features (e.g., annotating their begin and end, implement them in separate classes) would have facilitated your program comprehension? <*virtual*> Do you think that the annotations provided for each feature helped you understand the code? <*physical*> Do you think that the separation of features into classes helped you understand the code?	free text
EQ_5	Did you face an interruption (more than 5 minutes) for any of the 6 tasks?	checkbox for each task
EQ_6	Do you have any comments on the survey?	free text

A participant could receive 13 points in each iteration, which we reduced whenever they (1) selected a wrong answer or (2) did not select a correct answer. We did not ask these questions in the first iteration of our experiment (but they were related to the tasks) to avoid that our participants could record the correct answers for the remaining iterations.

Implementation

Tool setup

To support the tasks our participants should perform (i.e., marking lines) in an online experiment, we had to implement our own tooling. For this purpose, we implemented a simple client-server architecture that allowed us to display source code in different representations to the participants, who could mark each line by clicking on it. Moreover, we implemented a script that periodically checked whether new responses arrived, and would send out invitations to the last three iterations when enough time passed. We tested our implementation extensively ourselves and let three colleagues review and test the code as well as the tasks (a software engineer, a system administrator, and a PhD student). Note that none of these colleagues participated in our actual experiment.

5.2.2 RO-T_4: Impact of Representation on Program Comprehension

Results for tasks

In Figure 5.5, we display how many participants solved each task correctly — distinguishing between the three different representations of feature traces (i.e., our independent variable). Additionally, we show whether a participant indicated problems in comprehending the experiment (CP) or not (NCP). In Table 5.7, we summarize the corresponding completion times for each task. Note that we considered only participants who stated no interruptions, and removed extremely large outliers (i.e., times that were more than twice above the third quartile for the representation and task). So, we excluded 22 data points, which is the difference between undistrubed and included participants in Table 5.7. Next, we discuss these results based on four observations, which we complement with statistical tests.

Observation 1: Comprehension problems had no impact on the results

Comprehension problems

We can see that the distributions of participants solving a task correctly or incorrectly with and without comprehension problems are similar. Some tasks even show an identical distribution regarding comprehension problems, for instance, the virtual representation of Task 6 led to five correct and four incorrect solutions in either case. Based on this sanity check, we argue that potential comprehension problems do not threaten our results. This assumption is reasonable, since all participants had to comprehend the code from

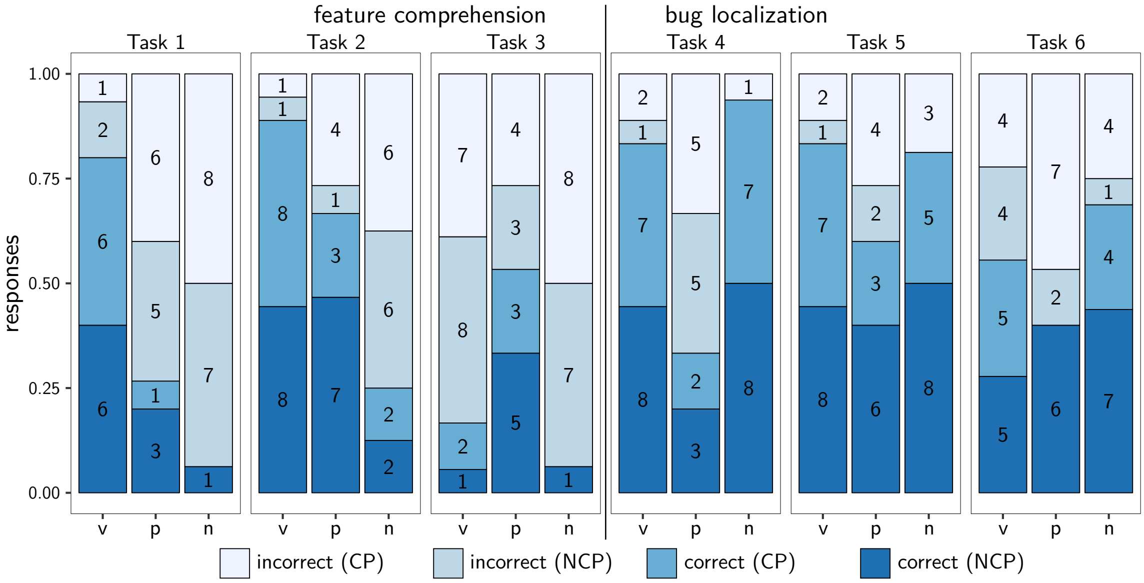


Figure 5.5: Distribution of correct and incorrect solutions for each representation of feature traces (v: virtual; p: physical; n: no) and task. We display whether the corresponding participants indicated comprehension problems or not (CP: Comprehension Problems; NCP: No Comprehension Problems).

scratch, which they are also doing regularly in practice. Moreover, the qualitative responses indicated that some of our participants considered it challenging to comprehend the code itself, not our tasks. Since the code and its included representation of feature traces were our study subject, this improves our confidence in our assumption.

Testing Observation 1

To test our assumption, we hypothesized that the differences between participants who stated comprehension problems and those who did not are not threatening our results. Consequently, we derived the null-hypothesis that correct and incorrect solutions are equally distributed, which we tested with Fisher's exact test [Fisher, 1922]. None of our 18 tests revealed any significant differences, which is why we could not reject our null-hypothesis. This allowed us to continue with the assumption that comprehension problems did not impact our participants' performance. Consequently, we did not consider comprehension problems in our remaining discussion.

Observation 2: Feature representations facilitated feature comprehension

Feature comprehension

We can see in Figure 5.5 that both representations of feature traces led to more correct answers compared to no traces. Particularly, without traces only one participant could solve Task 1 and Task 3 correctly. This finding seems not surprising, since the explicit tracing of feature locations facilitates the comprehension of features by allowing developers to focus on the relevant part of the source code. Interestingly, participants who worked on the physical representation had more problems identifying a feature interaction compared to those working on the virtual representation (Task 1). In contrast, the physical representation led to a higher number of correct solutions when explaining how features interact (Task 3). Potentially, it was easier for our participants to identify feature interactions if the features' code was close to each other — aligning to our previous findings on action-at-a-distance. However, this loss of context may facilitate understanding the interaction based on the data-flow, since method calls indicate whether variables may be modified at a different location. Also, a physical representation allows developers to more easily identify variables that are globally accessible, which may be harder in virtual representations that involve

Table 5.7: Overview of the completion times (in minutes) of our participants.

	Task 1			Task 2			Task 3			Task 4			Task 5			Task 6		
	v	**p**	**n**	**v**	**p**	**n**	**v**	**p**	**n**	**v**	**p**	**n**	**v**	**p**	**n**	**v**	**p**	**n**
part.																		
und.	10	10	9	13	12	15	16	14	15	18	13	16	18	13	15	16	10	16
inc.	10	8	9	12	11	13	14	14	13	16	11	15	16	12	14	15	10	14
times																		
min	2.91	2.23	2.72	0.44	1.14	0.91	0.70	0.67	0.52	0.38	0.66	0.61	1.63	1.47	0.57	0.61	1.30	0.76
mean	13.07	5.51	12.27	1.72	3.26	3.30	2.73	2.26	1.84	1.19	2.40	1.58	3.03	2.90	2.91	3.23	2.59	1.49
med.	11.23	4.03	9.75	1.06	2.63	2.09	2.04	2.11	1.68	1.07	1.79	1.21	2.66	2.54	2.28	3.20	2.50	1.23
max	25.02	12.73	22.92	4.90	8.48	11.96	7.29	4.70	3.90	2.33	6.37	4.09	6.84	5.95	7.55	8.82	5.05	3.48
sd	8.34	3.59	7.54	1.43	2.34	3.14	1.78	1.30	0.89	0.52	2.01	1.00	1.45	1.37	2.01	2.16	1.19	0.75

v: virtual representation – p: physical representation – n: no representation
part.: participants – und.: undisturbed – inc.: included – med.:median – sd: standard deviation

potentially irrelevant context. These findings align well with the related work and our own studies that we discussed in Section 5.1 — thus providing supportive empirical evidence.

Observation 3: Physical representation hampered bug localization

Bug localization

For the last three tasks, we can see that participants who worked on the physical representation consistently identified fewer bugs correctly. Surprisingly, they even performed worse for a faulty set label that was part of a single feature (Task 4). We expected them to perform better for this task, but the incorrect solutions revealed that most participants selected the wrong lines within the right feature. The qualitative responses indicate that this situation may be caused by the fact that the physical units (i.e., classes) represented features instead of logical objects in the sense of object-oriented programming. Arguably, this issue relates to the loss of context (i.e., action-at-a-distance) we discussed for Observation 2. Interestingly, the virtual representation seems to have no considerable impact compared to no traces. However, this may be caused by the simple bugs we used, for which virtual feature representations may not have been useful.

Testing Observations 2 & 3

We tested Observations 2 and 3 on all tasks simultaneously to account for learning effects. To this end, we again used Fisher's exact test, aiming to refute $H_{0\text{-}1}$. Three of our 18 test results were significant, and thus refute our null-hypothesis. Namely, we found significant differences between virtual and no feature representations for Tasks 1 ($p < 0.001$, corrected $p = 0.0028$) and 2 ($p < 0.001$, corrected $p = 0.0029$). These results improve our confidence in our second observation for virtual representations, but not for physical ones. Moreover, the results revealed significant differences between physical and no presentations for Task 4 ($p < 0.001$, corrected $p = 0.0031$), supporting Observation 3.

Observation 4: Feature representations did not impact efficiency

Analysis time

The time our participants took to analyze their code did not differ heavily between the representations. We can see that Task 1 required far more time compared to all other tasks, which is not surprising considering that our participants had to comprehend the code first. Afterwards, they could rely on the knowledge they obtained by inspecting the code, speeding up all other tasks. During the other tasks, all participants took between 1.19 and 3.2 minutes on average with similar minimum and maximum times. As a consequence, the different feature representations seem to have neither a positive nor negative impact on the time developers spend to perform a task. To test for $H_{0\text{-}2}$, we compared the time distributions for each representation using the Kruskal-Wallis test [Kruskal and Wallis, 1952]. None of the tests indicated a significant correlation ($p > 0.3$). So, we cannot reject $H_{0\text{-}2}$, and thus argue that our observation is reasonable.

Table 5.8: Overview of our participants' qualitative comments on feature representations.

response	# mentioned		
	virtual	**physical**	**none**
participants	18	15	16
	analysis strategy		
get big picture of the code	7	6	12
look for keywords	4	2	8
use search function	0	1	3
follow annotations	8	n/a	n/a
follow class names	n/a	7	n/a
	challenges		
code quality	9	6	6
code length	7	0	1
missing IDE	4	4	3
feature location	2	0	3
missing knowledge	1	3	1
	code design		
positive	14	9	n/a
unsure	2	2	n/a
negative	2	3	n/a
virtual representation	n/a	0	4
physical representation	1	n/a	5
explicit representation	n/a	n/a	3

RO-T$_4$: Impact of Representation on Program Comprehension

Our findings provide empirical evidence that virtual representations of feature traces can facilitate developers' program comprehension, while physical representations can hamper bug localization. Moreover, we found no significant impact of feature representations on developers' analysis time.

5.2.3 RO-T$_5$: Developers' Perceptions of Feature Representations

Qualitative responses

We summarize our participants' qualitative responses regarding the different feature representations in Table 5.8. Some numbers do not sum up to the total number of participants, since we allowed them to elaborate and provide multiple insights in their comments. To summarize these comments and understand our participants' opinions on analysis strategies, challenges, and code design, we employed open-card sorting [Zimmermann, 2016].

Analysis Strategies

Analyzing the code

25 of our participants started their analysis with a general code exploration, aiming to comprehend the overall code structure and behavior. However, the details of this analysis varied between participants. Some only skimmed over the code, while others inspected specific code constructs in more detail, for instance, methods and labels. Not surprisingly, 15 of our participants relied on the explicit representations of features traces, if these existed in their code example. Moreover, several participants focused on keywords (14) and used the search functionality of their browser (4), particularly those working on code without any feature representation. These responses align with the general search patterns reported in previous and our own studies (cf. Section 4.3.3) [Krüger et al., 2019a; Wang et al., 2013].

Challenges

Mentioned challenges

Regarding the challenges they faced, 21 of our participants mentioned quality issues of the code. However, most of these concerns were connected to our design decisions for the experiment, such as the code length (8), missing comments, and unsuitable identifier names. We removed comments to avoid biases and tried to reduce the code size, but the examples had to be large enough for feature representations to be useful. Similarly, we aimed to avoid the use of IDEs (11) to avoid biases. Aligning to our previous findings in Chapter 4, five of our participants stated that their missing knowledge challenged their ability to comprehend the code. Still, for this first iteration of our experiment, we intended that every participant had the same level of expertise on the examples. Surprisingly, two participants working on the virtual representation of feature traces had problems in locating features. One of them stated that:

> *"[T]he biggest challenge for me was that all of the features are in a single place, just written one after another."*

This response aligns to the regularly discussed problem of feature annotations cluttering the source code we highlighted in Section 5.1.

Code Design

Virtual representation

Most participants who had examples with explicit representations of feature traces stated a positive perception towards these. Namely, 14 of the 18 participants working with the virtual representation (i.e., feature annotations) reported that these helped them. Some even indicated that the traces were elementary for locating and comprehending features:

> *"Yes, they did. In fact, without the annotations (provided that they are correct), it would have been significantly more difficult to understand which part of the code does what."*

A few participants also criticized this particular representation, arguing that comments imply poor code quality:

> *"[N]o, adding comments in the code is a bad sign, it screams that code is not self explanatory enough."*

Arguably, such responses are connected to clean-code principles [Martin, 2008] and the problem of maintaining and trusting code comments we discussed before [Fluri et al., 2007; Nielebock et al., 2019]. However, considering our quantitative data, we see the mostly positive perception as an indicator that virtual representations, and particularly feature annotations, are a suitable means to support developers.

Physical representation

Similarly, nine of our 15 participants who worked on the physical representation reported a positive perception. Mainly, the participants stated that the modularization allowed them to locate features faster:

> *"It helps [to] logical[ly] aid to decide where to start."*

However, the negative responses also indicate that it was harder for our participants to identify which feature was relevant for a task, due to their missing knowledge:

> *"Yes, I understood the intent [...] with this sorting, naming and separation. It was still unfamiliar and took more time than it would have with familiar code."*

One participant may have summarized the consequent pros and cons best:

> *"On the one hand, it made the classes small and locating possibly relevant code easy. On the other hand, interactions were more difficult to spot because I had to switch between different classes."*

These perceptions align to our previous insights and existing empirical studies, for instance, that action-at-a-distance can be problematic in physical representations. Seeing also our quantitative data, it seems that such representations of feature traces are useful, but must be carefully evaluated to avoid confusion.

No representation

Finally, we asked the participants whose code examples involved no explicit feature representation to reason about adding such traces. As we can see, 11 of our 16 participants considered any form of feature representation helpful:

> *"Features could have been implemented in a more organized way. [W]e clearly need more than one class here."*

In more detail, four of our participants favored a physical representation (i.e., more classes), while five favored a virtual one (i.e., feature annotations):

> *"More comments and better restructuring of the code should be more helpful."*

These perceptions underpin our previous findings, and suggest that implementing an explicit feature presentation in a variant-rich system facilitates developers' tasks.

RO-T$_5$: Developers' Perceptions of Feature Representations

Explicit feature representations extend code analyses, are positively perceived, and seem unproblematic to use. Overall, virtual representations (i.e., feature annotations) are arguably the most helpful in practice, since they are easy to introduce and led to an improvement of our participants' performance.

5.2.4 RO-T$_6$: Feature Representations and Developers' Memory

Memory performance

Finally, we discuss our findings regarding the impact of feature representations on developers' memory. As mentioned, only 18 participants participated in the second iteration, with only eight participating in all iterations. We display their correctness in answering our questions in Figure 5.6. Data points connected through a line represent the responses of the same participant in the corresponding iterations.

No impact

Since we have only a small number of participants in this part of our experiment, we cannot reliably perform statistical tests. However, we do not observe any significant differences between the different representations, essentially indicating null results regarding their impact on developers' memory. In general, all participants performed quite well with most correctness values between 60 % and 80 %, independent of the time that passed. Interestingly, many participants performed better after a slight drop in the second iteration. After investigating this issue, we found that the increased correctness resulted from our participants selecting fewer wrong answers. Next, we discuss two hypotheses that need further experiments to properly refute or confirm them, but that seem reasonable seeing the results of this experiment and our previous findings in Chapter 4.

Hypothesis 1: Developers are good at remembering features

Remembering features

Developers perform better at remembering specific types of knowledge [Kang and Hahn, 2009], and our previous studies suggest that they can recall high-level abstractions of the code more reliable than details [Krüger and Hebig, 2020; Krüger et al., 2018e]. The questions we used for our experiment were concerned with features and bugs, representing more abstract concepts that developers aim to remember according to our previous studies. We can observe the same pattern again in this experiment, adding evidence to our hypothesis that developers can reliably recall what features exist in a variant-rich system. So, the

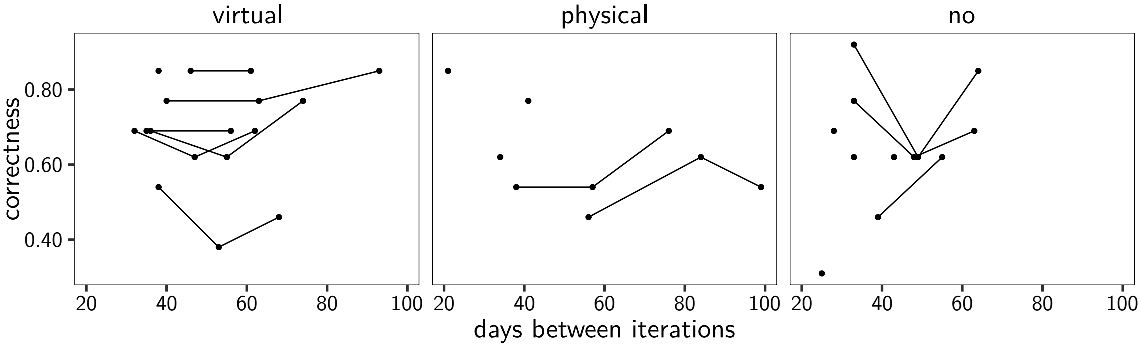


Figure 5.6: Correctness of our participants when answering questions from memory, distinguished by the feature representation.

notion of feature-oriented software development for variant-rich systems [Apel and Kästner, 2009a; Apel et al., 2011; Passos et al., 2013] seems to be suitable for practice. Moreover, for re-engineering a platform from cloned variants, this hypothesis indicates that developers' knowledge is highly valuable to identify features and perform a domain analysis.

Hypothesis 2: Feature representations do not impair developers' memory

Representations and memory

Our results reveal no substantial differences in the correctness of our participants over time. As we can see, the patterns in each group differ slightly, but most participants performed similarly for the second and fourth iteration. We cannot confidently explain the drop during the third iteration without further studies. Closely related to our first hypothesis, we hypothesize that feature representations may not benefit, but do also not impair developers' ability to recall the features of a variant-rich system. As a consequence, feature representations may not facilitate memorizing abstractions of the code, but they can still facilitate the actual program comprehension and allow to automatically analyze the code. In the context of re-engineering a variant-rich system, this mostly relates to the distinction of feature identification (i.e., developers recalling existing features and potential dependencies) and feature location (i.e., actually locating the features in the source code). The former is already important for scoping the platform, while the latter becomes more relevant for the actual re-engineering and all other activities of platform engineering (cf. Chapter 6).

RO-T$_6$: Feature Representations and Developers' Memory

Explicit feature representations seem to neither impair nor benefit developers' memory. So, they are mostly helpful for recovering knowledge during program comprehension.

5.2.5 Threats to Validity

Threats to validity

We aimed to understand how developers perform when facing different representations of feature traces. Since we focused on strengthening the external validity of our experiment [Siegmund et al., 2015], we could not control all confounding factors in program comprehension (e.g., cognitive biases), and used an online setup, we face additional internal threats. In the following, we discuss the different threats to validity of our experiment.

Construct Validity

Validity of responses

Few participants stated that they had problems comprehending our experiment or the concepts of virtual and physical representations. We used check questions and performed sanity checks to account for potential misunderstandings. Namely, we compared the correctness of participants who stated comprehension problems to those who did not and found no significant differences (cf. Section 5.2.2). Moreover, regarding the efficiency of our participants (**RO-T$_4$**), we removed all responses for which they stated that they were interrupted. Unfortunately, we faced a few technical issues for which we lost single data points. Precisely, three participants could not solve the first task, which is why we have only 15 instead of 18 data points for Task 1 for the virtual representation. Also, one participant who worked on physically represented features did solve only the tasks, but did not answer the survey questions (cf. Table 5.6). We decided to keep the response, but included it in the group with comprehension problems for our sanity check. Finally, we removed all unfinished or incomplete responses. These measures cannot fully overcome potential threats to the construct validity, but mitigate their impact. Additionally, most participants stated that their main concerns were related to the actual source code, not the experimental design — which would not threaten the construct validity.

Internal Validity

Code examples To construct our code examples, we modified one file of an existing open-source platform. While open-source and industrial platforms exhibit similar properties [Hunsen et al., 2016], our modifications (e.g., reduced code size, used representations of feature traces, introduced bugs) changed these properties to some extent. Consequently, our results may be biased. While we cannot fully overcome these threats (e.g., another physical representation may have yielded better results), we performed all changes to motivate our participants and mitigate the impact of other confounding factors (e.g., comments).

External factors We implemented our own web-interface to limit our participants' ability to use different tools for analyzing the source code. However, we kept identifier names and syntax highlighting to simulate a development environment, and did not control for tool usage (e.g., searches of web-browsers). We can also not ensure that all participants performed our experiment under the same conditions (e.g., noise level, interruptions, stress). Still, this was a conscious decision, since we aimed to study developers in real-world settings to improve the external validity. Consequently, various external factors may bias our results, but they reflect practice in which developers are influenced by a variety of such factors and can use different tools.

Developers' memory Regarding the last three iterations of our experiment that are concerned with memory, we face the same internal threats on participants' characteristics we discussed for our studies in Chapter 4. In addition, our participants have been aware that we conducted follow-up experiments and may have taken notes. Still, some wrote to us that they did not try to memorize details of the first iteration and could not recall much information, due to the time that passed, their daily work, and a different set of questions — which we used particularly to mitigate this threat. A larger threat is the small number of responses we received, limiting our ability to derive insights. Nonetheless, we obtained interesting results and combine these with our previous findings to mitigate this threat.

External Validity

Participants' background Each software developer has an individual background considering, for instance, their expertise with certain tasks, programming languages, or development processes. To mitigate the threat that we could not generalize our results, we invited experienced software developers from different organizations and countries. Moreover, we controlled for programming experience. Our data indicates that the participants represent a rather homogeneous group, allowing us to compare their responses while mitigating such external threats.

Subject system While our subject system has been used as an example in other empirical studies, it has been developed by researchers. As a result, it may not resemble the properties of real-world variant-rich systems, even though it has been developed for that purpose. In a real-world setting, additional factors (e.g., complexity, size) may impact the pros and cons of feature traces, limiting the generalizability of our results. Nonetheless, since developers' cognitive processes are comparable for the same system (independent variable), our results on feature traces (dependent variable) are valid.

Conclusion Validity

Interpretation of results

We focus on describing the observations of our experiment, limiting the use of statistical tests to avoid their problems [Amrhein et al., 2019; Baker, 2016; Wasserstein and Lazar, 2016; Wasserstein et al., 2019]. To mitigate threats of misinterpreting correlations and observations, we carefully explain and discuss our results. Moreover, we rely on quantitative and qualitative data to improve the confidence in our findings. Finally, we provide all of our artifacts in an open-access repository to allow other researchers to replicate our experiment, which may result in varying results depending on the concrete setup, participants, and code examples, among other factors. We argue that these measures mitigate threats to the conclusion validity of our experiment.

5.3 Traceability and Configurability

Experiment on configurability

Our findings to this point indicate that virtual representations of feature traces are the most useful ones for practice, since they have small adoption costs, facilitate feature comprehension, and do not interfere with developers' memory. However, annotations (as the most common virtual representation) are used not only for tracing, but also for configuring. Other studies indicate that the granularity and complexity of annotations challenge developers' program comprehension and impact their perception [Malaquias et al., 2017; Medeiros et al., 2015, 2018]. Unfortunately, all of these studies involve configurable annotations (i.e., of the C preprocessor) only, making it hard to understand whether the way annotations are used impacts program comprehension differently. In our experiment [Fenske et al., 2020], we analyzed how refactored (i.e., less complex) C-preprocessor annotations impact developers. Moreover, one of the two tasks we defined was concerned explicitly with configuring the code examples, allowing us to study challenges of comprehending the code itself versus configuring the code.

Section contributions

In this section, we investigate the following three sub-objectives to **RO-T**:

RO-T$_7$ *Analyze the impact of annotation complexity on program comprehension.*

We started our experiment by investigating whether the complexity of configurable annotations (i.e., "`#ifdef` hell" [Lohmann et al., 2006; Spencer and Collyer, 1992; Tartler et al., 2011]) impacts' developers program comprehension. For this purpose, we measured how effectively (i.e., in terms of correct solutions) our participants performed two tasks on five code examples. The insights on this sub-objective help us understand how annotations should be implemented to facilitate developers' program comprehension.

RO-T$_8$ *Study developers' perceptions regarding annotation complexity.*

Second, we again elicited our participants' perceptions regarding the different code examples and particularly the complexity of the annotations used. So, we collected qualitative data with the same participants and code examples, which has not been done by previous experiments (cf. Section 5.1.2). Using our results, we can understand how the use of annotations impacts developers' perceptions, helping organizations to define guidelines and motivate the use of annotations.

RO-T$_9$ *Discuss the differences between measurements and perceptions.*

Finally, we discuss differences we found between our qualitative and quantitative data in more detail. Interestingly, our participants' performance contrasts their perception regarding the complexity of the annotations in our examples, also reflecting on the use of annotations for configuring. Based on our analysis, we argue that feature annotations are suitable for tracing features, but the additional complexity of configuring them challenges program comprehension — which helps organizations in scoping how to implement traceability versus variability.

We published all artifacts related to the design of our experiment as well as the anonymized results in an evaluated open-access repository.[22] In the following, we first describe our experimental design in Section 5.3.1 and potential threats to validity in Section 5.3.5. We present and discuss the findings for each of our sub-objectives in Section 5.3.2, Section 5.3.3, and Section 5.3.4, respectively.

5.3.1 Eliciting Data with an Experiment

Online experiment

We designed an online experiment with our code examples (i.e., original or refactored) representing the *independent variable*. As in our previous experiment, we controlled for our participants' experience, but with a retrospective analysis instead of treating it as an independent variable [Siegmund and Schumann, 2015]. This time, we measured only our participants' ability to correctly solve a task as *dependent variable* for **RO-T$_7$**. Moreover, we elicited our participants' perceptions on Likert-scales to consider them as a *dependent variable* for **RO-T$_8$**. We remark that we did not use the response times as a dependent variable (i.e., efficiency), since our objective was only on correctness (seeing that performance was not impacted before) and our setup for this experiment did not allow to reliably measure times. Nonetheless, we removed unreasonably fast responses from our analysis, namely those indicating that a participant only clicked through the experiment. We defined two *null-hypotheses* we aimed to refute:

H$_{0\text{-}3}$ Different annotation complexity does not impact our participants' correctness.

H$_{0\text{-}4}$ Different annotation complexity does not impact our participants' perceptions.

We used statistical tests to reject or confirm these hypotheses, relying on the R statistics environment [R Core Team, 2018–2020]. Again, we use hypothesis testing only to improve the confidence in our observations. Next, we describe the actual setup of our experiment.

Code Examples

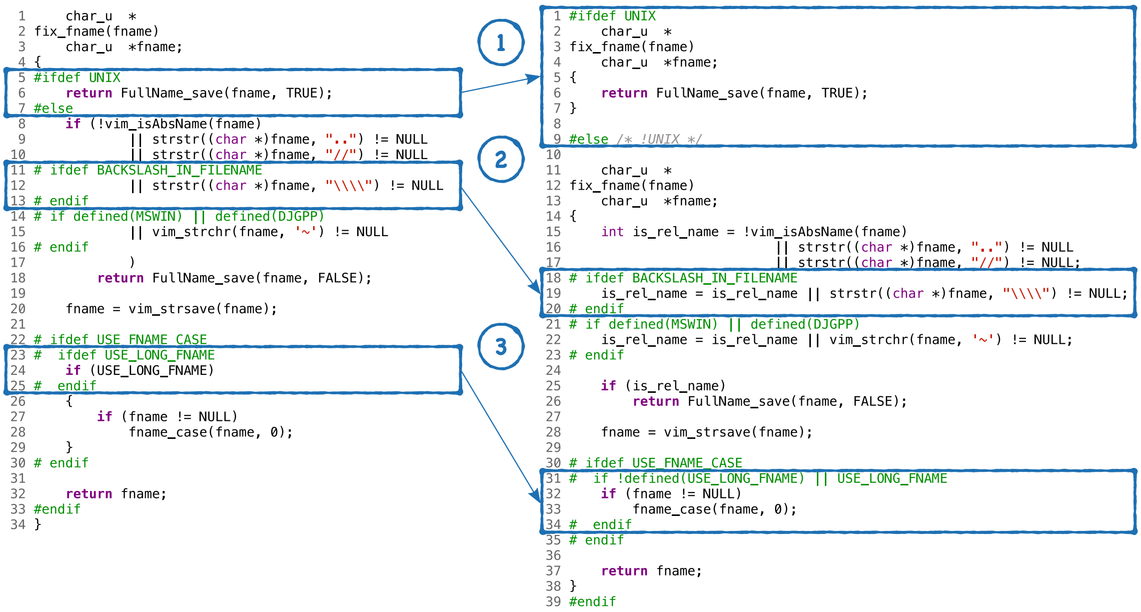
Code examples

During our experiment, we showed each participant five code examples. For each example, the participants had to perform two program-comprehension tasks and rate their perception regarding the use of annotations. We built upon the dataset of Fenske et al. [2015] to select five functions from two real-world text editors, Vim and Emacs. Aiming to achieve a greater impact of our refactorings, we picked examples that are considered particularly smelly — meaning that they should be harder to comprehend according to several metrics. From the dataset, we picked the code examples: `Vim18`, `Vim15`, `Vim13`, `Emacs12`, and `Emacs11`. We refactored each of the examples (except `Vim18`) to address our sub-objectives, which we mark with a corresponding extension in the remainder of this section (e.g., `Vim15_R`).

Refactorings

When refactoring the examples, we aimed to reduce the complexity of the existing annotations (i.e., C-preprocessor directives) to facilitate program comprehension. In parallel,

[22] https://doi.org/10.5281/zenodo.3972411



```
1     char_u  *
2 fix_fname(fname)
3     char_u  *fname;
4 {
5 #ifdef UNIX
6     return FullName_save(fname, TRUE);
7 #else
8     if (!vim_isAbsName(fname)
9             || strstr((char *)fname, "..") != NULL
10            || strstr((char *)fname, "//") != NULL
11 # ifdef BACKSLASH_IN_FILENAME
12            || strstr((char *)fname, "\\\\") != NULL
13 # endif
14 # if defined(MSWIN) || defined(DJGPP)
15            || vim_strchr(fname, '~') != NULL
16 # endif
17            )
18        return FullName_save(fname, FALSE);
19
20     fname = vim_strsave(fname);
21
22 # ifdef USE_FNAME_CASE
23 #  ifdef USE_LONG_FNAME
24     if (USE_LONG_FNAME)
25 #  endif
26     {
27         if (fname != NULL)
28             fname_case(fname, 0);
29     }
30 # endif
31
32     return fname;
33 #endif
34 }
```

```
1 #ifdef UNIX
2     char_u  *
3 fix_fname(fname)
4     char_u  *fname;
5 {
6     return FullName_save(fname, TRUE);
7 }
8
9 #else /* !UNIX */
10
11    char_u  *
12 fix_fname(fname)
13    char_u  *fname;
14 {
15    int is_rel_name = !vim_isAbsName(fname)
16                            || strstr((char *)fname, "..") != NULL
17                            || strstr((char *)fname, "//") != NULL;
18 # ifdef BACKSLASH_IN_FILENAME
19    is_rel_name = is_rel_name || strstr((char *)fname, "\\\\") != NULL;
20 # endif
21 # if defined(MSWIN) || defined(DJGPP)
22    is_rel_name = is_rel_name || vim_strchr(fname, '~') != NULL;
23 # endif
24
25    if (is_rel_name)
26        return FullName_save(fname, FALSE);
27
28    fname = vim_strsave(fname);
29
30 # ifdef USE_FNAME_CASE
31 #  if !defined(USE_LONG_FNAME) || USE_LONG_FNAME
32    if (fname != NULL)
33        fname_case(fname, 0);
34 #  endif
35 # endif
36
37    return fname;
38 }
39 #endif
```

Figure 5.7: Refactorings to reduce the annotation complexity in Vim15, left the original and right the refactored version.

we aimed to preserve the underlying code and its behavior. As a result, the original and refactored code examples had identical functionality, configuration options, and indentation. If the original developers used comments to clarify the feature of an #else or #endif, we added identical comments to our refactored examples. We employed three refactorings for which previous studies indicate that they benefit program comprehension [Malaquias et al., 2017; Medeiros et al., 2015, 2018]. In Figure 5.7, we exemplify all refactorings on a function from Vim, with the circled numbers (i.e., ①, ②, ③) referring to the corresponding refactoring:

R_1 Extract alternative function

In Vim15, Emacs12, and Emacs11, we refactored large blocks of annotated code with alternative implementations (i.e., the #else corner case in Section 5.1.3) into two functions. The survey of Medeiros et al. [2015] indicates that developers prefer such alternative function definitions over complex code constructs. For our example (①), we refactored one function definition for Unix-like operating systems (lines 1–8 on the right) and one for all other operating systems (lines 9–39). While the code increased in size, the functions individually are shorter, more cohesive, and comprise fewer nested annotations. As a result, it should be easier for developers to comprehend the source code and how the configurable annotations impact it.

R_2 Discipline directives

We refactored one and four cases of undisciplined annotations in Vim15 and Vim13, respectively. For this purpose, we followed the advice of Medeiros et al. [2018] to resolve the regularly reported aversion of developers for undisciplined annotation [Malaquias et al., 2017; Medeiros et al., 2015] — which our participants expressed in their comments, too. For our example (②), the original code (on the left) in lines 8–17 is undisciplined, meaning that it is below statement level (i.e., lines 12 and 15 of the if statement are annotated). We refactored the code into a variable (line 15 on the right) that is optionally modified in additional statements (lines 19 and 22). Consequently, the refactored code should be less complex, since features are now traced and configured on statement level.

R_3 Unify compile- and runtime-time variability

In Vim15, we identified a combination of compile-time and runtime-time variability, namely an if statement and an #ifdef annotation that check for the same feature. We refactored the code to comprise only compile-time variability (i.e., #ifdef). For our example (③), the original code can be seen on the left in lines 23–25, and the refactored one on the right in lines 31–34. So, we replaced this mix of feature traces and configurability with a consistent representation, which should facilitate program comprehension.

Considering our participants' comments and previous findings, these refactorings should help developers comprehend the code. Note that we employed all refactorings that were relevant for a code example, which is why we can only judge the overall impact of reducing the annotation complexity, but not the impact of individual refactorings.

Validation of refactorings

We employed all refactorings manually and asked colleagues from other organizations to validate the resulting examples. In the end, we identified a small number of faults that we corrected before conducting our experiment. Still, some participants claimed to have found other faults in our examples. When inspecting these claims, we found that only one was justified: We inadvertently refactored an #if into an #ifdef, thus changing the syntax and semantic of Emacs11_R compared to its original. However, the problem was not this change, but that one of our questions still referred to an #if, potentially leading to confusion. Since the responses to this question were not remarkably different and the difference is small, we argue that the impact of this fault was marginal and does not threaten our findings.

Experimental Design

Versions and questions

In the beginning, we decided to develop two versions of our experiment (E_1 and E_2) that involved different code examples. We started each version with Vim18 to compare the participants of each experiment directly to each other, which allowed us to identify potentially systematic differences between them. Then, we alternated original and refactored examples (cf. Table 5.9) to elicit the data for addressing our sub-objectives, while also avoiding learning biases. Overall, our experiment included 14 questions in three parts that we presented in the same order as in Table 5.9. Note that we presented the experiment in six sections: one for the background and one for each of the five code examples (showing Q_{8-11}).

Background questions

In the first part of our experiment (Q_{1-7}), we elicited background information on our participants, namely their age, gender, roles, and experiences. We designed these questions based on existing guidelines for measuring developer experience [Siegmund et al., 2014]. These questions allowed us to control for confounding factors that may result from our participants' characteristics (e.g., age, experience) in a retrospective analysis [Siegmund and Schumann, 2015].

Questions on code examples

In the second and third part of our experiment (Q_{8-11}), each participant had to solve two program-comprehension tasks and rate their perception of the code. As said, we showed these questions for each code example individually, resulting in five sections during our actual experiment. With Q_8, we asked our participants to select the one correct statement about the code out of a set of options. With Q_9, we asked our participants to define a combination of conditions that would result in a specific line being executed — essentially, they had to configure the code. These two questions allowed us to reason about the impact of using annotations for traceability versus configuring: We expected that Q_9 would be more challenging to answer, since the participants had to understand how each annotation impacted the source code. Note that we did not ask for the same line number, but exact same line of code (thus, we adapted the line number for each example). Afterwards, we asked our participants to asses the use of annotations in each example and how well they

Table 5.9: Overview of the questions and answers in our experiment on annotation complexity. We denote the different versions of our experiment with E_1 and E_2. The questions were the same for each code example, we adapted only the line numbers.

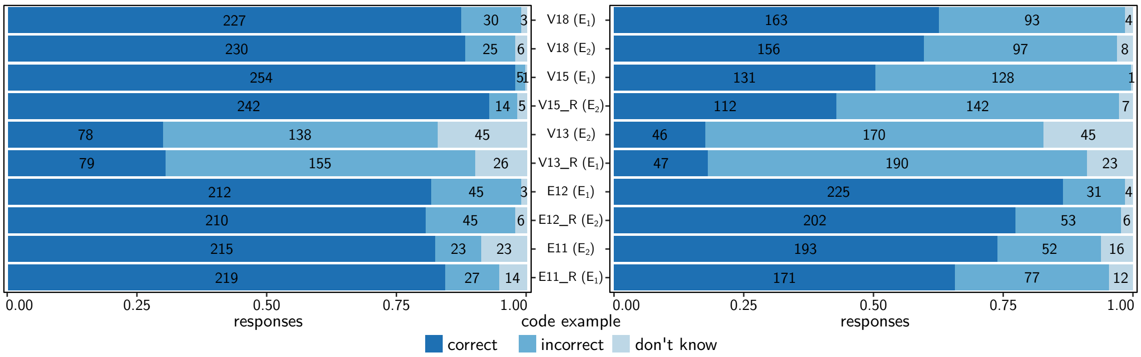
id	**questions & answers**				
background					
Q_1	How old are you?				
	○ 15–19 years . . . ○ 65+ years (5 year periods)				
Q_2	Gender				
	○ Female ○ Male ○ Other ○ Prefer not to tell				
Q_3	How many years of programming experience do you have?				
	Open number				
Q_4	How many years of experience with C/C++ do you have?				
	Open number				
Q_5	Roles in projects				
	Multiple selection (e.g., Developer) and open text				
Q_6	Which open-source projects have you worked on so far?				
	Open text				
Q_7	How would you rank your programming skills in C/C++?				
	○ Beginner ○ Intermediate ○ Advanced ○ Expert				
code examples					
Q_8	Which of the following statements is true?				(Task 1)
					single selection out of
	`Vim18`	E_1	`Vim18`	E_2	5 options
	`Vim15`	E_1	`Vim15_R`	E_2	5 options
	`Vim13_R`	E_1	`Vim13`	E_2	5 options
	`Emacs12`	E_1	`Emacs12_R`	E_2	5 options
	`Emacs11_R`	E_1	`Emacs11`	E_2	6 options
Q_9	When would line <x> be executed?				(Task 2)
					choosing a combination out of
	`Vim18`	E_1	`Vim18`	E_2	9 conditions
	`Vim15`	E_1	`Vim15_R`	E_2	11 conditions
	`Vim13_R`	E_1	`Vim13`	E_2	11 conditions
	`Emacs12`	E_1	`Emacs12_R`	E_2	7 conditions
	`Emacs11_R`	E_1	`Emacs11`	E_2	9 conditions
for each example					
Q_{10}	Do you consider the use of preprocessor annotations in the example appropriate?				
	○ Yes ○ No, because (Open text)				
Q_{11}	Please rate the presented code regarding the following questions:				
$Q_{11\text{-}1}$	How easy was it to understand this code?				
$Q_{11\text{-}2}$	How easy would it be to maintain this code?				
$Q_{11\text{-}3}$	How easy would it be to extend this code?				
$Q_{11\text{-}4}$	How easy would it be to detect bugs in this code?				
	for each a Likert scale: ○ very hard ○ hard ○ easy ○ very easy				

could work on the code (i.e., comprehending, maintaining, extending, bug fixing). To measure these perceptions, we used a simple yes/no assessment (Q_{10}) and four-level Likert scales (Q_{11}). Finally, we aimed to formulate $Q_{8\text{–}11}$ in a way that did not emphasize our research objective to avoid any biases towards or against annotations.

Participants

Inviting participants

We aimed to improve the external validity of our experiment by increasing its population size, and thus mitigating coverage as well as sampling biases [Siegmund et al., 2015]. For



(a) Comprehension Task 1 (Q_8): Which of the following statements is true?

(b) Comprehension Task 2 (Q_9): When would line $<x>$ be executed?

Figure 5.8: Summary of the correctness our participants achieved in both program-comprehension tasks. Note that we abbreviate our examples with initial letter (V: Vim, E: Emacs) to improve the readability of the figure.

this purpose, we invited C developers from various open-source projects on GitHub who publicly posted their e-mail addresses on their GitHub page — aiming to mitigate ethical issues of inviting participants this way [Baltes and Diehl, 2016]. To sample projects, we built on previous studies of Liebig et al. [2010, 2011] and Medeiros et al. [2015], as well as GitHub's trending projects in October 2018 (e.g., FFmpeg,[23] redis[24]). Overall, we invited 7,791 developers with 1,117 starting and 521 finishing our experiment (≈7 %). The large population size is above the minimum (385) required to achieve a confidence level of 95 % for hypothesis testing. Moreover, the 521 participants who finished our experiment were almost evenly split between our two versions (260 to 261).

Participants' background

While each group of participants comprised varying experiences and roles, their demographics were highly similar. Namely, most participants were 32 to 42 years old for E_1 and 27 to 42 years old for E_2 (Q_1) with equal median and mean values (37 and 36, respectively). Most of our participants were male with E_1 involving three females, one other, and 16 who preferred not to state their gender, and E_2 involving eight females, five others, and nine who preferred not to state their gender (Q_2). Regarding their general programming experience (Q_3), most participants in E_1 sated 11 to 25 years, while most in E_2 stated 12 to 25 years (identical median and mean values of 20 years). We received almost equal responses for the C/C++ programming experience of both groups (Q_4), with most participants stating 8 to 20 years of experience (identical median and mean values of 15 years). Our participants were mainly developers (E_1: 250, E_2: 249), and some worked also as team managers (E_1: 76, E_2: 69), project managers (E_1: 57, E_2: 57), and in quality assurance (E_1: 40, E_2: 40). Note that we allowed multiple answers to Q_5, which is why these numbers do not add up to 100 %. In Q_6, our participants stated that they worked on a variety of open-source projects, such as the Linux kernel and its distributions, Git, PostgreSQL, or OpenSSL. On average, the participants' self-assessments regarding their programming skills (Q_7) were "advanced" (E_1: 3.32, E_2: 3.29). Due to the high similarity of both groups, we do not need to control for experience to mitigate threats to the validity of our experiment.

Baseline results

To further analyze whether we can compare the results of both versions of our experiment, we used the same baseline example (`Vim18`) in each. Without going into details (explained shortly), we display the results for the program-comprehension tasks in Figure 5.8, for the

[23] https://github.com/FFmpeg/FFmpeg

[24] https://github.com/antirez/redis

Table 5.10: Results of statistically testing our observations on annotations.

observation	test	p-value	effect	effect size	result
5 (Q_8: Task 1)	Fisher's exact test	0.18	(negative tendency)	—	not significant
6 (Q_9: Task 2)	Fisher's exact test	<0.001	negative	OR=0.74	61 % vs. 54 % correct
7 (Q_{10}: appropriateness)	Fisher's exact Test	<0.001	positive	OR=1.60	52 % vs. 64 % positive ratings
8 (Q_{11}: code quality)	Wilcox & Cliff's delta	<0.05	(positive tendency)	0.05–0.07	negligible

appropriateness rankings in Figure 5.9, and for the perceptions regarding the difficulty to work on the code in Figure 5.10. We can see that the participants of both versions performed almost identical when solving the tasks, and have similar perceptions regarding the use of annotations. This indicates that our sampling of participants did no bias or imbalance our results. So, we can compare between both versions to address our sub-objectives.

5.3.2 RO-T$_7$: Annotation Complexity and Program Comprehension

Program-comprehension tasks results

We display the results for our two program-comprehension tasks in Figure 5.8. For each task, we show how many of our participants submitted a correct or incorrect solution. In addition, our participants could state that they "don't know" the solution. We remark that in Figure 5.8a for `Vim15` the five and the one are actually two numbers (i.e., incorrect and "don't know," respectively). Finally, we provide an overview of all statistical tests we conducted in Table 5.10. As for our previous experiment, we discuss our results based on four different observations.

Observation 5: Refactoring had only marginal impact for Task 1

Results Task 1

We can see in Figure 5.8a that our data shows only marginal differences in the correctness between the original and refactored code examples for Task 1. Overall, the amounts of correct, incorrect, and "don't know" responses are highly similar for each code example. Moreover, we can see that `Vim13` and its refactored counterpart seem particularly hard to comprehend. For all other examples, the participants of either version performed quite well with over 80 % correctness for each.

Observation 6: Refactoring had a slightly negative impact for Task 2

Results Task 2

As we assumed, we can see in Figure 5.8b that Task 2 seemed to be far more challenging for our participants. The increased difficulty of comprehending how annotations can be configured, and thus impact the code, is clearly visible. Moreover, we can see that for almost all code examples (except `Vim13`), our participants performed slightly worse on the refactored version. Additionally, if we consider only incorrect solutions, our data shows that our participants performed better for all original versions.

Testing Observations 5 & 6

We used Fisher's exact test [Fisher, 1922] to test $H_{0\text{-}3}$ for both tasks. As we can see in Table 5.10, the results suggest that Observation 1 may be purely by chance ($p = 0.18$). In contrast, the differences we found for Observation 2 are significant ($p < 0.001$), which means that we reject $H_{0\text{-}3}$ for Task 2. To quantify the effect size, we computed an odds ratio (OR) [Bland and Altman, 2000] to compare the correctness between the two different versions of our code examples. This revealed that the odds for a correct answer decrease from 595:381 (original examples) to 532:462 (refactored examples), accounting for an OR of 0.74. Consequently, the chance to obtain a correct answer drops by roughly 7 % (from 61 % to 54 %) for the refactored code examples. Even if we pessimistically consider "don't know" answers as incorrect, this tendency remains ($p < 0.01$, $OR = 0.78$).

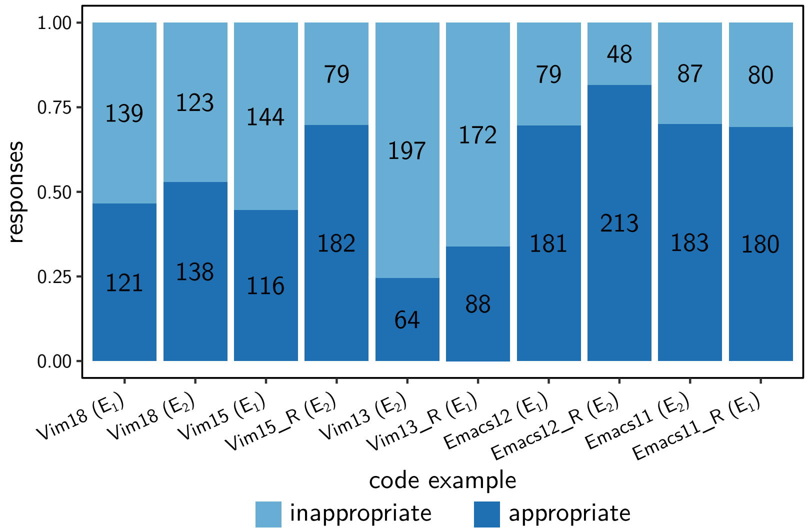


Figure 5.9: Participants' subjective rating of the C-preprocessor annotations used (Q_{10}).

RO-T_7: Annotation Complexity and Program Comprehension

Our results suggest that refactoring configurable annotations into a less complex form does not facilitate program comprehension.

5.3.3 RO-T_8: Developers' Perceptions of Annotations

Subjective ratings results

We summarize our participants' subjective assessment of the appropriateness of the C-preprocessor annotations used in our code examples in Figure 5.10. Furthermore., we display our participants' ratings of the difficulty to comprehend each example ($Q_{11\text{-}1}$) in Figure 5.10. We omit the plots for maintaining, extending, and bug fixing the code ($Q_{11\text{-}2\text{–}11\text{-}4}$), since these received highly similar responses. Analyzing the data, we can derive two more observations.

Observation 7: Refactored code was considered more appropriate

Appropriateness ratings

We can see in Figure 5.9 that our participants considered the refactored annotations as more appropriate in most cases. Only `Emacs11` is an exception, receiving a higher score compared to its refactored counterpart. The largest difference (25 %) occurred for `Vim15` (cf. Figure 5.7), for which our refactoring increased the rating from 45 % to 70 %. Finally, we can see that of all four code examples that received scores below 50 %, only `Vim13_R` is a refactored one (which was also the most challenging in Q_8 and Q_9). Our findings support previous evidence that developers prefer refactored annotations (e.g., less complex, more disciplined).

Observation 8: Refactored code was considered easier to work on

Activity ratings

When asking our participants how easily they could perform different activities on the code, we observed a similar pattern. Namely, we can see in Figure 5.10 that most participants perceived the refactored code examples as easier to comprehend (with the exception of `Emacs11`). However, the differences between the original and refactored code examples are not as large as they have been for Q_{10}. This observation is highly interesting, since it seems that rating annotations not on their own, but in the context of the code they configure, impacts developers' perceptions.

Testing Observations 7 & 8

To test for $H_{0\text{-}4}$, we used Fisher's exact test and an odds ratio for Observation 7. For Observation 8, we used the Mann-Whitney U test [Mann and Whitney, 1947] for the

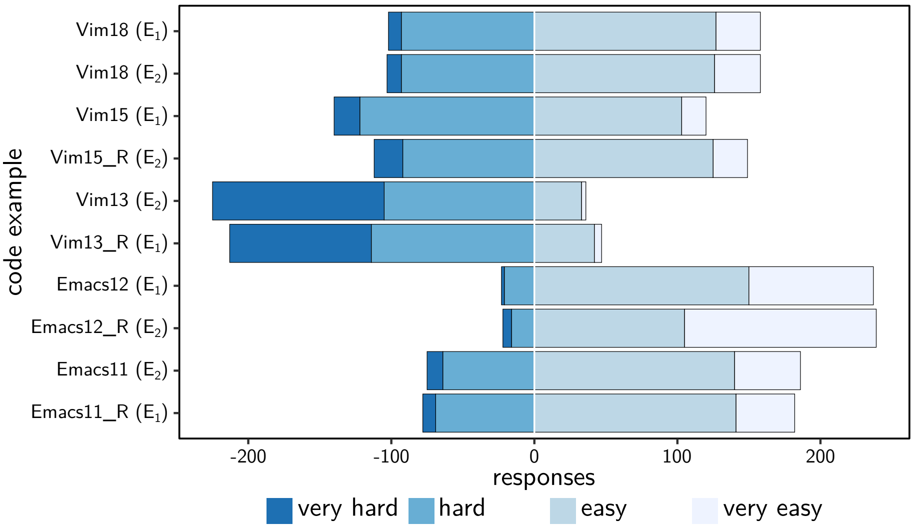


Figure 5.10: Participants' subjective comprehensibility rating for each code example ($Q_{11\text{-}1}$).

significance, and Cliff's delta [Cliff, 1993] to measure the effect size. As we can see in Table 5.10, Fisher's test implies a significant ($p < 0.001$) difference in the perception of original and refactored code examples. The ratings rise from 544:489 to 663:379, resulting in an odds ratio of 1.60 (meaning a 12 % rise in appropriateness ratings for refactored code). Finally, the Mann-Whitney U test also indicates significance ($p < 0.05$), but the effect size for all four questions (i.e., $Q_{11\text{-}1\text{–}4}$) is negligible with Cliff's delta ranging from 0.05 to 0.07.

RO-T_8: Developers' Perceptions of Annotations

We find that developers consider refactored (e.g., disciplined) annotations more appropriate. However, if they judge the annotations in the context of the surrounding source code, the differences become negligible.

5.3.4 RO-T_9: Differences in the Results

Differences in data

Combining our previous observations, we derived further insights that help define how an organization should implement its feature traceability. Namely, we first discuss the differences in our participants' performance and perception, which imply an interesting dilemma. Then, we reason on the impact of refactoring and configuring annotations on feature traceability.

Correctness Versus Preference

Performance versus perception

Arguably our most interesting insight are the contradicting findings with respect to correctness (**RO-T_7**) and perception (**RO-T_8**). Building on previous studies, we expected that our refactorings would improve the perceived quality, but it is surprising that they failed to improve our participants' actual performance — which actually worsened. At first glance, the worsened program comprehension conflicts previous studies (cf. Section 5.1.2). However, none of these studies compared performance and perception for the same code examples with the same participants, and they mainly involved novices. Apparently, our setup actually helped us to uncover an unknown dilemma.

The dilemma

The actual dilemma for developers is to decide whether they implement annotations in a style they prefer (e.g., more disciplined), even though our results suggest that this hampers their program comprehension. Existing research may have overestimated the importance of refactoring annotations, due to the focus on developers' perceptions instead of experimental data. Such surprising results have also been found in other contexts, for instance, for code clones [Kapser and Godfrey, 2008], code smells [Sjøberg et al., 2013], and refactoring in general [Tufano et al., 2015]. We are the first to uncover such a dilemma in the context of C-preprocessor annotations. Since annotations, and the C preprocessor in particular, are established mechanisms to implement traceability and configurability in variant-rich systems, this dilemma is highly relevant for many organizations and open-source communities.

Annotations Versus Code

Differences in perceptions

When we reflected in more detail on developers' perceptions, we found another difference that may cause and help to reason about the dilemma. Namely, the ratings for original and refactored code examples showed larger differences for Q_{10} than for Q_{11}. The former was concerned specifically with the C-preprocessor annotations, while the latter was concerned with the code quality in general. As a consequence, our participants may have considered other factors more important in Q_{11}, for example, identifier names, indentation, or code complexity. More precisely, many participants seem to have ignored the C-preprocessor annotations.

Impact of refactorings

Investigating our participants' qualitative remarks, we derived insights on the impact refactoring the annotations had on the general code. For example, the refactoring extract alternative function seemed to have benefited `Vim15_R`, but not `Emacs11_R` and `Emacs12_R`. A reason for this seems to be that the Emacs examples are smaller and less complex, so that the refactoring actually made the code more complex, as mentioned by some participants:

> *"[T]he function definitions are duplicated. This can confuse static analysis tools, but worse, it can confuse humans."*
>
> `Emacs11_R`

Aligning to previous studies, our participants stated to dislike fine-grained annotations:

> *"Preprocessor macros should not be used like this, ever, because it makes the code hilariously and needlessly complicated and very hard to comprehend."*
>
> `Vim18`

Nonetheless, the impact of disciplining annotations in `Vim13` was small, and `Vim13_R` still received the second-worst rating. So, the main problem may not have been the discipline, but the sheer amount of annotations that was equal in both code examples.

Problems of the source code

Since `Vim13` and `Vim13_R` received heavy criticism, we used open-coding on our participants' comments (232 and 195, respectively) to understand the reasons in more detail. We could identify three major themes: comprehensibility, complexity, and code quality. Interestingly, how often each theme appeared differed between both code examples. For `Vim13`, comprehensibility was stated in 49 % of all comments, while code quality appeared only in 14 %. In contrast, for `Vim13_R`, code quality was mentioned in 53 % of all comments, while comprehensibility occurred only in 34 %. So, disciplining annotations apparently resulted in our participants considering them more appropriate, but they also seemed to consider the actual code as more problematic. This finding aligns with our previous findings (cf. Chapter 3) that the code quality of a variant-rich system is key for its success, and indicates that C-preprocessor annotations may hide the actual problems of the source code.

Tracing Versus Configuring

Use of annotations

The previous points indicate that configurable annotations may hamper program comprehension. In contrast, our previous experiment (cf. Section 5.2.2) showed that annotations used only for tracing features facilitate program comprehension. We can see a similar pattern when comparing Task 1 and Task 2: Our participants performed considerably better when we asked them about the source code itself (Task 1) compared to asking them about the configurability of the code (Task 2). This underpins that configurable annotations add a layer of complexity that potentially impairs program comprehension. In contrast, using annotations only for tracing does not add this complexity, while still guiding developers when inspecting the source code. So, there is a fundamental difference between both uses. Considering all of our findings in this and the previous chapters, we would recommend that an organization should trace the features of its variant-rich system in the source code using non-configurable annotations. To enable configuring of a platform, the organization has to carefully assess which variability mechanisms to use and how it interferes with the source code as well as traceability.

RO-T$_9$: Differences in the Results

Reflecting upon the differences in our results, we found that:

- *Our participants performed worse on the refactored annotations, even though they perceived them as more appropriate.*
- *Improving the perceived quality of annotations seems to shift developers' attention to problems of the underlying source code.*
- *Using annotations for tracing, but not configuring, may help organizations trace the features in a variant-rich system without challenging program comprehension.*

5.3.5 Threats to Validity

Threats to validity

Similar to our previous experiment, we aimed to strengthen the external validity of this experiment. Again, we used an online setup, which is why we could again not control for all confounding factors. Next, we discuss the consequent threats to validity.

Internal Validity

Wording

We aimed to phrase our questions in a way that would not bias our participants. Still, some questions could have implied a critical perception of C-preprocessor annotations. We aimed to mitigate such problems by testing our setup with colleagues. Nonetheless, some participants stated that they found Q$_9$ and Q$_{10}$ ambiguous, which may also be related to our wording. So, the internal validity of our experiment is threatened by potential misunderstandings.

Employed refactorings

Code smells and refactoring choices are subjective, meaning that some developers may not perceive the same problems with the code. To prepare our examples, we built upon existing studies of experienced developers and the consequent refactoring advice. Moreover, we involved a large number of participants to tackle the problem that a small sample may distort our results in a specific direction. Still, there may have been better refactoring choices, which is why the construction of our examples remains a threat to the internal validity.

Completion time

We decided to minimize the time a participant required to complete our experiment to limit the dropout rate (e.g., due to time constraints, motivation, fatigue). However, on average, our participants completed the experiment in half an hour, which remains a quite long period that may have discouraged some developers from participating. While a shorter experiment may have been a better alternative, it would be challenging to impossible to observe some of our findings. So, while the time required to conduct our experiment remains a threat, we argue that we made a reasonable decision.

External Validity

Tooling

We aimed to improve the external validity of our experiment by using real-world code examples and inviting experienced software developers. Finally, our population size was large enough to ensure a high power of our statistical tests and allow us to identify reliable observations. Furthermore, we controlled our participants' experiences in a retrospective analysis to validate whether we could compare between both versions of our experiment. However, since we conducted an online experiment, some participants may have used additional tools, which threatens the external validity of our results.

Code examples

While we used real-world examples, they are both from open-source text editors. Moreover, we refactored the examples ourselves and they are rather small. Still, research suggests that open-source and industrial software exhibit similar characteristics regarding the use of C-preprocessor annotations [Hunsen et al., 2016], and that developers consider the refactorings we employed reasonable (cf. Section 5.1.2). Additionally, the design decisions we took to improve the internal validity and obtain enough data points (e.g., duration, two versions of the experiment, unfamiliar code examples) may threaten the external validity. Unfortunately, any empirical study involving human subjects involves such trade offs between internal and external validity [Siegmund et al., 2015].

Background factors

Our participants' background factors and characteristics threaten the external validity of our experiment. However, as we discussed before, many human factors are hard to impossible to control in program-comprehension research, for instance, participants' motivation, knowledge, memory strength, or cognitive processes. We mitigated such threats by controlling for our participants' experience, involving a large population size, limiting learning biases (i.e., using different code examples), reducing the workload, and removing unfinished responses. Also, the detailed responses we received in the open-text questions suggest that most of our participants were motivated.

Conclusion Validity

Interpretation of results

As for our previous experiment, we focus on observations that we can derive from our results. We detail our data before discussing it, aiming to avoid misinterpretation of correlations or observations [Amrhein et al., 2019; Baker, 2016; Wasserstein and Lazar, 2016; Wasserstein et al., 2019]. To allow other researchers to validate and replicate our experiment, we published all artifacts as well as the anonymized quantitative and qualitative data in an open-access repository. This way, we mitigate threats to the conclusion validity, but other researchers may derive different findings when replicating our experiment, for instance, due to the involved participants, code examples, or employed refactorings.

5.4 Summary

Chapter summary

In this chapter, we investigated how an organization can implement feature traceability in its variant-rich system. Initially, we discussed how to characterize different techniques for feature traceability based on three dimensions, and how the levels of these dimensions could potentially impact developers' program comprehension. Based on our own and other researchers' empirical studies, we defined two experiments. First, we conducted an experiment in which we compared the impact of different feature representations. The results indicate that virtual representations (e.g., annotations) facilitate program comprehension and do not impair developers' memory. Second, we conducted an experiment to compare the uses of feature annotations, indicating that using feature traces also for configuring challenges program comprehension — since it adds a layer of complexity.

Summarizing contributions

Our contributions in this chapter provide a detailed understanding and experimental data on the pros and cons of feature traceability. For practice, our results are a helpful means to define a traceability strategy and understand the impact of variability mechanisms. As a consequence, an organization can facilitate its developers' tasks and document feature locations. Regarding research, our data extended existing studies considerably and can help to design new techniques for implementing, managing, and querying feature traces. Moreover, we showed that developers' perceptions regarding the appropriateness of configurable annotations (i.e., of the C preprocessor) does not align to their performance — a dilemma that was not identified in previous studies. Abstractly, our results suggest the following core finding:

RO-T: Traceability

Implementing feature traceability facilitates developers' program comprehension and should ideally be independent of the variability mechanism used in the platform.

Connection to other research objectives

The findings we reported in this chapter align well with our previous experiences of re-engineering variant-rich systems and the expenses of feature location we reported in Chapter 3 (**RO-E**). Precisely, we showed that establishing feature traces in a system can facilitate feature location and program comprehension, and thus can help to reduce costs. Regarding developers' memory, which we discussed in Chapter 4 (**RO-K**), we could not find an improvement due to feature traces. Still, our results show that feature traceability improves program comprehension, and thus tackles the knowledge problem by recording important information directly in the source code. Finally, our results are closely related and guide several practices and processes, for instance, how to establish feature traceability throughout the different assets of a variant-rich system. We discuss these practices next in Chapter 6 (**RO-P**).

6. Round-Trip Engineering Practices

This chapter builds on publications at ESEC/FSE [Krüger and Berger, 2020b; Nešić et al., 2019], SPLC [Krüger et al., 2017a, 2020d; Strüber et al., 2019], VaMoS [Krüger and Berger, 2020a], Empirical Software Engineering [Lindohf et al., 2021], and the Journal of Systems and Software [Krüger et al., 2019c]

Chapter structure

In this last chapter, we synthesize our previous findings into frameworks for platform-(re-)engineering practices (**RO-P**), particularly for the planning and monitoring of a (re-)engineering project. To this end, we first define an overarching process model that provides an understanding of current practices and how they are connected (Section 6.1). Then, we provide concrete principles on the initial phase of (re-)engineering a variant-rich system: analyzing the domain to construct a feature model (Section 6.2). Lastly, we report a multi-case study on assessing the maturity of a variant-rich system, which helps an organization decide to what extent it wants to adopt platform engineering (Section 6.3). Our contributions in this chapter are processes and recommendations for planning, initiating, steering, and monitoring the (re-)engineering of a variant-rich system — with a focus on platform engineering, which we found to be economically beneficial (cf. Chapter 3). These contributions are particularly valuable for practitioners who can adopt them to plan their projects. For researchers, we provide a detailed understanding of current practices and highlight potential for further research.

Conceptual framework of practices

We display a more detailed overview of our conceptual framework regarding round-trip-engineering practices in Figure 6.1. Any variant-rich *system* has a certain *scope* (i.e., the domain in which it operates) and *maturity level* (i.e., the degree of platform engineering employed) — with both properties helping an organization to define the current as well as envisioned state of the system. The variant-rich system relies on the *reuse strategy* with its *processes* and consequent *activities* (e.g., bug fixing, quality assurance). These activities are performed by *developers* who use certain *practices* depending on the employed processes (e.g., continuous software engineering [Bosch, 2014; Fitzgerald and Stol, 2014] for integrating new features into a platform). Which (e.g., forking in GitHub) and how (i.e., for clone & own or as feature forks [Dubinsky et al., 2013; Krüger and Berger, 2020b; Krüger et al., 2019c, 2020e; Stănciulescu et al., 2015]) the developers use techniques depends on the current maturity level. We build upon our previous findings to derive the practices we discuss in this chapter. Consequently, the practices we describe directly impact the economics, knowledge, and traceability for (re-)engineering a variant-rich system.

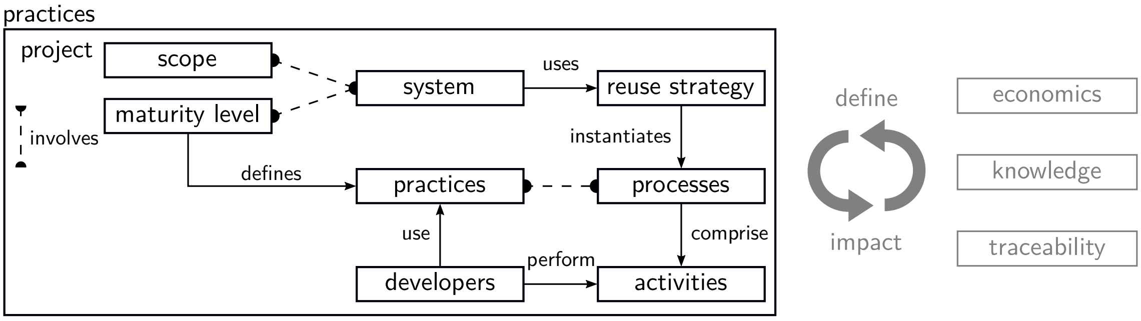


Figure 6.1: Details of the practices objective in our conceptual framework (cf. Figure 1.1).

6.1 The Promote-pl Process Model

Existing process models

Recall that we display the typical structure of software product-line process models in Figure 2.3 [Apel et al., 2011; Czarnecki, 2005; Kang et al., 2002; Northrop, 2002; Pohl et al., 2005; van der Linden et al., 2007]. While there exist minor differences between the individual process models, none of them seems to be in line with recent practices [Berger et al., 2020]. For instance, Pohl et al. [2005] propose to perform a "[c]ommonality analysis first," which focuses on proactively establishing a platform before any variant. However, industrial experiences [Assunção et al., 2017; Berger et al., 2013a, 2014a; Fogdal et al., 2016; Krüger, 2019b] indicate that incrementally extending or re-engineering a platform is far more common. In addition, the resulting platform is often evolved through its variants, from which customer-requested features are propagated to the platform [Krüger and Berger, 2020b; Strüber et al., 2019]. So, existing process models for platform engineering do not reflect current practices for adopting and evolving platforms (cf. Section 3.2).

Limitations of existing process models

For a better understanding of the (mingled) limitations of existing process models, we exemplify three that are based on insights from Danfoss [Fogdal et al., 2016; Jepsen and Beuche, 2009; Jepsen and Nielsen, 2000; Jepsen et al., 2007] and our own experiences [Krüger, 2019b; Krüger and Berger, 2020a,b; Krüger et al., 2019a,c; Kuiter et al., 2018b; Lindohf et al., 2021; Nešić et al., 2019; Strüber et al., 2019]:

Separated domain and application engineering: In contrast to the strict separation of domain and application engineering in existing process models, we experienced that both phases are highly interacting. In the same sense, Fogdal et al. [2016] report that at Danfoss "[...] there was no strict separation between domain and application engineering in the product projects [...]." So, a new process model for platform engineering should connect the two phases more closely.

Evolution of the platform: Existing process models assume that the platform itself is evolved to derive new variants. However, most organizations and open-source developers use feature forks to develop new features on a variant that they may merge at some point in time [Krüger and Berger, 2020b; Krüger et al., 2019c, 2020e; Stănciulescu et al., 2015]. Fogdal et al. [2016] state that Danfoss employs feature forks so that "[...] projects could keep their independence by introducing product-specific artifacts as new features. Later on, when a change was assessed, there would be a decision on whether the change should be applied to other products and thus should be integrated into the core assets." So, a new process model should consider different evolution patterns for variant-rich systems.

Adoption strategies: Finally, existing process models focus on the proactive adoption of a platform only. However, most organizations start with clone & own to avoid the high investments and risks associated with a new platform [Clements and Krueger, 2002; Krüger, 2016; Krüger et al., 2016a; Schmid and Verlage, 2002], and re-engineer or incrementally extend the resulting variants into a platform later on. For Danfoss, Fogdal et al. [2016] report that the re-engineering was required, but took more time than anticipated, particularly for introducing contemporary platform-engineering technologies: "Introducing pure::variants and establishing feature models for both code and parameters, and finally including the requirements, would take another two years." So, a new process model should describe the different adoption strategies.

Even though such limitations are well-known, research has not focused on updating the existing process models [Berger et al., 2020; Rabiser et al., 2018]. Of existing literature studies on platform engineering [C and Chandrasekaran, 2017], the ones of Laguna and Crespo [2013], Fenske et al. [2013], and Assunção et al. [2017] are the closest to this goal. Laguna and Crespo provide an overview of 23 publications on re-engineering a platform, based on which Fenske et al. derive a taxonomy, but no detailed process model. Assunção et al. synthesize an abstract process model (i.e., three high-level steps) for re-engineering a platform based on a systematic literature review (included in Table 6.1). While these works are complementary to ours, they mostly promote that a contemporary and updated process model is required — which we contribute.

Section contributions

In the following, we [Krüger et al., 2020d] present *promote-pl (PROces MOdel for round-Trip Engineering of Product Lines)*, an updated process model for platform engineering that we derived from existing research and our own experiences in the previous chapters. More precisely, we address the following three sub-objectives of **RO-P**:

RO-P$_1$ *Collect empirical data on contemporary platform-engineering practices.*

Our first sub-objective was to elicit practices (as partial orders of activities) reported in the recent literature. Such data helped us to construct promote-pl more systematically and not only based on our own experiences and intuitions, mitigating external threats to validity. The data we collected provides an overview for researchers and practitioners into recent platform (re-)engineering practices, helping them to scope and study such practices.

RO-P$_2$ *Construct a process model to order the elicited practices.*

Next, we used the partial orders we elicited to construct promote-pl itself. For this purpose, we matched the partial orders to each other based on our experiences and refined the model iteratively until we achieved agreement, leading to an abstract and a detailed version of promote-pl. Promote-pl serves as a contemporary process model for researchers and practitioners to understand how platforms are adopted and evolved in practice.

RO-P$_3$ *Analyze the relations of promote-pl to contemporary software-engineering practices.*

Lastly, we discuss particularly those parts of our empirical data and promote-pl that relate to contemporary software-engineering practices. Our results show how modern practices (e.g., continuous software engineering) relate to and impact platform engineering. Our discussion provides insights for researchers and practitioners into current advancements, and highlights open opportunities for research.

We report our methodology to elicit empirical data in Section 6.1.1 and the corresponding threats to validity in Section 6.1.5. Then, we study each of our sub-objectives individually in Section 6.1.2, Section 6.1.3, and Section 6.1.4.

6.1.1 Eliciting Data with a Systematic Literature Review

Process modeling

A process model helps describe *how* something is performed in a real-world process [Curtis et al., 1992]. In contrast, a development methodology defines an assumed best practice and a process theory (which is sometimes used synonymous) specifies a universal description of a process [Ralph, 2019; Sjøberg et al., 2008]. We construct promote-pl from empirical data and do not claim that it is complete or describes best practices, which is why we construct a process model according to the distinction of Ralph [2019]. Arguably, completeness (in terms of all possible process instances) and universal best practices are not achievable, considering the various technologies an organization may use, for instance, in terms of tools, variability mechanisms, or testing strategies.

Guidelines for process modeling

We are not aware of guidelines that focus specifically on constructing a process model. Instead, we relied on recommendations for defining process theories [Ralph, 2019; Sjøberg et al., 2008], which is why we used three information sources (explained shortly):

1. We built upon our own knowledge of the literature to resemble an integrative literature review [Snyder, 2019], which is a helpful means to critically reflect, synthesize, and re-conceptualize on a mature research area.
2. We conducted a systematic literature review by manually searching through the last five instances of relevant publication venues to identify and incorporate further contemporary practices.
3. We used our own experiences with practitioners to structure our data, match activities, and construct promote-pl.

We use these information sources to collect empirical evidence for constructing promote-pl, and thus strengthen its validity. Next, we describe each information source in more detail, and provide an overview of all selected publications in Table 6.1.

Knowledge-Based Literature Selection

Selection criteria

We relied on our experiences from previous literature reviews [Krüger, 2016; Krüger and Berger, 2020b; Krüger et al., 2019a; Nešić et al., 2019] to elicit an initial set of publications we deemed topical and relevant for our process model. Finally, we included all publications that fulfill the following inclusion criteria (IC):

IC_1 The publication is written in English.

IC_2 The publication reports activities of platform engineering, and defines at least one partial order (i.e., a sequence of execution) between these.

IC_3 The publication describes activities of recent (i.e., five years) experiences (e.g., case studies, interviews) or synthesizes them from such experiences (e.g., literature reviews).

We used an initial selection to scope our methodology and added further publications later on.

Publications selected from knowledge

We started with five publications that describe well-known process models for platform engineering as a baseline for promote-pl (marked BL in Table 6.1). Note that the publication of Northrop [2002] does not fulfill IC_2, because it defines no partial orders. We still included it, since this process model is well-established in research. In addition, we included 12 other publications (marked ER in Table 6.1) that partly involve our owns (asterisked in Table 6.1).

Systematic Literature Selection

Search strategy

We extended our knowledge-based literature selection using a systematic literature review [Kitchenham et al., 2015]. Precisely, we performed a manual search through five conferences (ASE, ESEC/FSE, ICSE, SPLC, VaMoS) and seven journals (EMSE, IEEE Software, IST, JSS, SPE, TOSEM, TSE). For each conference, we covered the last five instances of research and industry tracks, covering 2015 to 2019 (and 2020 for VaMoS). Considering the journals, we covered 2016 to 2020, including online-first publications. We conducted this search on April 7$^{\text{th}}$ 2020 through DBLP and the journals' websites for online-first publications. To elicit valuable platform-engineering practices, we focused on a recent period and major venues to ensure topicality and quality. We certainly missed publications, but we argue that our selection provides a reasonable overview and serves the goal of constructing a process model [Ralph, 2019; Snyder, 2019].

Inclusion criteria

Due to our search strategy and to assess the larger variety of publications, we extended our previous inclusion criteria as follows:

IC$_4$ The publication is published at the research or industry track of one of the aforementioned, peer-reviewed venues (i.e., excluding journal first or other publications).

IC$_5$ The publication reports a process that has been used in practice, not only a proposal (e.g., for new testing strategies).

With these additional inclusion criteria, we aimed to ensure that we elicit real-world data, and not visions or sketches of how a new research tool may be employed.

Publications included from the review

During our manual search, we selected 16 new publications (marked SR in Table 6.1). We remark that we do not count publications we already included during our knowledge-based literature selection. In the end, we identified 33 publications as relevant for constructing promote-pl, covering various venues (not surprisingly, mostly SPLC as the flagship conference on platform engineering) and platform-engineering processes (e.g., variant-based evolution, platform re-engineering).

Industrial Collaborations and Studies

Own experiences

Much of the research we present in this dissertation built on collaborations with industrial partners or on analyzing open-source communities that use platform engineering. For example, we collaborated with Axis to understand the economics of software reuse in Section 3.2 [Krüger and Berger, 2020b], and studied processes for re-engineering cloned variants in Section 3.3 [Krüger, 2017; Krüger and Berger, 2020a] as well for developing Marlin, Bitcoin-wallet, and Unix-like distributions in Section 4.3 [Krüger et al., 2019c, 2020e]. Also, in the remainder of this chapter, we report on experiences we elicited at Saab [Lindohf et al., 2021] and with different platform-engineering practitioners [Nešić et al., 2019]. We used the experiences we gained during these and our ongoing collaborations to analyze and order the data we elicited from the publications we selected. In particular, we resolved unclear partial orders during the construction of promote-pl and used our insights to discuss contemporary software-engineering practices.

Data Extraction

Extracted data

From each publication, we extracted standard bibliographic data. To construct promote-pl, we extracted all platform-engineering activities that are mentioned in their exact phrasing and partial order (if described). Note that we did not extract "standard" software-engineering activities, such as requirements elicitation. We also extracted the scope of the publication, for instance, variant-based evolution or platform re-engineering. Finally, we documented all specific software-engineering practices that are mentioned in the publications. We used a table to document the resulting data, which we summarize in Table 6.1.

Process-Model Construction

Method for constructing promote-pl

We employed the following methodology to construct promote-pl:

1. We analyzed the five baseline process models to identify their commonalities and differences, helping us to obtain a better understanding of the models and to start with unifying terminologies (leading to Figure 2.3). Still, the most important insight was the definition of how to identify and document partial orders.
2. We suggested relevant publications and conducted our systematic literature review.
3. We read each publication to decide which fulfilled our inclusion criteria (i.e., based on the order title, abstract, full text) and extracted the data we needed.
4. We listed all unique activities (over 150), based on which we resolved synonyms, defined terms, and abstracted activities using an open-card-like sorting method [Zimmermann, 2016]. For instance, we unified "product," "system," and "variant" to variant only. We performed this unification step carefully to not overly abstract activities, resulting in 99 distinct activities (cf. Table 6.1).
5. We matched the partial orders and activities to identify similarities and define partitions of promote-pl (e.g., adoption and evolution scenarios).
6. We constructed promote-pl by merging all partial orders based on re-appearing patterns and similar activities. Moreover, we structured all elements according to the identified partitions and resolved redundancies as far as possible.
7. We verified promote-pl by explaining it to another researcher during an interview in which we reasoned on our design decisions, alternative representations, and the data from which we derived each element. In the end, we performed smaller changes to make promote-pl more comprehensible and fixed a few unclear orders.

Using this methodology, we aimed to strengthen the validity of promote-pl and allow other researchers to replicate and verify its design.

6.1.2 RO-P$_1$: Contemporary Platform-Engineering Practices

Elicited data

We summarize all 33 publications and the activities we elicited from them in Table 6.1. Considering the scope, we can see that most publications cover the re-engineering of a platform and the evolution via variants — both topics have become major directions for platform-engineering research [Assunção et al., 2017; Berger et al., 2019; Krüger et al., 2020a; Nieke et al., 2019; Rabiser et al., 2018, 2019; Strüber et al., 2019]. In the following, we discuss our data to provide a detailed understanding of contemporary platform-engineering practices.

Activities

Variations in activities

Despite unifying the terminologies used in the publications, we kept 99 unique activities. These are too many activities to incorporate into promote-pl, but we kept them for two reasons: First, the publications report on a variety of software-engineering methodologies (e.g., model-driven, agile), domains (e.g., web services, automotive systems), tools (e.g., fully automated build systems), variability mechanisms (e.g., runtime parameters, C

Table 6.1: Overview of the 33 publications on platform-engineering processes we analyzed and the activities described in these (using our unified terminology).

	reference	**scope**	**activities (≺: partial order – •: separator – &: parallelism – |: alternatives – [...]: sub-activities)**
BL	Kang et al. [2002]	PE	scope & budget platform ≺ analyze platform requirements & model variability ≺ design architecture ≺ design system model ≺ refine architecture ≺ design assets • analyze variant requirements & select features ≺ design & adapt architecture ≺ adapt assets & build variant
BL	Northrop [2002]	PE	develop assets • engineer variant • manage platform • design architecture • evaluate architecture • analyze platform requirements • integrate assets • identify assets • test • configure • scope platform • train developers • budget platform
BL	Czarnecki [2005]	PE	analyze domain ≺ design architecture ≺ implement platform • analyze variant requirements ≺ derive variant
BL	Pohl et al. [2005]	PE	scope platform ≺ analyze domain [analyze commonalities ≺ analyze variability ≺ model variability] ≺ design architecture ≺ implement platform ≺ test platform • analyze variant requirements ≺ design variant ≺ derive variant [configure ≺ implement specifics ≺ build variant] ≺ test variant
BL	Apel et al. [2013a]	PE	analyze domain [scope platform ≺ model variability] ≺ implement platform ≺ analyze variant requirements ≺ derive variant
ER	Rubin et al. [2015]	PR	analyze commonality & variability [compare requirements ≺ diff variants ≺ model variability] ≺ design architecture [extract architecture ≺ evaluate architecture ≺ refine architecture & variability model] ≺ develop assets • merge variants ≺ refactor ≺ add variation points [diff variants ≺ refactor] ≺ model variability ≺ derive variant • analyze commonality & variability [model variability ≺ compare requirements & tests & diff variants ≺ refine variability model] ≺ extract platform
ER	Fogdal et al. [2016]	PR; EV	diff variants ≺ analyze variability ≺ model variability ≺ add variation points ≺ adopt tooling ≺ compare requirements ≺ map artifacts • develop assets [propose asset ≺ analyze asset requirements ≺ design asset ≺ implement asset ≺ test asset] • release platform [plan release ≺ produce release candidate ≺ test platform] • release variant [scope variant ≺ derive variant ≺ test variant]
ER	Assunção et al. [2017]	PR	analyze commonality & variability [locate features] ≺ model variability ≺ re-engineer artifacts
ER*	Krüger et al. [2017a]	PR	diff variants ≺ locate features ≺ model variability ≺ map artifacts
ER*	Kuiter et al. [2018b]	PR	model variability ≺ adopt tooling • domain analysis • implement platform • analyze variant requirements • derive variant • configure
ER	Martinez et al. [2018]	PR	train developers ≺ analyze domain ≺ model variability ≺ implement assets [analyze documentation | diff variants ≺ refactor]
ER*	Krüger et al. [2019a]	PR	analyze variability ≺ locate features ≺ map artifacts
ER*	Krüger et al. [2019c]	EV	propose asset ≺ analyze asset requirements ≺ assign developers ≺ fork platform ≺ implement asset ≺ create pull request ≺ review asset ≺ merge into test environment ≺ test asset ≺ merge into platform ≺ release platform
ER*	Nešić et al. [2019]	PE; PR; EP; EV	plan variability modeling ≺ train developers ≺ model variability ≺ assure quality [evaluate model • test model]
ER*	Strüber et al. [2019]	PR; EV	adapt variant ≺ propagate adaptations • analyze domain ≺ analyze variability ≺ locate features • extract platform • model variability • extract architecture • refactor • test platform • test variant
ER*	Krüger and Berger [2020a]	PR	train developers ≺ analyze domain ≺ prepare variants [remove unused code ≺ translate comments ≺ analyze commonality ≺ diff variants] ≺ analyze variability ≺ extract architecture ≺ locate features ≺ model variability ≺ extract platform ≺ assure quality
ER*	Krüger and Berger [2020b]	EV	Scope Variant ≺ Design variant ≺ derive variant ≺ adapt variant ≺ assure quality
SR	Weber et al. [2015]	PR	analyze variability [diff variants & identify fork points ≺ classify adaptations ≺ merge bug fixes | [name assets ≺ merge assets into hierarchy]] ≺ add variation points ≺ model variability ≺ locate features ≺ extract platform ≺ configure
SR	Yue et al. [2015]	EV	scope variant [analyze variant requirements • design variant • configure] ≺ budget variant ≺ design & implement variant [analyze variant requirements ≺ design & evaluate variant ≺ implement & adapt variant ≺ — | propagate adaptations] ≺ configure & test variant
SR	Capilla and Bosch [2016]	EP	analyze variant requirements ≺ define build rules ≺ configure & derive variant ≺ test variant
SR	Iida et al. [2016]	PE; PR	scope platform ≺ engineer platform [design system model ≺ design architecture & implement platform ≺ model variability] ≺ derive variant [design variant [design variant model ≺ scope variant ≺ select features] ≺ evaluate design [evaluate design logic ≺ configure] ≺ design variant ≺ implement variant] ≺ test variant
SR	Koziolek et al. [2016]	PE; PR	analyze domain [gather information sources ≺ define reuse criteria ≺ collect information ≺ analyze & model variability ≺ extract architectures ≺ evaluate results] ≺ budget platform
SR	Nagamine et al. [2016]	PE	engineer platform [analyze platform requirements ≺ design architecture & implement platform ≺ implement assets] ≺ derive variants • manage platform
SR	Cortiñas et al. [2017]	PE	analyze platform requirements [analyze domain ≺ scope platform ≺ model variability] ≺ design architecture ≺ evaluate architecture & map artifacts ≺ derive variant
SR	Gregg et al. [2017]	EV	fork platform ≺ test platform ≺ merge into platform
SR	Hayashi et al. [2017]	EV	derive variant [scope variant ≺ plan variant [define variant backlog ≺ estimate efforts ≺ plan development] ≺ build variant [create backlog ≺ time-box control]] • manage platform [scope & budget platform]
SR	Usman et al. [2017]	PE	model variability ≺ design system model ≺ derive variant
SR	Young et al. [2017]	PE	design architecture ≺ add variation points ≺ model variability ≺ configure ≺ derive variant
SR	Hayashi and Aoyama [2018]	EV	define variant backlog ≺ implement variant [analyze variant requirements ≺ implement assets ≺ test variant] ≺ add variation points [design variation points ≺ refactor ≺ test platform]
SR	Pohl et al. [2018]	PE; PR; EP	analyze platform requirements ≺ analyze commonality & variability ≺ design architecture ≺ implement platform • analyze variant requirements ≺ scope variant [identify assets & define new assets] ≺ implement assets ≺ integrate assets ≺ configure ≺ test variant • map artifacts • model variability • unify variability
SR	Horcas et al. [2019]	PE	Analyze Domain [Specify Properties ≺ model variability • analyze variant requirements [configure ≺ optimization]] • derive variant [configure ≺ integrate assets ≺ test variant] • implement platform
SR	Marchezan et al. [2019]	PR	plan development [assign developers ≺ assign roles ≺ analyze documentation] ≺ assemble process [select techniques ≺ adopt tooling ≺ assign tasks] ≺ extract platform [execute assembled process ≺ document assets ≺ document process]
SR	Sayagh et al. [2020]	PE; PR; EB; EV	add variation points ≺ adopt tooling • manage knowledge • resolve configuration failures • assure quality

BL: BaseLine; ER: Expert Review; SR: Systematic Review
EP: Evolution on Platform – EV: Evolution on Variant – PE: Platform Engineering – PR: Platform Re-engineering

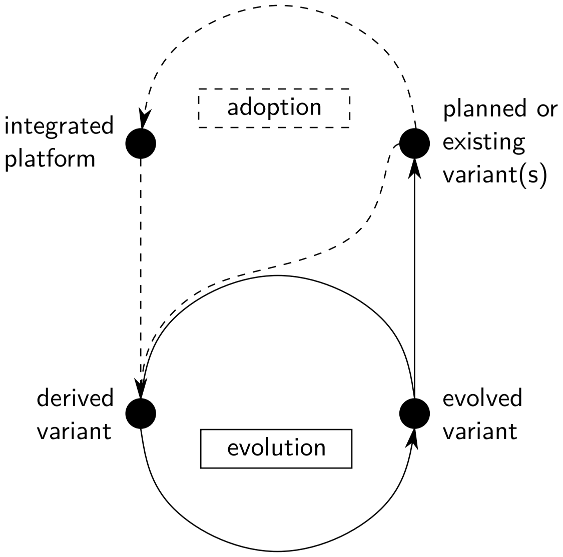


Figure 6.2: Abstract representation of promote-pl.

preprocessor), and development phases (e.g., variant derivation, business analysis). Second, the level of granularity in which activities are reported varies heavily, for instance, some publications simply refer to "derive product," while others detail the individual steps of this activity (e.g., "build"). Such differences make it impossible to unify all terms and incorporate all activities, which is why we focused particularly on those activities that appeared regularly in similar partial orders.

Partial Orders

Partial orders

We can see in Table 6.1 that we identified 42 partial orders (not counting alternatives or sub-orders). Interestingly, no exact order occurs more than once, due to the variations in activities. However, we can see similarities between orders within the same scope (e.g., platform re-engineering), while the orders are rather dissimilar between scopes. This finding highlights that we need an updated process model to incorporate other scopes and practices besides the proactive adoption.

Ambiguity of activities

Besides the variations in activities, a major reason for the high dissimilarity of the partial orders seems to be ambiguity of what a certain activity encompasses. For example, "analyze domain," "analyze commonality/variability," and "scope platform" are closely related, and thus combined in many partial orders. Interestingly, the exact ordering of these activities varies or they are sub-activities of each other. Consequently, there seems to be a lack of agreement or missing understanding of how specific activities are defined. We tackled this problem by carefully reading all definitions in the publications and by relying on the descriptions of Pohl et al. [2005] to reason about activities.

RO-P$_1$: Contemporary Platform-Engineering Practices

By eliciting platform-engineering practices, we found a diverse set of activities and processes, which do not align well to established process models. In addition, many activities and terms seem to be ambiguous to researchers and practitioners.

6.1.3 RO-P$_2$: Promote-pl and Adaptations

Abstract promote-pl

We display the abstract representation of promote-pl in Figure 6.2. The *adoption* phase involves all three adoption strategies for platform engineering [Clements and Krueger, 2002;

Krueger, 2002; Schmid and Verlage, 2002]: proactively based on set of planned variants, re-engineering existing variants, or incrementally extending one variant (i.e., the "planned or existing variant" beomces the "derived variant"). In the *evolution* phase, a derived variant is evolved by integrating new features (cf. Section 3.2.3). That variant may be evolved on its own (i.e., clone & own) or is integrated back into the platform by merging the complete variant or its features (i.e., returning to *adoption*).

Detailed promote-pl

In Figure 6.3, we display the detailed representation of promote-pl. We customized UML activity diagrams [Object Management Group, 2017] to improve the comprehensibility, resulting in nine diagram elements (displayed in the bottom left in Figure 6.3):

1–3) We indicate the locations of six overarching processes (i.e., adoption, evolution, management) that relate to the abstract representation of promote-pl (cf. Figure 6.2), but do not impact the actual model.

4) *Start Nodes* essentially keep their meaning from standard UML. However, in standard UML a workflow would be initiated at all start nodes at the same time, whereas we allow only a single start node to initiate promote-pl.

5–6) *Activities* and *Activity Edges* keep the same representations and meanings as in standard UML.

7) *Concurrent Activities* are related to fork and join nodes in standard UML. We simplified our diagram by connecting activities that are (or can be) performed simultaneously with dashed arrows.

8) *Decision Nodes* keep their meaning from standard UML, but we allow them to have only one outgoing activity edge to represent optional workflows.

9) *Situational Alternatives* simplify the representation of two workflows: First, to display that one workflow (i.e., from "test platform" to "integrate variants") is only relevant while re-engineering variants. Second, to display that the start node "develop variant" also represents the incremental adoption strategy.

Note that promote-pl does not include end nodes, since ending the evolution via its round-trip engineering style would imply that the variant-rich system is discontinued.

A Different Decomposition

Change in decomposition

As we motivated and our data confirms, organizations do not strictly distinguish domain and application engineering. Instead, different adoption and evolution strategies for a platform and its variants are more important for organizations. This represents a different decomposition, resulting in teams that employ domain and application engineering in parallel, for instance, while evolving a variant of the platform (cf. Section 3.2.3). In promote-pl, we represent this major change in the primary concern of interest by decomposing adoption and evolution processes, instead of domain and application engineering. Since this is the main difference to existing process models, we now describe each of the overarching processes in promote-pl in more detail.

Management Process

Platform management

Some baseline process models comprise management activities — sometimes as a separated phase, but more often integrated into domain engineering. Our data aligns to the process model of Northrop [2002], indicating that all management activities should run in parallel to the actual platform engineering. The management process (↻) involves seven mingled

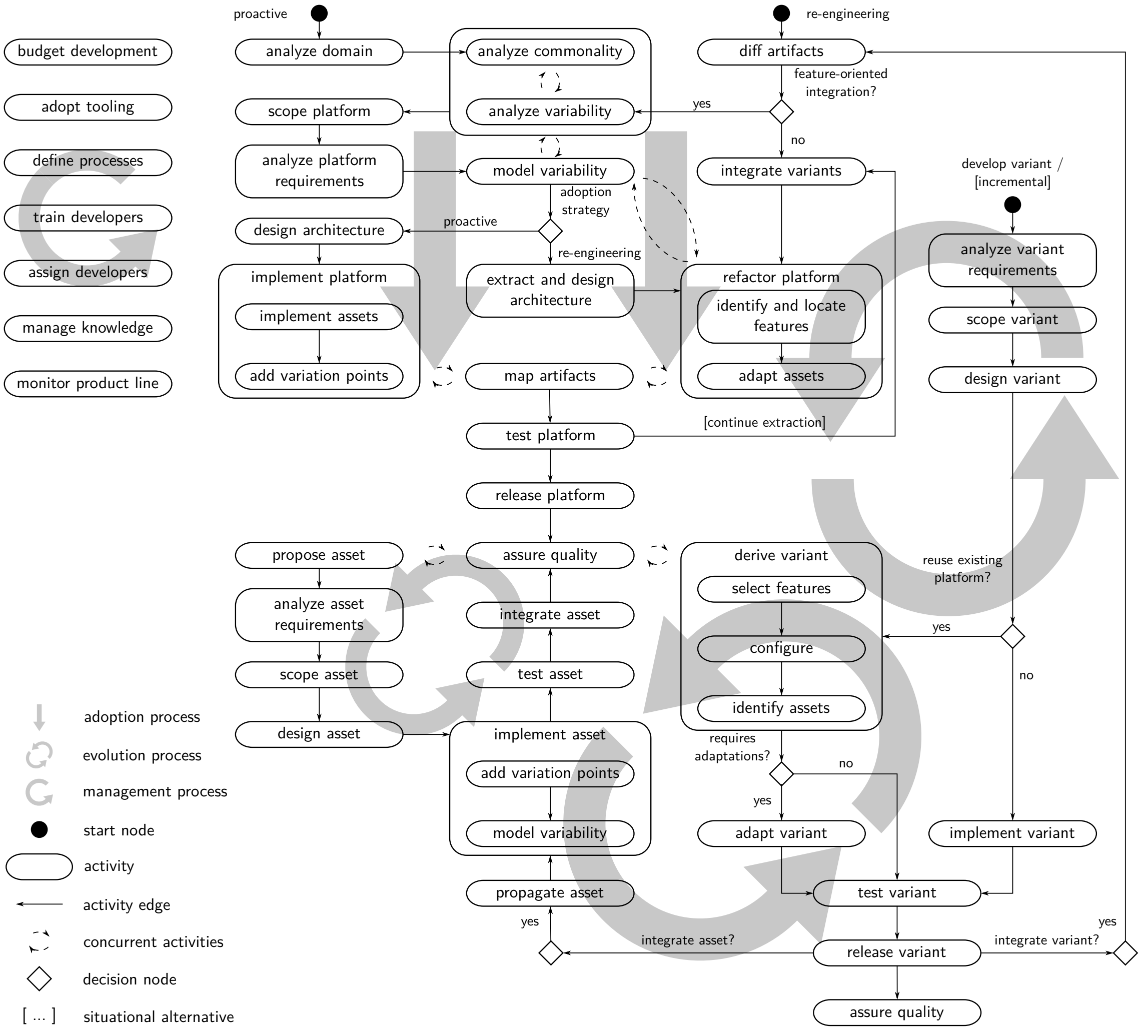


Figure 6.3: Detailed representation of promote-pl.

activities that are practically important to enable an organization to successfully plan and employ platform engineering: “budget development,” “adopt tooling,” “define processes,” “train developers,” “assign developers,” “manage knowledge,” and “monitor product line.” As identified by Rabiser et al. [2018], such activities have gained little attention in recent research compared to developing the platform itself. In this dissertation, we provide important insights that guide particularly these activities and relate them to the actual development. Namely, we provide economic data (cf. Chapter 3) that is relevant for management (e.g., budgeting, training) and development activities (e.g., scoping). We elicited data that is relevant for managing knowledge (cf. Chapter 4) and investigated different techniques (and consequent tooling) for tracing features (cf. Chapter 5), which both impact not only management but also development activities (e.g., mapping artifacts, adding variation points). As we can see, this dissertation focuses on management and earlier activities of adoption processes, for which we provide further insights in the remaining sections of this chapter (e.g., scoping, analyzing the domain, monitoring).

Adoption Processes

Proactive adoption

Not surprisingly, the proactive process (left ↓) is similar to domain engineering in the baseline process models. First, an organization performs a domain analysis and defines the consequent commonalities and variability between variants. Based on the results, the organization scopes the actual platform and defines its requirements, which leads to a variability model. Since the variability model captures the outcome of the commonality and variability analyses, these activities may be executed in parallel. All activities and their order up to this point represent "domain requirements engineering" and "domain design" in Figure 2.3, but as a more flexible process. The following activities focus on designing and implementing the platform (i.e., "domain implementation" in Figure 2.3). In contrast to the baseline process models, we make two activities explicit that occurred regularly in our data: "add variation points" and "map artifacts" (e.g., tracing features to assets) — for which we also provide further insights in Chapter 5 and the remainder of this chapter. Then, the organization tests, releases, and quality assures the platform. Overall, the proactive process resembles the domain-engineering phase explained by Pohl et al. [2005], except for the separated management process and the described adaptations we incorporated based on our data.

Re-engineering-based adoption process

The first major extension to existing process models is the re-engineering process (right ⬇). We found two main instantiations for this process, which both usually start with diffing artifacts. First, an organization may perform a full-fledged feature-oriented integration, essentially leading to the top-down analyses of the proactive process. However, after creating the variability model, organizations usually re-engineer and adapt the architecture of a suitable existing variant instead of developing a new one. The re-engineered platform is refactored to locate, adapt, and trace features that shall be integrated. Second, an organization can decide to integrate variants without scoping the platform in advance. Instead, the platform is constructed by integrating variants and refactoring the resulting code to distinguish features, representing a bottom-up process. To systematically manage the platform, the organization must create the variability model during the refactoring. The remaining activities are identical to the proactive process, but (especially for the second instantiation) the organization may decide to integrate further variants or features iteratively. We can see several similarities between the proactive and re-engineering process, but there are important differences, since the existing variants do not require a domain analysis (i.e., they are already established in the market) and allow to build on existing artifacts that can be refactored.

Incremental adoption process

The incremental process was not explicitly mentioned in any of the recent publications we identified. Still, this process is unproblematic, since it represents a special instantiation of the variant-based evolution. Namely, an organization implements a variant without a platform, and extends that variant by directly integrating new features (i.e., in contrast to creating separate clones for new features). So, promote-pl incorporates all established adoption strategies, and we can see that particularly the re-engineering and incremental processes mingle domain and application engineering.

Evolution Processes

Customer requests

The evolution of a platform is usually initiated by customers requesting new features. As a consequence, our three evolution processes all start with an organization developing a new variant and the typical application-engineering activities (i.e., "ananlyze variant requirements," "scope variant," "design variant"). After the organization analyzed what features and consequent assets are needed for the new variant, it can decide how to perform the actual development and whether to evolve the platform.

Platform-based evolution process

Platform-based evolution (left ↻) is the process implicitly defined in existing process models. In fact, the evolution in this process represents mainly domain-engineering, not application-engineering. So, a new feature for the platform is proposed, designed, implemented (also adding variation points in assets and incorporating it into the variability model), tested, and integrated. For this purpose, an organization can rely on feature forks [Krüger and Berger, 2020b; Krüger et al., 2020e], but the core idea is continuous integration and close cooperation with the platform engineers. After evolving the platform itself, the desired variant can be derived by selecting features, specifying the right configuration, and integrating the consequent assets. Note that assets can be identified fully automatically (e.g., using configuration managers), but in some cases this is not possible and developers have to manually identify and pull assets from their sources. Before testing, releasing, and quality assuring the variant, it may need individual adaptations that will not be integrated into the platform. While this is the idealized process, we found that most organizations rely on one of the following two processes.

Asset propagation process

When employing variant-based evolution using asset propagation (right ⟳), an organization configures, derives, and clones the variant from its platform that is closest to the new one. We [Krüger and Berger, 2020b] experienced that such a clone may involve the platform itself, for example, if the organization intends to develop highly innovative variants that may stay independent (cf. Section 3.2). After developing the new variant, the organization may consider the new feature relevant for other variants in the platform. In such a case, the corresponding assets must be propagated to the platform, leading to the second part of the platform-based evolution process. While the feature's assets must not be designed anew, they still require adaptations to fit the platform (e.g., considering feature interactions) before they can be tested and integrated. A key cost factor impacting this process is the co-evolution of the variant: If the variant stays independent for too long, it becomes more and more challenging to adapt it to the platform (or other variants) [Krüger and Berger, 2020b; Strüber et al., 2019]. In such cases, it is more likely that an organization employs the third evolution process.

Variant integration process

Variant-based evolution using variant integration (⟳) defines a process for re-integrating complete variants instead of individual features. This process is often employed when variants have not been synchronized with their platform for a longer time (e.g., not merged in version-control systems) [Krüger and Berger, 2020b; Strüber et al., 2019], for example, because of highly innovative features, clone & own development, or incremental platform adoption (i.e., incrementally developing and adding a new variant at a time). However, our data also shows that the re-integration of such variants follows the re-engineering-based adoption process of diffing and refactoring the co-evolved variants — relying on the concrete instantiation (i.e., top-down or bottom-up) that is more feasible. We can see that the two variant-based evolution processes switch domain and application engineering, since an organization first implements a variant before integrating features into their platform.

Domain and Application Engineering

Domain and application engineering

As we described, domain and application engineering in promote-pl are far more mingled and partly reordered compared to existing process models. As a consequence, the typical activities of these two phases occur, but the actual processes iterate between domain and application engineering. Since these changes are based on experiences reported in recent publications, it seems that the two phases are cross-cutting concerns. They are still important and helpful to organize platform engineering, but promote-pl is an essential update to provide an practice-oriented, comprehensive, and contemporary process model.

RO-P$_2$: Promote-pl and Adaptations

While constructing promote-pl, we learned:

- *The primary concern of interest in platform engineering changed from domain and application engineering to adoption and evolution.*
- *The management process runs in parallel to development processes, but its activities are less often investigated in research.*

- *Existing process models required several adaptations to reflect on all adoption strategies and evolution processes.*
- *Variant-based evolution using asset or variant integration are the main processes to evolve a platform.*
- *Domain and application engineering represent cross-cutting concerns for contemporary practices, instead of the primary decomposition.*

6.1.4 RO-P$_3$: Contemporary Software-Engineering Practices

Contemporary practices

While analyzing the publications we display in Table 6.1, we also identified other software-engineering practices that have been combined with platform engineering. In the following, we discuss how these practices are related to promote-pl. So, we provide an overview of contemporary, trending software-engineering practices that are often combined with platforms to facilitate developers' activities.

Continuous Software Engineering

Continuous platform engineering

Continuous software engineering [Bosch, 2014; Fitzgerald and Stol, 2014; Humble and Farley, 2010] aims to improve the integration of different software-engineering phases, leading to contemporary practices, such as DevOps or continuous integration, deployment, and testing. The publications we analyzed emphasize that continuous practices are employed for variability management [Gregg et al., 2017; Pohl et al., 2018], with Berger et al. [2020] also reporting that their industrial partners demand respective tools and methods. To develop variant-rich systems continuously, an organization actually requires a configurable platform. For example, continuous deployment and testing can only be employed if a variant can be automatically configured, assembled, and tested — otherwise, the organization can only deploy a single (cloned) variant. Similarly, continuous integration focuses on rapidly re-integrating changes of different developer teams (e.g., using feature forks), which requires a platform.

Continuous and promote-pl

In promote-pl, we address continuous software engineering by integrating domain and application engineering more closely in iterative, round-trip-like processes. So, we resolve discrepancies that exist between the prevalent re-engineering-based or iterative adoption processes and continuous software engineering. Moreover, we integrate variant-based evolution processes, which differ based on the degree of co-evolution between the platform and its variants (see "release variant" wiht the decision nodes "integrate asset" and "integrate variant"). Overall, promote-pl emphasizes the close connection of platform engineering and continuous software engineering — particularly with proactive and incremental adoption as well as platform evolution and asset integration calling for employing continuous practices to reduce co-evolution, and thus costs.

Clone Management and Incremental Adoption of Platforms

Clone & own and promote-pl

Organizations typically rely on clone & own to develop variants [Berger et al., 2013a, 2020; Krüger, 2019b], often based on branching or forking mechanisms of version-control systems [Dubinsky et al., 2013; Krüger and Berger, 2020b; Stănciulescu et al., 2015; Staples and Hill, 2004]. To facilitate the co-evolution of variants [Strüber et al., 2019] before investing into a platform, different frameworks offer clone management to synchronize variants and keep an overview understanding [Antkiewicz et al., 2014; Montalvillo and Díaz, 2015; Pfofe et al., 2016; Rubin and Chechik, 2013a; Rubin et al., 2012, 2013]. A first step towards managing clones more systematically is to govern branching and merging practices by defining explicit rules and models that define when a clone should be created and re-integrated [Dubinsky et al., 2013; Staples and Hill, 2004]. Still, the increasing maintenance overhead of clone & own may require the adoption of a platform at some point. To avoid the risks of big-bang adoptions that integrate all variants at once (e.g., disrupted development, costs) [Clements and Krueger, 2002; Hetrick et al., 2006; Krüger et al., 2016a; Schmid and Verlage, 2002], researchers have proposed incremental adoption strategies to evaluate the benefits and costs of adopting a platform iteratively [Antkiewicz et al., 2014; Fischer et al., 2014; Krüger et al., 2017a]. Similarly, a common practice is to directly combine configurable platforms with clone & own [Berger et al., 2020; Krüger and Berger, 2020b]. Promote-pl incorporates clone & own as well as an incremental adoption process for platform engineering, and does also allow to combine both within a unified model.

Dynamic Variability and Adaptive Systems

Dynamic variability and promote-pl

Late and dynamic binding [Kang et al., 1990; Rosenmüller, 2011] are required to support adaptive systems, for example, for cloud [Dillon et al., 2010], cyber-physical [Wolf, 2009], or micrsoervice [Thönes, 2015] systems. These systems must adapt at runtime to react to resource variations, environmental changes, or the availability of assets. Consequently, adaptive systems require a platform that uses parametrization to tune specific features. However, parametrization is not necessarily a variability mechanism to implement a variant-rich system (i.e., features cannot be enabled or disabled, only tuned), until the adaptive system actually requires individual features for specific variants. As a result, adaptive systems are often adopted incrementally or by variant-based evolution, with several publications reporting on the consequent platform engineering with dynamic variability [Capilla and Bosch, 2016; Krüger et al., 2019c; Martinez et al., 2018; Yue et al., 2015]. However, adopting and evolving such dynamic and adaptive platforms requires improved techniques and methods [Assunção et al., 2020; Krüger et al., 2017b]. With promote-pl, we account for this rather incremental adoption strategy, covering that variation points may be iteratively added.

Agile Practices

Agility and promote-pl

Agile software engineering [Meyer, 2014; Moran, 2015] focuses on small incremental changes, fast feedback loops, and customer involvement. Moreover, most agile methodologies (e.g., SCRUM, XP, FDD) have the notion of features as units of functionality (cf. Section 4.3.1) and foster automated techniques, which require a configurable platform (as for continuous software engineering). We identified experiences on agile practices in two publications: Fogdal et al. [2016] report that platform engineering and agility do not conflict each other, as long as the developers know that they deliver their assets to a shared platform. Slightly contradicting this insight, Hayashi et al. [2017] experienced that deriving variants is not well suited for agile methodologies, because the short development cycles prevent developers from learning best practices. Nonetheless, agile practices facilitated the evolution of their platforms. Such experiences suggest that agile practices are important for platform engineering, which are well supported by promote-pl with its focus on iterative evolution.

Simulation in Testing

Testipng and promote-pl

Three of the publications we identified [Capilla and Bosch, 2016; Fogdal et al., 2016; Iida et al., 2016] and experience reports from other organizations [Berger et al., 2020; Lindohf et al., 2021] mention the use of simulations for testing a platform or its variants. We cover the relevant activities in promote-pl, but current research in this direction is limited. Namely, using simulations for testing is different to using sampling techniques to test a set of software variants, since safety-critical systems (e.g., power plants, cars, planes) require simulations to also test the interactions between hardware and software. In addition, simulations may exhibit other properties (e.g., for data transfers) or additional features (e.g., for monitoring the simulation), which require simulation-specific assets and variation points [Lindohf et al., 2021]. Promote-pl involves all relevant activities and can help researchers design supportive techniques for simulating test environments for platforms.

RO-P$_3$: Contemporary Software-Engineering Practices

We identified five contemporary software-engineering practices that build upon, are required by, or work well with platform engineering: continuous software engineering, clone management, adaptive systems, agility, and simulation in testing. Promote-pl covers all activities required to instantiate these practices for platform engineering, advancing on existing process models.

6.1.5 Threats to Validity

Threats to validity

We are not aware of guidelines for constructing a process model. For this reason, we employed recommendations of similar research methods in a self-defined methodology. So, our methodology for constructing promote-pl may have introduced threats to its validity.

Construct Validity

Data interpretation

The publications we analyzed involved different terminologies. Moreover, the same activities have occurred in varying partial orders, which indicates that the constructs and the concepts they express vary between some publications. Consequently, the partial orders we elicited may not fully represent the ones intended by the original authors. To mitigate this threat, we elicited our data from 33 publications, read the description of all activities to avoid misunderstandings, and reasoned on all decision based on our own knowledge.

Internal Validity

Methodology

Our construction process for promote-pl may have introduced threats. For instance, we may have disregarded relevant publications, overlooked data, or not derived the most appropriate process model. We limited such threats by adapting recommendations for constructing process theory [Ralph, 2019], which suggest that secondary studies (e.g., a systematic literature review) are a reliable method to limit potential biases introduced by personal knowledge. Also, we verified promote-pl during an interview to ensure that our model is reasonable and comprehensible.

External Validity

General applicability

Our goal was not to construct a universal process theory, which is hardly possible. We aimed to capture contemporary platform-engineering practices and provide a model to describe and connect those practices. As a result, different properties of platform-engineering processes may limit the applicability of promote-pl in an organization, for instance, because of varying technologies (e.g., for feature traceability) or the involved developers (e.g., their knowledge). We aimed to mitigate such threats by synthesizing data from 33 publications and considering our own experiences from industrial collaborations. So, we argue that our abstraction provides a good understanding for any researcher or practitioner, but requires some adaptations to an organization's properties.

Conclusion Validity

Data and promote-pl

Other researchers may construct a different process model depending on the publications they select, their knowledge, or the construction process they employ. We mitigated such threats by explaining our methodology in detail, reasoning about our design decisions, and documenting all publications as well as activities we considered (cf. Table 6.1). Based on this information, other researchers can verify and replicate promote-pl.

6.2 Feature-Modeling Principles

Studying feature-modeling principles

As we can see in promote-pl (cf. Figure 6.3), a variability model (cf. Section 2.3) represents a key artifact for adopting a platform. For instance, a variability model helps to document the domain, scope the platform, manage features, configure variants, or perform automated analyses of the platform [Benavides and Galindo, 2018; Benavides et al., 2010]. As a consequence, a variability model is also highly relevant for all of our previous research objectives, since it supports (1) economic decisions by defining which features to (re-)engineer (**RO-E**); (2) developers' knowledge by documenting features and their dependencies (**RO-K**); and (3) feature traceability by (ideally) mapping its features to other artifacts (**RO-T**). Note that we focus on feature models (and feature diagrams as their visual representation), since they are the most commonly used form of variability models [Berger et al., 2013a; Czarnecki et al., 2012; Kang et al., 1990; Schobbens et al., 2006]. Interestingly, despite around three decades of research, there is little knowledge on how to construct feature models. In fact, only Lee et al. [2002] provide some experience-based guidelines for feature modeling, which, however, are almost two decades old. A systematically consolidated set of best practices for feature modeling (i.e., principles) would help practitioners to construct feature models, as well as researchers to improve tools and processes.

Section contributions

In this section, we [Nešić et al., 2019] present 34 feature-modeling principles that we synthesized from a systematic literature review of 31 publications and an interview survey with ten feature-modeling experts (we display an overview of our methodology in Figure 6.4). In detail, we tackle three three sub-objectives of **RO-P**:

RO-P$_4$ *Define feature-modeling phases to structure principles.*

Based on our data, we derived eight phases for categorizing feature-modeling principles. We arranged these phases in the most reasonable order of execution to discuss our actual principles in a structured way. The phases help researchers and practitioners obtain an overview understanding of the principles and their execution in practice.

RO-P$_5$ *Synthesize evidence-based feature-modeling principles.*

The core contribution of this section are the 34 feature-modeling principles. Note that the principles vary in their level of detail and their scope (i.e., some are applicable not only to feature modeling), since we report all that were stated in our data — and thus are apparently important. So, for practitioners and researchers, these principles define best practices for constructing feature models, highlight pitfalls, and indicate future research.

RO-P$_6$ *Discuss the properties and implications of the elicited principles.*

In the end, we discuss the different properties and implications of the feature-modeling principles. For instance, our sources highlight again that re-engineering cloned variants seems to be the most common adoption strategy for a platform, and that different trade-offs between principles exist. Discussing such issues helps practitioners decide which principles to employ in what context, and researchers to identify potential for improving processes and techniques.

We provide an open-access repository with our data (e.g., identified publications, relevant quotes).[25] In Section 6.2.1 and Section 6.2.2, we report the details of our systematic literature review and interview survey, respectively. Then, we describe our methodology to synthesize principles from both sources in Section 6.2.3. We describe the feature-modeling

[25] https://bitbucket.org/easelab/featuremodelingprinciples

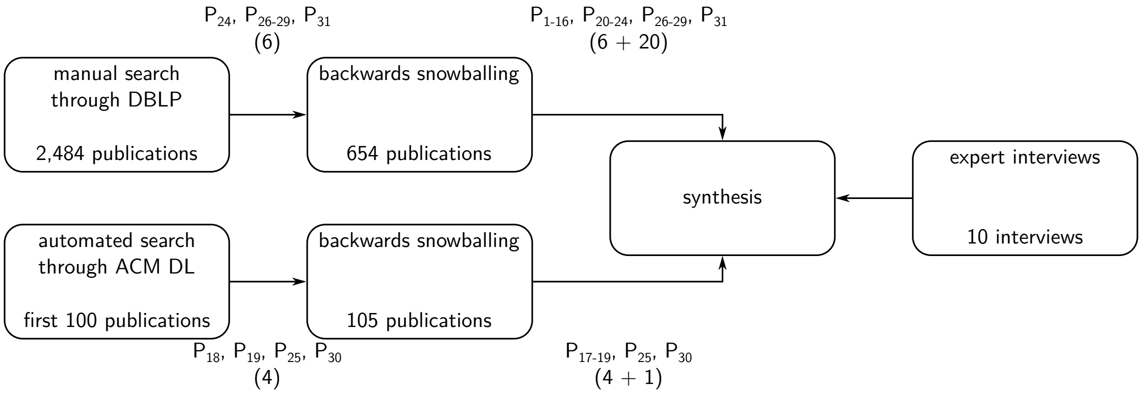


Figure 6.4: Overview of our methodology for eliciting feature-modeling principles.

phases in Section 6.2.4 and the corresponding principles in Section 6.2.5. Finally, we discuss important properties of the principles in Section 6.2.6 to tackle our last sub-objective.

6.2.1 Eliciting Data with a Systematic Literature Review

Systematic literature review

We conducted a systematic literature review to elicit feature-modeling principles from existing publications (cf. Figure 6.4). Note that we employed a more light-weight methodology regarding two steps: First, we omitted the quality assessment, since it is less relevant for structuring previous experiences [Kitchenham et al., 2015], and because the publications we identified followed no common methodology (e.g., experience reports, surveys, case studies). Second, we did not compute the typical statistics on the selected publications, since we were interested in the qualitative data in the publications and not their metadata (identical to our other systematic literature reviews in this dissertation).

Search Strategy

Manual search

We started our systematic literature review with a manual search of relevant publication venues through DBLP. Namely, we analyzed the last five instances (in June 2018) of industry and research tracks of ASE, ESEC/FSE, FOSD, ICSE, ICSME, ICSR, MODELS, SANER, SPLC, and VaMoS. As we display in Figure 6.4, we analyzed 2,484 publications in this step. We read titles, abstracts, and the full texts if needed to identify six relevant publications (asterisked in Table 6.2). From this initial set of publications, we started a first round of backwards snowballing [Wohlin, 2014] without a defined number of iterations (i.e., if we identified another relevant publication, we employed backwards snowballing on that publication, too). Again, we read titles, abstracts, and full texts until we covered all publications referenced in the ones we deemed relevant. Overall, we covered 654 referenced publications and identified 20 relevant ones.

Automated search

To improve the confidence in our sample of publications, we conducted an automated search using the ACM Guide to Computing Literature and the following search string:

(+Experience Report +(Variability OR Feature) +(Specification OR Model))

We sorted the returned publications by relevance (other criteria are less feasible, e.g., the publication date or number of citations) and analyzed the first 100 entries. By iterating though titles, abstracts, and full texts if necessary, we identified four relevant publications (marked with two asterisks in Table 6.2). During a second round of backwards snowballing on these publications (covering 105 references), we identified one more publication that was relevant for our study.

Selection Criteria

Inclusion criteria

We employed the following inclusion criteria (IC) to select relevant publications:

IC_1 The publication is written in English.

IC_2 The publication describes one of the following:

$IC_{2\text{-}1}$ practitioners' experiences of applying feature modeling;

$IC_{2\text{-}2}$ practices identified by researchers via the analysis of variant-rich systems; or

$IC_{2\text{-}3}$ experiences of tool-vendors and educators while training feature modeling.

IC_3 The paper is peer reviewed or a technical report.

We intentionally included technical reports, since they often provide detailed insights into industrial practices that are highly valuable for our study.

Data Extraction

Data extraction

From each publication, we extracted standard bibliographic data as well as all details on reported feature-modeling practices. We focused on analyzing experiences and lessons learned to understand what practices were helpful for an organization for what reason. This extraction was performed by two researchers, with two other researchers confirming whether the data was understandable, relevant, and sufficiently detailed.

Elicited Data

Selected publications

In Table 6.2, we provide an overview of the 31 publications we included into our systematic literature review. Not surprisingly, most of the publications are industrial case studies (14) and experience reports (10). Additionally, we included four open-source case studies, one questionnaire, one interview survey, and one literature review. The publications cover more than 20 years, have been published at different venues, refer to more than ten unique notations for feature modeling, and include 14 domains. Our data also aligns to the finding that re-engineering a variant-rich system is the most common adoption strategy (cf. Chapter 1), with 18 publications reporting on a re-engineering, one on a proactive, two on multiple, and one on a teaching context. In the last column, we show how many of our principles are based upon the practices reported in a publication.

6.2.2 Eliciting Data with an Interview Survey

Interview survey desing

To complement our systematic literature review, we conducted ten semi-structured interviews with practitioners and tool-vendors of nine organizations. We designed and conducted the interviews independently from our systematic literature review to avoid biases towards certain practices. While one interview was rather short (i.e., 25 minutes), the others took around one hour on average. If an interviewee applied feature modeling in their organization, we started with eliciting the context of the corresponding projects, involving properties, such as the domain, team structure, and variability mechanism. With tool-vendors, we aimed to understand the typical context in which their customers operate. Afterwards, we asked our interviewees what practices they employed for constructing and evolving feature models, for example, considering responsibilities, use cases, and model properties (e.g., size, dependencies). To conclude an interview, we asked what benefits and challenges are connected to feature modeling, aiming to understand which practices were successful. We conducted the interviews in person and via phone or Skype, recorded them, and created transcripts. In Table 6.3, we provide a brief overview of our interviewees and their platforms.

Table 6.2: Overview of the 31 publications on feature-modeling principles (fmp). One and two asterisks define the publications that served as starting sets for our first and second round of snowballing, respectively.

id	reference	method	notation/tool	domain	context	# fmp
P_{1}	Cohen et al. [1992]	ICS	FODA	defense	re-engineering	10
P_{2}	Griss et al. [1998]	ER/MD	FODA, UML	telecommunications	n/a	6
P_{3}	Kang et al. [1998]	ER/MD	FODA/FORM	multiple	n/a	6
P_{4}	Kang et al. [1999]	ICS/MD	FODA/FORM	telecommunications	n/a	8
P_{5}	Hein et al. [2000]	ICS	FODA, UML	automotive	re-engineering	22
P_{6}	Lee et al. [2000]	ICS	feature model	elevators	re-engineering	2
P_{7}	Kang et al. [2002]	ER	FODA/FORM	n/a	n/a	4
P_{8}	Lee et al. [2002]	ER	FODA	n/a	n/a	13
P_{9}	Kang et al. [2003]	ICS/MD	FODA/FORM	inventory system	re-engineering	7
P_{10}	Sinnema et al. [2004]	ICS	COVAMOF	multiple	re-engineering	3
P_{11}	Gillan et al. [2007]	ER	n/a	telecommunications	re-engineering	2
P_{12}	Hubaux et al. [2008]	OSCS	OmniGraffle	e-government	re-engineering	4
P_{13}	Schwanninger et al. [2009]	ICS	Pure::variants	industrial automation	re-engineering	8
P_{14}	Berger et al. [2010]	OSCS	Kconfig/CDL	Linux/eCos	n/a	4
P_{15}	Dhungana et al. [2010]	ICS	decision oriented	industrial automation	re-engineering	5
P_{16}	Hubaux et al. [2010]	LR	feature diagrams	n/a	n/a	4
P_{17}	Iwasaki et al. [2010]	ER	FORM	network equipment	re-engineering	3
P_{18}**	Boutkova [2011]	ER	Pure::variants	automotive	re-engineering	6
P_{19}**	Hofman et al. [2012]	ER	FODA UML profile	healthcare	re-engineering	3
P_{20}	Berger et al. [2013b]	OSCS	Kconfig/CDL	multiple	n/a	4
P_{21}	Berger et al. [2013a]	Q	multiple	multiple	multiple	5
P_{22}	Manz et al. [2013]	ICS	feature model	automotive	re-engineering	4
P_{23}	Berger et al. [2014b]	OSCS	Kconfig/CDL	systems software	proactive	4
P_{24}*	Berger et al. [2014a]	IS	multiple	multiple	multiple	5
P_{25}**	Derakhshanmanesh et al. [2014]	ER	Pure::variants	automotive	re-engineering	5
P_{26}*	Chavarriaga et al. [2015]	ICS	SPLOT	electrical transformers	re-engineering	10
P_{27}*	Gaeta and Czarnecki [2015]	ICS	SysML	avionics	re-engineering	3
P_{28}*	Lettner et al. [2015]	ICS	FeatureIDE	industrial automation	re-engineering	15
P_{29}*	Fogdal et al. [2016]	ICS	Pure::variants	industrial automation	re-engineering	5
P_{30}**	Nakanishi et al. [2018]	ER	FODA/FORM	n/a	teaching SPL	1
P_{31}*	Pohl et al. [2018]	ICS	Pure::variants	automotive	n/a	5

ER: Experience Report – ICS: Industrial Case Study – IS: Interview Survey – LR: Literature Review – MD: Method Definition
OSCS: Open-Source Case Study – Q: Questionnaire

Interviewees

Interviewees' experiencees and platforms

Our interviewees had multiple years of experience regarding feature modeling in practice, either from their own organization or as consultants. $I_{1–3}$ have different roles in organizations that develop feature-modeling and platform-engineering tools. They have decades of experience from consulting organizations that employ platform engineering, mostly for embedded systems. I_4 is a software architect for a large car manufacturer (≤99,000 employees, >400,000 cars per year) and is involved in the variability management of their three major platforms. I_5 is also a software architect working on the variability management for a large (≤25,000 employees) component vendor for industrial and end-user applications with an extensive set of variants that originated from clone & own. I_6 works as department lead (acting as architect and developer) for a small (≤50 employees) consulting organization that provides customized e-commerce applications, and was involved in developing a platform with 400 to 500 configurable variants. I_7 is a consultant who has experiences of using feature models with two different customers. I_8 is a team leader in a migration project for staff and course management. I_9 is a solution architect for a large international organization (≤140,000 employees) who works on a platform with around 15 variants for industrial automation. I_{10} is a team leader for a large automotive organization (≤30,000 employees), and is responsible for managing the configurable software for an embedded device that exhibits an exponential number of variants.

Interviewees' Platforms

Platform properties

The platforms of our interviewees who employ feature modeling in their organization have varying properties. I_4's platform comprises a number of hierarchically structured feature

Table 6.3: Overview of our ten interviews on feature-modeling principles.

id	role	domain	# features	# principles
I_1	consultant	(tool vendor)	n/a	14
I_2	consultant	(tool vendor)	n/a	10
I_3	consultant	(tool vendor)	n/a	6
I_4	architect	automotive	≥1000	6
I_5	architect	industrial automation	≤1000	6
I_6	architect/developer	web shops	≤40	5
I_7	consultant	(various)	n/a	4
I_8	team leader	e-learning	≥1000	11
I_9	architect	industrial automation	≤100	9
I_{10}	team leader	automotive	n/a	5

models that comprise hundreds (high levels in the hierarchy) to thousands (lower levels) of features. Moreover, the platform relies on various variability mechanisms to account for different suppliers. I_5's platform binds features statically at build time (e.g., using the C preprocessor) and has a single, moderately sized feature model. I_6's platform uses a single feature model and allows to configure features by using a self-developed Java preprocessor. I_8's platform involves runtime binding and one large feature model with few Boolean features. I_9's platform involves a configurable build system to configure C++ components and one moderately sized feature model. I_{10}'s platform comprises various feature models to manage components as well as the versions and fine-grained features of these components. Moreover, the feature models manage configuration parameters for each feature, and the platform uses a configuration file to self-adapt while booting.

6.2.3 Synthesizing Principles

Practices and principles

We read all publications and interview transcripts to identify feature-modeling practices, triangulated these, and synthesized them to principles. For this purpose, a practice refers to a concretely stated way of how something was or should be done. A principle is a generalized practice that (1) is concerned with an established feature-modeling concept; (2) is domain independent; and (3) is relevant in the context of contemporary software-engineering practices.

Synthesis

We extracted almost 190 instances of feature-modeling practices, which we documented in (typically) one to three sentences and with a link to the source. Next, we iteratively joined redundant practices, while keeping the different modeling contexts in mind (e.g., the employed modeling process, properties of the model and project). Practices that stemmed from different contexts were strong candidates for feature-modeling principles, since they seem to be domain independent and up-to-date. If we identified contradicting practices, we analyzed their contexts to decide whether they were depending on specific properties. For example, feature models for systems software regularly include cross-tree constraints, whereas most interviewees recommended to avoid such constraints. We analyzed this contradiction in more detail and found that the decision for or against modeling cross-tree constraints actually depends on the user who configures the feature model. Namely, if the feature model is intended for end users (e.g., online configurators [Thüm et al., 2018]), modeling cross-tree constraints ensures that the end users can derive valid configurations. In contrast, modeling cross-tree constraints may not pay off if domain experts of the organization configure variants, since they typically have the required knowledge (note that it may still be beneficial, as we discussed in Chapter 4). We continued with our iterative process until all four involved researchers reached consensus on the set of feature-modeling phases and principles. Moreover, our interviewees as well as additional collaborators from industry reviewed the principles and considered them valuable, improving our confidence in our results.

6.2.4 RO-P_4: Feature-Modeling Phases

Defining phases

In the beginning, we followed the categorization of Lee et al. [2002] to structure our principles. However, with an increasing and more diverse set of principles, we adapted this categorization. Finally, we assigned our principles to eight phases that we put into the most reasonable order for employing them in practice, but this order does not represent an actual modeling process. For instance, some principles are alternatives to each other, are optional, or have variations in themselves depending on the organization's context.

Phases of feature modeling

We defined the following eight phases to organize our principles:

Planning and Preparation (PP) involves principles that help an organization prepare the feature modeling itself, for instance, by defining the purpose of the model and the process for constructing it.

Training (T) involves principles for educating the involved stakeholders on how to construct a feature model (note that we identified training as an important cost factor in Chapter 3 and incorporated it as a management activity for promote-pl in Section 6.1).

Information Sources (IS) involves one principle for eliciting the information required to construct a feature model, which is directly related to Section 4.3.

Model Organization (MO) involves principles on how to design and structure a feature model, for instance, in terms of depths or decomposition.

Modeling (M) involves principles for constructing the feature model, and thus many are directly connected to promote-pl (cf. Section 6.1.3).

Dependencies (D) involves principles for handling dependencies between features, for instance, cross-tree constraints.

Quality Assurance (QA) involves principles for assessing that the constructed feature model actually covers the intended use case and platform.

Model Maintenance and Evolution (MME) involves principles that help an organization maintain a feature model, which is strongly related to, and impacted by, several principles of the previous phases.

Note that we plan to derive an actual feature-modeling process that is also connceted to promote-pl in our future work.

Phases with fewer sources

Interestingly, we identified fewer sources for supporting the principles related to planning and preparation as well as training compared to the other phases. Mainly, the sources supporting these principles are industrial case studies or our interviews with tool-vendors. A commonality is that these sources report experiences on an external expert guiding an organization while adopting feature modeling or platform engineering. So, while feature modeling is considered to be a simple and intuitive notation, the training seems to still require efforts. Such training is also key to define the purpose and scope of the feature model (explained shortly).

RO-P_4: Feature-Modeling Phases

While organizing our principles, we identified a set of eight intuitive phases for structuring feature-modeling processes.

6.2.5 RO-P$_5$: Principles of Feature Modeling

In this section, we detail our 34 feature-modeling principles, organized according to the phases we identified. A principle captures *what* should be done during feature modeling, or *how* it should be done. We discuss why each principle is relevant and how it is related to other principles, for which we use acronyms. Furthermore, we display identifiers to the publications (cf. Table 6.2) and interviews (cf. Table 6.3) from which we synthesized a principle in parentheses.

Planning and Preparation

Identifying stakeholders

PP$_1$: *Identify relevant stakeholders* (P$_1$). The most important stakeholders to start feature modeling are domain experts who have extensive knowledge (cf. Chapter 4) about the platform an organization wants to model. Such stakeholders can have diverse roles (e.g., architects, project managers, requirements engineers), depending on the organization's processes and domain. Identifying such stakeholders is a prerequisite for acquiring domain knowledge (IS$_1$) and conducting workshops (M$_1$). A second group of stakeholders involves the modelers who will construct and maintain the feature model. Typically, such modelers are software and system architects or product managers, whose usual work involves constructing abstract system models — and who can then be trained in feature modeling (T$_1$). Lastly, the users of the feature model are an important group of stakeholders. This group can involve various roles (e.g., developers, product managers, customers), depending on the purpose the feature model serves (PP$_3$). Defining the users of the feature model directly impacts model decomposition (PP$_4$), feature identification (M$_6$), and model views (M$_9$).

Unifying vocabulary

PP$_2$: *In immature or heterogeneous domains, unify the domain terminology* (P$_5$, P$_6$, P$_8$, P$_9$, P$_{18}$). To improve model comprehension and facilitate the construction process, it helps to define a unified terminology using descriptive terms (e.g., to name features). For instance, Lee et al. [2002] recommend to,

> *"in an immature or emerging domain, standardize domain terminology and build a domain dictionary. If not done, different perceptions of domain concepts could cause confusion among participants in modeling activities and lead to time-consuming discussions."*

Similarly, Boutkova [2011] state that

> *"the first step during the introduction of the feature-based variability modeling was the definition [...] [of] all relevant terms [...] [as a] precondition for the successful collective work."*

Defining purpose

PP$_3$: *Define the purpose of the feature model* (P$_{12}$, P$_{25}$, P$_{27}$, P$_{28}$, P$_{31}$). A feature model can serve mainly two purposes. First, a feature model may support the design and management of a platform, for example, by providing an explicit representation of the domain or as input for the platform scoping. Second, a feature model may support the actual platform development, for example, to coordinate feature teams or as input for configurators. As an example for combining both purposes, Lettner et al. [2015] report a case study in which three individual feature models exist and are linked to each other to model the problem (i.e., the domain), solution (i.e., the platform), and configuration (i.e., partial configurations) space. Hubaux et al. [2010] present another example from an open-source system that relies on a feature model for configuring only. However, Hubaux et al. note that it was unclear whether the model documented runtime or design-time variability. So, if different purposes shall be addressed, it seems better to rely on multiple, linked feature models or to define model views (M$_9$). Defining a distinct purpose for a feature model avoids costs, for instance, by reducing comprehension problems and discussions.

Defining decomposition criteria

PP$_4$: *Define criteria for feature to sub-feature decomposition* (P$_5$, P$_8$, P$_{12}$, I$_4$, I$_8$). It is not well-defined what the semantics of a feature-model hierarchy (i.e., parent-child relations) are besides configuration constraints [Czarnecki et al., 2006, 2012]. For example, the hierarchy may represent part-of relations, feature responsibilities, or a functional decomposition, with our sources (P$_8$, I$_4$, I$_8$) suggesting that the latter is most common (M$_6$). To construct a consistent feature model that focuses on a single purpose (PP$_3$), an organization should clearly define when a feature is split into sub-features. Hein et al. [2000] state that a good feature hierarchy is indicated by few cross-tree constraints (MO$_4$). Still, as confirmed by Hubaux et al. [2008], how to construct the actual feature-model hierarchy is usually far form obvious, and often requires prototyping.

Planning the process

PP$_5$: *Plan feature modeling as an iterative process* (P$_8$, P$_{26}$, P$_{28}$). Considering our previous insights on (re-)engineering processes for variant-rich systems (cf. Section 3.3, Section 6.1), it is not surprising that feature modeling should alternate between modeling and domain scoping. An iterative process (1) facilitates the safe construction of a feature model by employing smaller changes; and (2) gradually improves the modelers' expertise on feature modeling as well as the domain. For instance, Chavarriaga et al. [2015] report that an iterative process allows to

> *"(1) train the domain experts using simpler models, (2) practice with them how to introduce new features and constraints, and (3) define practices to review and debug the models continuously."*

Assigning modelers

PP$_6$: *Keep the number of modelers low* (P$_{24}$, P$_{26}$, P$_{29}$, I$_1$, I$_9$, I$_{10}$). Our data indicates that the number of stakeholders involved in the actual feature modeling should be low, potentially only a single modeler. In fact, in most cases only a few stakeholders have the overview domain-knowledge required to create the feature model. For example, I$_1$ explicitly mentioned that

> *"it's usually the individual subsystem leaders or their architects or the lead designers in their subsystems that can capture the feature models."*

Similarly, I$_9$ stated that only project managers and architects participate in the feature-modeling process.

Training

Familiarizing with platform engineering

T$_1$: *Familiarize with the basics of product-line engineering* (P$_{17}$, P$_{29}$, P$_{31}$, I$_1$). The stakeholders who participate in the feature modeling should be educated in basic concepts of platform engineering (e.g., variability mechanisms, variability modeling, configuring), and particularly the tools and notations used for modeling. Obtaining an intuition about the relations between daily programming concepts (e.g., classes, data types), feature types, and graphical representations in the modeling tools facilitates the construction process. For example, I$_1$ mentioned that it helps to explain that a Boolean feature corresponds to a single checkbox (e.g., in a configurator), whereas enum features correspond to multiple checkboxes.

Training on small system

T$_2$: *Select a small (sub-)system to be modeled for training* (P$_8$). According to Lee et al. [2002], the modelers should first work on a smaller example system. Moreover, that system should exhibit mainly commonality and no fixed deadline for releasing it into production. Thus, by allowing for arbitrary fast feedback (PP$_5$), such a methodology facilitates training activities and can lead to a faster acceptance.

Conducting a pilot

T$_3$: *Conduct a pilot project* (P$_{17}$, I$_1$). Ideally, feature-modeling training (after performing T$_1$ and T$_2$) is conducted through a guided pilot project over several days (e.g., I$_1$ recommended three-day workshops). For re-engineering projects in particular, I$_1$ reported that it is necessary to explain to stakeholders (typically developers) how to abstract differences between variants' implementation into domain concepts (i.e., features). Developers usually refer to specific implementation details, which can be abstracted by reasoning on the differences multiple times (M$_3$), leading to more and more abstract descriptions. I$_1$ mentioned this as a key experience:

> *"every time they say something you say 'why', and now kind of abstract up one level you go why, and you know, after that 3 or 4 whys they will probably get to the essential feature."*

Our data suggests that inspecting 20-50 configuration options, as well as identifying and modeling the corresponding features is a feasible target for a three-day workshop. The results of the pilot should be verified by configuring the obtained feature model (QA$_2$).

Information Sources

Eliciting information

IS$_1$: *Rely on domain knowledge and existing artifacts to construct the feature model* (P$_{1\text{–}5}$, P$_{15}$, P$_{18}$, P$_{22}$, P$_{26}$, P$_{28}$, P$_{29}$, I$_{1\text{–}3}$, I$_9$, I$_{10}$). From our data, we identified two information sources to support feature modeling that are directly connected to our findings in Chapter 4. First, domain experts (PP$_1$) can provide their knowledge about the domain, existing variants, and customer demands. Such information is typically elicited and documented during workshops (M$_1$). Second, existing artifacts (i.e., in re-engineering projects) can serve as information sources (cf. Section 4.3) to identify features and their dependencies. Such information is usually elicited by comparing commonalities and differences between variants (M$_2$, M$_3$). I$_1$ explains that the typical process for identifying and location features is similar to the ones we performed (cf. Section 3.3):

> *"we look at source code clones/branches/versions to get the product differences (e.g., by looking at ifdef variables), and identify and extract features from these differences manually."*

Model Organization

Limiting depth of the hierarchy

MO$_1$: *The depth of the feature-model hierarchy should not exceed eight levels* (P$_{14}$, P$_{20}$, P$_{24}$, P$_{30}$, I$_{1\text{–}5}$, I$_8$, I$_9$). Except for Nakanishi et al. [2018], no experience report mentions the depth of the feature-model hierarchy explicitly. However, surveys, open-source case studies, and most of our interviewees agree that the hierarchy usually has a depth between three and six levels. For instance, I$_4$ reported that

> *"at most I've seen three levels deep"*

and I$_1$ stated:

> *"We usually don't see them going more than 3 levels deep."*

Similarly, I$_9$ described that their feature-hierarchy

> *"is more spread than deep I would say; it is just the way this model was evolving."*

Even the Linux kernel reaches a maximum depth of eight levels [Berger et al., 2014b], highlighting that modelers avoid deep hierarchies to evade large sub-trees that challenge comprehension. One way to solve deep hierarchies it to split feature models into sub-models (MO$_3$),

Abstracting higher-level features

MO$_2$: *Features at higher levels in the hierarchy should be more abstract* (P$_1$, P$_3$, P$_8$, P$_9$, P$_{13}$, P$_{14}$, P$_{18}$, P$_{20}$, P$_{24}$, P$_{25}$, I$_8$). Our data shows that features that are higher in the feature-model hierarchy are also more visible to customers and represent more abstract domain concepts. In the middle levels, features typically represent a functional decomposition until they capture technical details (e.g., hardware, libraries) in the bottom level (PP$_4$).

Kang et al. [1998] distinguish between (highest to lowest) capability, operating-environment, domain-technology, and implementation-technique features, whereas Griss et al. [1998] follow the RSEB model [Griss, 1997], distinguishing architectural and implementation features.

Splitting large features

MO_3: *Split large models* (P_1, P_5, P_{12}, P_{13}, P_{15}, P_{18}, P_{19}, P_{22}, P_{25}, P_{26}, P_{28}, P_{31}, I_1, I_2, I_8) *and facilitate consistency with interface models* (P_5, P_{19}). Many of our sources agree that large feature models (i.e., thousands of features) should be split into smaller models.. For instance, in compliance with MO_2, features that represent user-visible concepts should be in a different model than those that represent implementation details. Otherwise, the increasing size of a feature model can become a problem, as Dhungana et al. [2010] report:

> *"our first brute-force approach was to put all model elements into one single model. This did not scale [...]. It also became apparent [...] that working with a single variability model is inadequate to support evolution in the multi-team development."*

Unfortunately, splitting feature models poses problems regarding consistency maintenance. One interesting recommendation to facilitate maintenance is to extract features that are related to inter-model dependencies into a separate feature model, a so-called interface feature model [Hein et al., 2000] or feature dependency diagram [Hofman et al., 2012]. Even if a single feature model is envisioned, it can be helpful to construct smaller ones for different stakeholders and merge these models later. Moreover, the smaller feature models can serve as prototypes that are iteratively refined (PP_5).

Avoiding cross-tree constraints

MO_4: *Avoid complex cross-tree constraints* (P_{11}, P_{19}, P_{23}, P_{24}, $I_{2–7}$). Modelers can use cross-tree constraints to define dependencies between sub-trees of a feature model. Still, complex constraints (i.e., arbitrary propositional formulas) challenge the comprehension, evolution, and maintenance of the feature model — and potentially the comprehensibility of configuration errors. For such reasons, almost all of our interviewees stated to avoid cross-tree constraints or rely only on simple ones, such as excludes, requires, and conflicts. If simplifications are not possible, an organization may capture complex constraints in the mapping between features and assets using presence conditions [Berger et al., 2014a]. One interesting practice reported by Hofman et al. [2012] is to highlight or tag features that are involved in cross-tree constraints to raise stakeholders' awareness.

Organizing feature groups

MO_5: *Maximize cohesion and minimize coupling with feature groups* (P_3, P_4, P_{14}, P_{26}, P_{28}, I_6). Our data implies that feature groups should involve related functionalities, while abstract features help structure the feature model. As a general principle, a high cohesion between features in a group and a low coupling to features of other groups (in terms of cross-tree constraints) imply suitable grouping. In this sense, Kang et al. [1998, 1999] state that high-level (in the feature hierarchy) alternative-groups imply limited reuse, while high-level and-groups and low-level or-groups imply high reuse.

Modeling

Conducting workshops

M_1: *Use workshops to extract domain knowledge* (P_8, P_{15}, I_1, I_3, I_9). Five of our sources report that workshops are the most efficient way to initiate feature modeling. For this purpose, stakeholders with detailed domain and system knowledge (PP_1) model different variants based on their knowledge. The workshops should then be used to identify features by discussing which differences between variants exist (M_2) for what reasons.

Focusing on differences first

M_2: *Focus first on identifying features that distinguish variants* (P_8, P_9, P_{15}, P_{18}, I_1, I_2). Six of our sources report that it is harder for most stakeholders to describe commonalities compared to features that are different between variants. For instance, Dhungana et al. [2010] explicitly sate:

> *"The variability of the system was therefore elicited in two ways: moderated workshops with engineers [...] to identify major differences and [...] automated tools to understand the technical variability at the level of components by parsing existing configuration files."*

Moreover, Lee et al. [2002] explain:

> *"Products in the same product line share a high level of commonality. Hence, the commonality space would be larger to work with than the difference space."*

Using bottom-up modeling

M$_3$: *Apply bottom-up modeling to identify differences between artifacts* (P$_8$, P$_{15}$, P$_{18}$, I$_{1\text{–}3}$). In the context of re-engineering projects, various artifacts (IS$_1$) can serve as information sources and should be analyzed for differences. Three of our interviewees stated that source code is typically analyzed first, allowing for automation through diff tools (P$_{15}$, I$_2$). Still, the identified differences are usually inspected manually, for instance, during workshops (M$_1$).

Using top-down modeling

M$_4$: *Apply top-down modeling to identify differences in the domain* (P$_5$, P$_{24}$, I$_2$). In contrast to bottom-up modeling, top-down modeling focuses on involving project managers, domain experts, and system requirements in workshops (M$_1$). I$_2$ explained:

> *"Top-down is successful with domain experts, more abstract features."*

So, features that are identified during top-down modeling are often common or abstract features of the variant-rich system (MO$_5$).

Combining strategies

M$_5$: *Use a combination of bottom-up and top-down modeling* (P$_{13}$, P$_{24}$, I$_2$). Since bottom-up and top-down modeling may yield different results (M$_2$), our sources highly recommend to employ combinations of both strategies (cf. Section 3.3, Section 4.3). Such a combination improves completeness and serves as a control mechanism by eliciting features from a high-level (top-down) and low-level (bottom-up) perspective. For example, Schwanninger et al. [2009] report that

> *"the feature model was built in a top-down and a bottom-up manner. [...] The user visible features became top level features, while internal features either ended up in the lower level of the feature model or in separate, more technical sub-domain feature models."*

Defining features

M$_6$: *A feature typically represents a distinctive, functional abstraction* (P$_5$, P$_8$, P$_9$, P$_{21}$, I$_1$, I$_3$, I$_6$, I$_7$, I$_9$). Even though some organizations use feature models for non-functional properties (e.g., performance, security), most sources agree that features represent functional concepts. Concretely, I$_3$ explained that a CPU type does not represent a feature (i.e., it is no user-visible functionality), and thus should be defined as a feature attribute instead. So, features should represent user-visible functionalities of the variant-rich system, typically abstracting a number of functional requirements.

Using spurious features

M$_7$: *If needed, introduce spurious features* (P$_{12}$). From the open-source case study of Hubaux et al. [2008], we identified the interesting idea that a

> *"spurious feature represents a set of features [...] that are actually not offered by current software assets, which is arguably paradoxical when modeling the provided software variability."*

As a concrete example, consider an application that falls back to a default language if the language selected by its user is not supported. A spurious feature could cover all languages that are not yet provided in the application. So, in contrast to most features, this feature would cover *non-existing* functionality.

Defining default values

M$_8$: *Define default feature values* (P$_{14}$, P$_{20}$, I$_8$). Three of our sources recommend to define default values for features if a feature model spans a very large configuration space. In such cases, configuring a variant is actually a reconfiguration problem. Interestingly, all three sources relied on tools that also allowed visibility conditions [Berger et al., 2013b], meaning that the default value can only be modified if that condition is satisfied.

Defining views

M$_9$: *Define feature-model views* (P$_{22}$, I$_8$). Some parts of a feature model may be irrelevant for certain stakeholders or tasks. Feature-model views allow to customize what a stakeholder sees, and thus help unclutter the visual representation. According to Manz et al. [2013],

> *"not all features and associated artifacts are relevant for an individual engineer [...] we realized user-specific views by development phase and abstraction specific feature models within a hierarchical feature model."*

The "abstraction specific feature models" are partial configurations of a model that another model can refer to. Similarly, I$_1$ reported on profiles, which use the same technique to expose a set of features in one model to a different model.

Using Boolean features

M$_{10}$: *Prefer Boolean type features for comprehension* (P$_{14}$, P$_{20}$, P$_{26}$, P$_{27}$, I$_2$, I$_{5–7}$, I$_9$). Most of our interviewees agreed that features are primarily of the Boolean type. For instance, I$_7$ stated the

> *"nice way of organizing configuration switches"*

with Boolean features as the primary strength of feature models. As a notable exception, I$_8$ reported mostly on non-Boolean features, which was caused by integrating application-logic specifications into the feature model. So, some domains, for example, operations systems, have a higher number of non-Boolean features [Berger et al., 2010, 2013b].

Documenting

M$_{11}$: *Document the features and the obtained feature model* (P$_1$, P$_{12}$, P$_{25}$, I$_{8–10}$). Even though a feature model represents documentation on its own, several sources stressed that the model must also be documented. For example, I$_8$ reported:

> *"we've put a lot of effort into extensively documenting all options and immediately document in the editor."*

Moreover, any new terms relating to the feature model (i.e., features, constraints) should be unified and documented (PP$_2$, PP$_5$) In contrast, I$_{10}$ reported that organizations often use the implementation of variants as documentation, which challenges re-engineering projects.

Dependencies

Considering experts

D$_1$: *If the models are configured by (company) experts, avoid feature-dependency modeling* (P$_{24}$, I$_1$, I$_2$, I$_{5–8}$). Most of our interviewees agreed that identifying cross-tree constraints is expensive and can complicate the maintenance of a feature model. Additionally, our interviewees agreed that modelers who configure the feature model usually know undeclared dependencies, for instance, I$_6$ stated:

> *"There were some cross-tree dependencies [...], but they weren't in the model (the one configuring the model needed to know them)."*

So, feature models for internal use typically exhibit few, simple cross-tree constraints (MO$_4$), as confirmed by I$_1$:

> *"Very few cross-tree constraints [...] typically requires and conflicts."*

Interestingly, I$_3$ estimated that circa 50 % of all features would be involved in explicit dependencies — highlighting the investments (cf. Chapter 3) that would be needed to document and trace such knowledge (cf. Chapter 4, Chapter 5). In practice, missing explicit dependencies seem to be replaced by domain-specific semantics, for instance, new features may be recommended by others. Note that our sources for this principle reflect on small- to medium-sized feature models (tens to hundreds of features) or small configuration spaces (tens of variants).

Considering users

D$_2$: *If the main users of a feature model are end-users, perform feature-dependency modeling* (P$_{10}$, P$_{14}$, P$_{28}$, I$_4$). Depending on the purpose of a feature model (PP$_3$), we found recommendations that oppose the previous principle (D$_1$). Namely, if a feature model is configured by end-users (e.g., customers) or exhibits thousands of features, our sources recommend to explicitly model cross-tree constraints. Thus, an organization can ensure that defined configurations are valid by relying on choice propagation and conflict resolution [Benavides et al., 2010; Berger et al., 2020; Thüm et al., 2018].

Quality Assurance

Validating feature models

QA$_1$: *Validate the obtained feature model in workshops with domain experts* (P$_8$, P$_{26}$, P$_{28}$, I$_1$). As for the construction (M$_1$), several sources recommend to conduct workshops with domain experts to review the feature model. Lettner et al. [2015] suggest to involve domain experts with different roles, since

> *"they can focus on their area of expertise, i.e., product management, architecture, or product configuration aspects. Our results further show that detailed domain expertise is required for defining the feature models […]."*

During the workshops, I$_1$ emphasized to discuss

> *"what are the right names for the features, what are the right ways of structuring the features, try the process of first creating the new product [configuration] that never existed before […]."*

Finally, Lee et al. [2002] recommend to involve domain experts who did not participate in the construction process to validate whether the feature model can be used intuitively.

Deriving configurations

QA$_2$: *Use the obtained feature model to derive configurations* (P$_1$, P$_{3–5}$, I$_1$, I$_8$). Expectedly, a feature model should enable an organization to define configurations for existing variants. Moreover, our data indicates that deriving configurations for new variants and verifying whether that variant is meaningful, serves as a good indicator that the feature model reflects its domain. If it is not possible to derive any novel configurations, re-engineering a platform may not be a feasible investment (cf. Section 3.3). Finally, I$_8$ revealed that they

> *"[…] have a workshop with customer[s] where we discuss how things need to be configured in detail to adhere to their domains."*

Using regression testing

QA$_3$: *Use regression tests to ensure that changes to the feature model preserve previous configurations* (I$_9$, I$_{10}$). Two of our interviewees stated that they relied on regression testing to ensure that updates of the feature model would preserve existing configurations. I$_9$ detailed that they define and test (during continuous integration) reference variants from the model that involve different feature combinations used in their real-world systems. I$_{10}$ emphasized the need for employing regression testing on feature-model changes, stating that they do

> *"testing, a lot of testing."*

Model Maintenance and Evolution

Using centralized governance

MME$_1$: *Use centralized feature model governance* (P$_{13}$, P$_{14}$, P$_{17}$, P$_{18}$, P$_{24}$, P$_{26}$, P$_{31}$, I$_1$, I$_4$, I$_5$). According to several sources, having a specific employee (or team) who governs the feature model ensures consistent evolution and enables strict access control. Namely, I$_1$ reported that

> *"somebody [...] chief architect or whoever [...] becomes the lead product-line engineer, okay, so they really own the overall feature model."*

Such a specific role seems to be particularly used in closed environments in which an organization has full control over its feature model, and less in community-driven or open-source projects. If multiple feature models exist, our sources recommend that each has an individual maintainer or team.

Versioning the feature model

MME$_2$: *Version the feature model in its entirety* (P$_{20}$, I$_4$, I$_8$, I$_9$). Four of our sources recommend to version the feature model in its entirety. Versioning each feature individually would lead to inconsistencies and conflicts that require manual resolution, for example, because different change operations refactor the same features [Kuiter et al., 2019, 2021]. I$_8$ also stated that some features that become obsolete in a certain version of the model may be preserved to ensure backwards compatibility, and are usually marked as deprecated.

Defining new features

MME$_3$: *New features should be defined and approved by domain experts* (P$_{26}$, I$_5$, I$_8$, I$_{10}$). Before adding a new feature to a feature model, an organization has to specify the feature, define its testing strategy, and reason on its impact on the existing platform (i.e., addressing some feature facets we described in Section 4.3). Consequently, relevant domain experts should approve new features. For instance, I$_5$ described that they

> *"[...] have to make sure that the work is done properly. Because [...] you can be stopped by simple technical issues, like we cannot merge back, because the branch is frozen."*

RO-P$_5$: Principles of Feature-Modeling

Based on our empirical data, we synthesized a set of 34 interrelated feature-modeling principles. These principles provide an understanding of how to construct feature models; and help employ our other findings in this dissertation, due to the direct relations.

6.2.6 RO-P$_9$: Properties of Feature-Modeling Principles

Consensus in data

As we can see in Table 6.2, most of the publications we studied report on re-engineering experiences. Consequently, most of our feature-modeling principles also directly relates to platform re-engineering, which we highlighted in the previous section and connected to our other findings in this dissertation. Moreover, most of our sources agreed on how to elicit features (PP$_2$, IS$_1$, M$_2$, M$_3$), the semantics of features (M$_6$, M$_{10}$), and what properties a feature model should exhibit(MO$_1$–MO$_4$).

Context-dependent principles

Some principles are relevant only in a certain context. For instance, the decision on whether to explicitly model dependencies (D$_1$, D$_2$) depends directly on who configures the feature model. Similarly, some principles propose alternative solutions to the same problem. For example, large feature models can be split up (MO$_3$) or organized by using feature-model views (M$_9$) — with both principles aiming to achieve a separation of concerns depending on the involved stakeholder. Note that MO$_3$ is based on 15 sources, while we derived M$_9$ only from two. However, a reason for this may be the missing tool support for feature-model views — we are aware of only two tools by Acher et al. [2013] and Hubaux et al. [2011].

Pros and cons

Employing any principle has its own pros and cons. For example, defining feature-model views (M$_9$) facilitates centralized governance (MME$_1$) and does not require interfaces between different models. Alternatively, multiple models could be used, with each model likely having a flatter hierarchy (MO$_1$) and less complex cross-tree constraints (MO$_4$). While this would facilitate maintenance, multiple models do not promote centralized governance. Even though our data implies different trade-offs between principles, it requires future work to elicit actual data on the pros and cons of each principle.

Notations and domains

Our systematic literature review revealed over ten different feature-modeling notations, which have been employed in 14 distinct domains. Despite this variety, we noted that few notations were used regularly, namely:

- FODA/FORM [Kang et al., 1990, 1998], which has been proposed as the first notation for feature modeling; and
- pure::variants [Beuche, 2012], which is an established tool for feature modeling in industry and builds upon FODA/FORM.

Similarly, we can see two main domains, automotive and industrial automation. Nonetheless, none of our feature-modeling principles enforces a specific notation or domain — achieving our goal of deriving general-purpose principles that are relevant for any domain.

RO-P$_6$: Properties of Feature-Modeling Principles

The properties of our feature-modeling principles indicate that they are useful for constructing feature models in any notation and domain. However, their pros, cons, and relations must be studied in future work to reason on their impact on practice.

6.2.7 Threats to Validity

Threats to validity

In this section, we discuss potential threats to the validity of our feature-modeling principles. We remark again that our principles are based on those considered important by researchers and practitioners, and thus have been reported. Since this results in different levels of granularity, some are generalizable beyond feature modeling or seem rather trivial.

Construct Validity

Misunderstandings during interviews

We started every interview with an introduction during which we aimed to understand the interviewee's context. Following recommendations for semi-structured interviews [Hove and Anda, 2005; Seaman, 1999; Shull et al., 2008], we allowed each interviewee to thoroughly explain their domain and its terminology. Based on these explanations, we adapted our questions to avoid misunderstandings and ensure that we could correctly interpret our data. Still, there may have been misunderstandings that we could not prevent.

Internal Validity

Completeness of feature-modeling principles

We cannot guarantee that our feature-modeling principles are best practices or cover all relevant experiences. To mitigate these threats, we conducted a manual search of relevant publication venues, verified the set of publications with an automated search, and employed backwards snowballing. Furthermore, we interviewed practitioners from various domains and with considerable experiences on feature modeling. Using these data sources, we aimed to ensure that our feature-modeling principles reflect domain-independent practices that are valuable for constructing a feature model for any variant-rich system — even though they may not be complete.

External Validity

Transferring principles to other domains

We aimed to ensure that our feature-modeling principles can be transferred to other domains, processes, and organizations. For this purpose, we built upon research publications, technical reports, and interviews to cover a diverse range of feature-modeling experiences. Still, some of our principles are only relevant in certain contexts, which we describe explicitly.

Conclusion Validity

Interpretation of data

We may have misinterpreted some of our data, which threatens the principles we derived from that data. However, we elicited qualitative data from publications and interviews with practitioners in a systematic and replicable process. To mitigate the threat of misinterpreting data, we analyzed it with four researchers, confirmed principles in different sources, refined our results until we achieved consensus, and asked external practitioners to assess our principles. For these reasons, we claim that our principles are reasonable and practically important, even though we may have misinterpreted individual data points.

6.3 Maturity Assessment for Variant-Rich Systems

Maturity assessments

As we explained especially in Chapter 3, adopting platform engineering is a strategical decision that requires considerable investments and impacts all processes. However, we also found that an organization can benefit from incremental steps towards a platform, for instance, by implementing more systematic clone management [Berger et al., 2020; Krüger and Berger, 2020b; Pfofe et al., 2016; Rubin and Chechik, 2013a; Rubin et al., 2012, 2013]. So, particularly for re-engineering projects, an organization should first (e.g., based on a domain analysis and feature modeling) assess the current status (i.e., maturity) of its variant-rich systems, based on which it can define goals and evaluate their economical impact. Besides cost models (cf. Section 3.1.1), several researchers have proposed techniques to scope, plan, decide on, or assess platform engineering [Ahmed and Capretz, 2010a,b; Ahmed et al., 2007; Bayer et al., 1999; Bosch, 2002; Frakes and Kang, 2005; Frakes and Terry, 1996; Kalender et al., 2013; Koziolek et al., 2016; Niemelä et al., 2004; Rincón et al., 2018, 2019; Schmid et al., 2005; Tüzün et al., 2015]. Of such techniques, the family evaluation framework [van der Linden, 2005; van der Linden et al., 2004, 2007] represents arguably the most flexible, light-weight, and well-known framework for assessing the maturity of any variant-rich system, independently of the underlying processes. Still, we are not aware of reports on how to operationalize the family evaluation framework for larger platforms and in the context of modern software-engineering practices.

Section contributions

In the following, we [Lindohf et al., 2021] tackle the missing experiences on using the family evaluation framework by reporting an action-research-based [Davison et al., 2004; Staron, 2020] multi-case study [Bass et al., 2018; Leonard-Barton, 1990; Runeson et al., 2012] in which we used the family evaluation framework to assess the maturity of nine platforms at a large organization. More precisely, we defined the following three sub-objectives of **RO-P**:

RO-P$_7$ *Elicit how the family evaluation framework can be operationalized.*

We evaluated the maturity of nine platforms, for which we had to operationalize the abstract definitions of the family evaluation framework in practice. Concretely, our core contribution involves descriptions of how to adapt the framework to an organization's domain, how to elicit as well as analyze information, and how to derive actionable items from the assessment. The results of this sub-objective help practitioners employ the family evaluation framework in their own organization, and researchers in designing or adapting maturity assessments.

RO-P$_8$ *Identify challenges of employing the family evaluation framework.*

Building on our results and feedback from our collaborators, we identified challenges and potential pitfalls that may occur when using the family evaluation framework. Our results highlight several problems that may affect the assessment, and are partly connected to our other research objectives. These insights help organizations prevent problems of using the family evaluation framework and indicate new research directions.

RO-P$_9$ *Collect the benefits of employing the family evaluation framework.*

Finally, we collected a set of major benefits the organization experienced solely from operationalizing the family evaluation framework (i.e., without actually implementing changes). These experiences highlight that the family evaluation framework (and the contributions in this dissertation) can have immediate value for an organization. So, our insights help other organizations to decide whether to employ the family evaluation framework and adopt platform engineering.

The artifacts of our study are almost completely available (i.e., all interview questions without those violating confidentiality) in our open-access article [Lindohf et al., 2021]. In Section 6.3.1, we present the methodology we employed to operationalize the family evaluation framework. Then, we summarize potential threats to the validity of our study in Section 6.3.5, before we address our sub-objectives in Section 6.3.2, Section 6.3.3, and Section 6.3.4, respectively.

6.3.1 Eliciting Data with a Multi-Case Study

Methodology

In the following, we first introduce the family evaluation framework and our subject organization, before we report our actual study design.

The Family Evaluation Framework

Family evaluation framework

As far as we know, the family evaluation framework represents the most general and comprehensive assessment framework for platform engineering, which can be used for planning and monitoring the (re-)engineering of variant-rich systems. The family evaluation framework has been developed in the context of large EU ITEA projects in collaboration with industrial partners [Pohl et al., 2005; van der Linden et al., 2004], and aims at assessing the maturity of platform engineering at an organization — but it is actually applicable to any variant-rich system. To structure the assessment, the family evaluation framework builds on the BAPO concerns (cf. Section 2.3): business, architecture, process, and organization.

Framework structure

We display an overview of the family evaluation framework in Figure 6.5. As we can see, the dimensions of the family evaluation framework span the BAPO concerns, and are further separated into three to four aspects. Each aspect is assessed based on five levels that reflect the maturity of a variant-rich system with respect to that aspect. The family evaluation framework lists requirements for achieving each level in each aspect. Note that the levels are not directly connected, which means that a platform may achieve different levels in each dimension. Nonetheless, there is an indirect connections, since achieving a higher level in one dimension may require progress in another (i.e., the BAPO principle).

Resulting assessments

The outcome of an assessment with the family evaluation framework is a profile that specifies the level a platform achieved in each dimension. As an example, consider the architecture dimension, which involves three aspects: the degree of asset reuse, the platform's reference architecture, and the extend of variability management. For each level, the family evaluation framework specifies the following requirements:

1. Independent development: Each variant is developed from scratch as an individual system, without reusing any existing artifacts.
2. Standardized infrastructure: The architecture incorporates third-party software and the consequent variability.
3. Software platform: Common features are implemented as a configurable platform, allowing to combine their assets — but without configuration support.

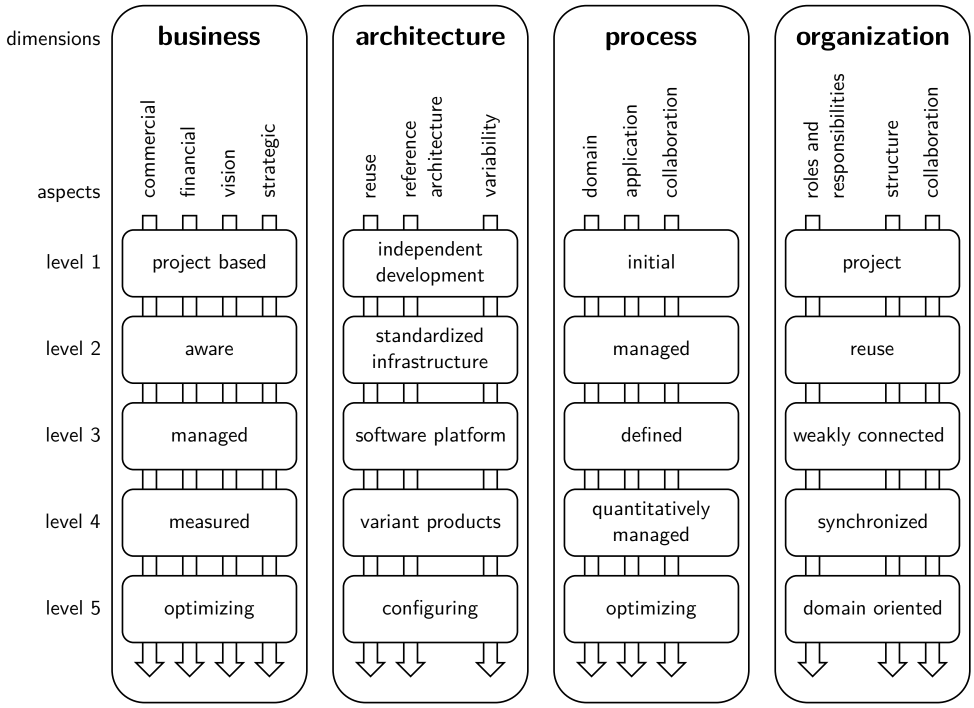


Figure 6.5: The family evaluation framework based on van der Linden et al. [2007].

4. Variant products: A full reference architecture for the platform specifies variability, defines configurations, and systematically manages asset reuse.
5. Configuring: The platform allows to configure and derive variants automatically, with minimized divergence between domain and application engineering.

Even though these levels help to assess the maturity of a platform, they are limited. First, the distinctions between levels are vague, which challenges a precise mapping during an assessment (e.g., clone & own does not fit precisely into any level of the architecture dimension). Second, the family evaluation framework does not specify how to elicit the information required for assessing a level. Finally, and most importantly, an organization may not desire to achieve a higher level in the family evaluation framework if its current practices work well, which is why the resulting profile must be carefully interpreted

The Subject Organization

Saab Aeronautics Simulation Center

For our cases, we considered platforms of the Saab Aeronautics Simulation Center (hereafter: Simulation Center). The Simulation Center is an in-house software developer for various types of aircraft simulators, such as prototype, system, or training simulators. These simulators are primarily delivered to the parent organization, Saab AB, which has around 16,000 employees (the Simulation Center has around 300). In fact, only four platforms are intended for external customers, while another fifteen are delivered to internal "customers" only. Half of the internal platforms comprise software only, while all other platforms involve various types of artifacts, such as electronics, mechanics, and software.

Platform engineering at the Simulation Center

The Simulation Center has worked on adopting platform engineering for several years. In fact, the Simulation Center adopted basic platform-engineering principles as early as 2010 [Andersson, 2012] for a platform with 1.4 million lines of code in 275 modules of various programming languages (e.g., Ada, C, C++, Fortran 77) — which is one of the early cases of industry adopting systematic platform engineering. After this case, the Simulation Center

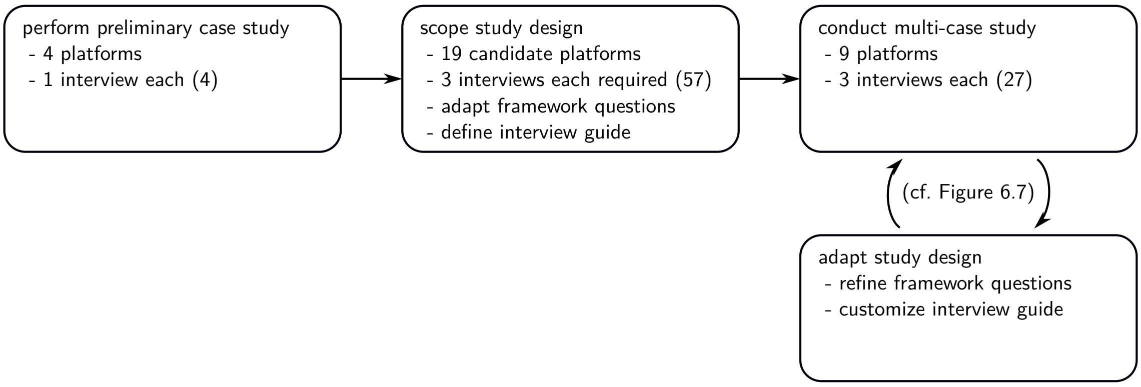


Figure 6.6: Overview of our methodology for using the family evaluation framework.

found that platform engineering based on the practices defined in the software product-line community considerably facilitated its software development. As a consequence, all systems should be re-engineered into platforms if the investments could be justified (cf. Chapter 3). Among others, the Simulation Center aimed to:

- Establish a common terminology between stakeholders to refer to domain concepts.
- Unify its processes for platform engineering.
- Create fully reusable components.
- Achieve faster time-to-market, while reducing development costs.
- Improve the software quality and ideally fix each bug only once.

Considering these goals, the Simulation Center used the family evaluation framework to assess whether it was heading in the right direction, to define concrete actions for improving, and to codify a method to systematically assess the maturity of its platforms.

Drivers of variability

The platforms of the Simulation Center inherit the high degree of variability of the actual aircraft software, and incorporate additional simulation-specific features. Consequently, these platforms exhibit more features, and thus variability, than the actual aircrafts. Moreover, the Simulation Center faces additional requirements, for instance, due to export control licenses, strict need-to-know policies, or secrecy issues. The requirements resulting from such restrictions lead to features that are unrelated to the aforementioned drivers or customers. Instead, it may simply be necessary that the Simulation Center develops the same feature twice: once with classified code, and once with open code.

Study Design

Preliminary case study

We summarize the overall design of our study in Figure 6.6. In the beginning, we performed a preliminary case study with three students, who conducted four interviews on four different platforms to understand the questions of the family evaluation framework and the domain of the Simulation Center. The assessments we delivered on the platforms during a presentation were considered valuable by the respective stakeholders and managers. In addition, we found that the questions defined in the family evaluation framework needed adaptations to fit the domain and tackle actual problems in practice. We also experienced that it was helpful to involve different stakeholders in the interviews, since they can provide complementary insights on a platform. Finally, we agreed that adapting the questions and conducting a larger study was necessary to tackle our sub-objectives and derive actionable items for the Simulation Center.

Multi-case study

Building on our insights, we designed a multi-case study that was based on action research. So, we combined multiple cases to elicit data from different sources, which allowed us to generalize our results — improving the internal and external validity [Siegmund et al., 2015]. To this end, we used the family evaluation framework to assess nine platforms, each exhibiting different properties and stakeholders. We elicited qualitative data by conducting interviews with three stakeholders of each platform (i.e., a manager, an engineer, a technical lead). For this purpose, we designed a structured interview guide, which a principle engineer at the Simulation Center used to conduct the interviews. All interviews were transcribed into a table, from which we synthesized our results.

Interview guide

We adapted the questions in our interview guide to handle the varying properties of our subject platforms (e.g., pure software compared to software and hardware). For example, we added questions that were specific about each platform, and removed questions if they were irrelevant for a certain stakeholder role (e.g., our guide involved some questions only for managers). Consequently, we adapted our methodology based on the problems and demands at the Simulation Center, as proposed for action research. We argue that our adaptations are more feasible to understand how the family evaluation framework can be applied in practice, instead of strictly following the questions defined almost two decades ago. To track our adaptations, we documented for each question the corresponding aspect in the family evaluation framework, relevant stakeholders, and whether it was organization- or platform-specific.

Subject Platforms

Subject platforms

We show an overview of our nine subject platforms (out of 19 at the Simulation Center) in Table 6.4. As we can see, the platforms (several with over 1 million lines of code) involved various application domains (e.g., components, simulators) and team sizes. Moreover, we can specify different types of platforms:

- AS, TacS, OS, and VS comprise only software.
- CS and IO involve software and hardware components.
- DS and TS include software, hardware, and variants integrated from other platforms, essentially representing multi-product lines [Holl et al., 2012].
- DP is composed of assets and variants for documentation.

Interestingly, the platforms' origins vary greatly: Some emerged over decades through system evolution, while newer ones are more clearly defined. Still, the outer boundaries of each platform are clear. Related to our previous findings (cf. Section 6.1), the degree of separation between domain and application engineering varies drastically among different platforms. While the customer-specific features of evolved variants are clear, the newer platforms have not been scoped systematically. Additionally, features are usually defined in manifest-like files, while the way how features are defined is different between platforms. Aligning to our previous findings (cf. Section 6.2), feature dependencies are never formally defined, but documented in natural-language documentation or kept in developers' memory. Since the platforms rely on manifest files (each involving 30 to 300 assets), features are basically components — but the high-level notion of features is often not established.

Interviews

After our preliminary study, we conducted another 27 interviews over a period of 11 months. Approximately, we invested 100 hours in preparations, 150 hours in conducting the interviews (interviewer and interviewees), 180 hours in reporting, and 250 hours in the training of all participants. Initially, we considered to assess all 19 platforms, for which we estimated that 57 interviews would be required. However, there are several legacy platforms among these

Table 6.4: Overview of our subject platforms for using the family evaluation framework.

id	domain	team size
DS	development simulators	80
AS	aircraft simulation	60
TS	training simulators	30
CS	computer systems	20
IO	in/out systems	20
TacS	tactical simulation	20
DP	documentation and publications	10
OS	operating station	10
VS	visualization	5

19 (e.g., legacy versions of those in Table 6.4) that the Simulation Center plans to merge into the current versions. For other platforms, the Simulation Center could not justify the investments, since these will soon reach their end-of-service. While such platforms will not be assessed, the results for the other platforms still helped working with those. For instance, the additional knowledge obtained through our training is used in the whole Simulation Center.

Roles and Responsibilities

Involved stakeholders

Several stakeholders of the Simulation Center were involved in our study:

Interviewer: The interviewer led the assessments with the family evaluation framework. Consequently, the interviewer conducted the interviews, wrote internal reports, managed the platform training, and communicated with the platform managers.

Interviewee, manager: Platform managers are responsible for multiple platforms, which is why we may have interviewed some of them multiple times. These managers guide the long-term development, implement processes, and coordinate between projects. Each manager invested around 2.5 hours into each interview, and 1 more hour into preparing as well as prioritizing other interviews.

Interviewee, technical lead: Technical leads guide the technical development of a platform within a specific scope, typically a number of projects. So, these leads are experts regarding the platform architectures and the processes employed. Each technical lead invested around 2.5 hours into their interview.

Interviewee, engineer: Engineers — primarily software engineers — develop the platform, which is why they have knowledge regarding the daily work, implementation specifics, and intended designs. Each engineer invested around 2.5 hours into their interview.

Project management: While not directly involved in the assessments, the project management received the actionable items that originated from it. By prioritizing these items, the project management defined a road-map for every platform.

Platfrom management: The platform management involves all platform managers of the Simulation Center. They are responsible for the strategical planning of the platform initiative and initiated the assessments. Thus, besides their interviews, the platform managers invested also into platform training (four-hour workshop) as well as internal decisions and coordination.

In the remainder of this section, we refer to exactly these roles, especially regarding the interviewer and interviewees.

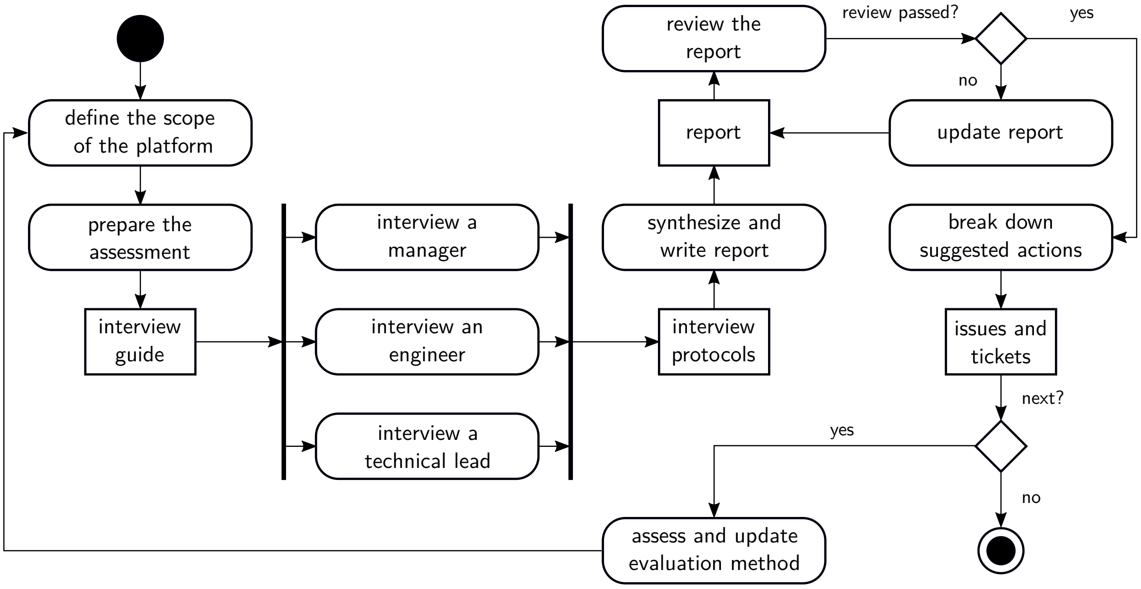


Figure 6.7: Overview of our assessment methodology.

6.3.2 RO-P$_7$: Adapting and Using the Family Evaluation Framework

Assessment method

From our overall methodology (cf. Figure 6.6), we extracted the key steps for using the family evaluation framework. We illustrate these steps in Figure 6.7, ranging from preparations to the definition of action items. In the following, we describe the individual steps of this methodology to provide a detailed understanding of how the family evaluation framework can be used in practice.

Defining the Scope of each Platform

Scoping platforms

For some of the platforms we assessed, it was unclear what their actual scope was. To resolve this issue, the manager of each platform filled in a template, in which we asked for (1) the **name** of the platform; (2) a first definition of the platform's **scope**; (3) the **purpose** of the platform (e.g., faster time-to-market); (4) a list of **customers**; (5) a list of **existing variants**; (6) a list of **features** or a feature model; (7) a description of the **organization** around the platform (e.g., platform teams); (8) an abstract description of the platform **architecture** (e.g., variability mechanism); and (9) a description of the **processes** for developing the platform. Based on the first entries, we obtained an overview understanding of each platform. The last three entries align explicitly to the BAPO concerns, except for the business concern (which we incorporated into customers and processes).

Agreeing on the scopes

Each manager presented the completed template to all other platform managers to receive feedback and disseminate the results. For this purpose, we used a workshop that was moderated by the interviewer and allocated 15 minutes for each presentation. All feedback after the actual presentation was documented by the presenter and led to updates in the templates. Through this workshop, we identified that some platforms were well-established, while the managers discussed about others. Most commonly, they did not agree on the scope of a platform, for instance:

- Some platforms seemed to represent two individual platforms with shared assets. The final choice on this issue was usually based on organizational or business concerns.
- Some platforms covered other platforms, implying that the scope was too broad. This issue occurred mainly for platforms that integrated variants of an external platform

into their own variants. While the developers should only integrate that external variant, they often felt responsible for that one and added it to their platform.

The information we elicited with the templates and during the workshop improved the general understanding and agreement on the scope of each platform, and guided us during the adaptation of our interview questions.

Preparing the Interviews

Eliciting additional information

The interviewer elicited additional information on each platform to further prepare the interviews. By reading, for instance, system descriptions, requirements, process definitions, or informal guides, the interviewer collected the knowledge needed to guide interviewees and prevent misunderstandings. For example, the concepts of domain and application engineering were well-known, but some interviewees did not know these terms. Using their additional knowledge, the interviewer could map the terminology at the Simulation Center to the research terminology.

Defining questions

Afterwards, we compiled a set of 67 questions for our interviews (47 for managers, 42 for technical leads, 38 for engineers). So, while we used the same questions across the platforms, we adapted them according to our interviewees' roles and their respective knowledge. To define our questions, we translated the examples by van der Linden et al. [2007] into the terminology of the Simulation Center (e.g., a "software asset" became a "configuration item"). Adapting the questions according to the organization's terminology is an investment, but we believe it pays off for two reasons:

1. The interviewer must improve their knowledge on the family evaluation framework and platform, since adapting a question means translating the abstract concepts into organization-specific instances. Missing such knowledge or using a generic template would impair the assessment, and potentially yield useless results.
2. The interviewees get questions in a familiar language and terminology, which allows them to focus solely on answering the questions.

For the Simulation Center, this investment was distributed among all platforms and over time, since the assessments shall be repeated in the future. Despite such benefits, adapting the questions has also drawbacks: The interviewer must acquire the required knowledge and is the only one who can properly synthesize the results, which may bias the findings.

Conducting the Interviews

Interviewees' roles

For each platform, we picked three interviewees: the closest platform manager, the highest technical lead, and a domain engineer. By involving experts with different roles, we aimed to complement our data on each dimension of the family evaluation framework. Actually, we experienced that the differences between interviewees' answers to the same question can be highly valuable. For instance, some managers pointed to specified processes, which the other two interviewees were unaware of. So, involving interviewees with different roles helped us identify the problem that a process may be defined, but is not communicated.

Prioritizing interviews

While we considered to perform all 57 interviews required to cover the 19 platforms of the Simulation Center, we decided that the investments would be too high. Even though we skipped platforms that will be merged, we still had to prioritize the remaining platforms. Since we had not previous maturity assessments, we considered mainly (1) the number of existing and planned variants (i.e., few variants do not require a platform); (2) the level of development (i.e., platforms with little evolution yield smaller benefits); and (3) the number of stakeholders working on the platform (i.e., more stakeholders are affected by changes). For instance, platforms with three existing variants, no planned variants, few stakeholders, and only bug-fixing activities received a low priority.

Interview conduct

We started each interview with a 30-minutes introduction into the basic concepts of platform engineering as well as the BAPO concerns and their levels in the family evaluation framework. To elicit reliable information, we explained that the levels do not represent grades, and that higher levels may not be suitable for a specific platform. Additionally, it helped that we adapted the questions, since they were not directly connected to the family evaluation framework anymore — which avoided that interviewees would worry about the actual assessment. We initiated discussions with the interviewees by asking our role-specific questions based on 11 categories, which helped structure the interviews and improved the comprehensibility (see the appendix of our open-access article [Lindohf et al., 2021] for the detailed mapping). Each interview required 1.5 to 2.5 hours.

Reports

We decided early on to write an extensive results report for each platform. The motivation for these reports was that the actions we derived from the assessment would likely initiate long-term processes. However, the Simulation Center cannot guarantee that the same employees will be available during the whole life-span of a platform. Initially, we planned to record and transcribe the interviews, but some interviewees felt uncomfortable with recordings. For this reason, the interviewer took notes that were visible to the interviewee and could be commented on. In the end, our notes comprised the interviewee's final answer to every question, additional comments, and further insights on the assessment. These notes served as input for our analysis and reports.

Analyzing the Interviews

Assessing levels

We relied on our notes, the interviewer's memory, and the additional documents we inspected to assess the maturity level of a all nine platforms in each BAPO dimension. Again, we found that it helped to involve different stakeholders. While a single answer was rarely precise enough to assess the levels, synthesizing from all answers as well as the discussions usually yielded a clear picture. If we had problems understanding an answer, we contacted the interviewee and asked for clarifications. Note that we did not try to achieve consensus between interviewees with conflicting answers — finding discrepancies was typically key to assess a platform's maturity.

Example of analysis

As an example, consider the question: *How is the connection between a variant and its assets managed? Do you ever write the names of the variant on the assets itself?* This question is vital in the context of systematic platform engineering: If assets are directly mapped to a specific variant, the platform does actually not enable systematic asset reuse. For one platform, we received the following answers:

> *"The environment where the asset is allowed to be installed in is written on the asset itself, this in turn makes a weak connection to the available variants (as a variant can only exist in one environment usually). Very few assets are in other ways connected to a specific variant."*
>
> manager

> *"We never write on the asset where it is to be used. However, a single product variant is often the driver for the development of an asset or feature."*
>
> engineer

> *"The strategy is to never write on the asset where it will be used. But there are exceptions."*
>
> technical lead

We can see that these answers indicate no clear consensus. While all interviewees agreed that writing variant names on assets is a bad practice, it seems unclear to what degree this was done in this platform.

Example assessment

This question was concerned with the architecture dimension. Based on all 14 questions in that dimension, we assessed the platform in the above example to be on level 2, "standardized infrastructure" (we display this example assessment in Figure 6.8), because:

- The platform involved techniques fore reusing assets, mostly based on late binding (e.g., parsing a configuration file at runtime). While there was no general reuse strategy, the employed ad hoc reuse worked well in practice.
- We identified three distinct architectures for this platform, posing the question whether its scope was too broad.
- Variability was managed within each asset, which is why they were more strongly related to application engineering and not the platform.

To achieve level 3, we suggested to start with the following actions:

- Define the architecture in more detail and understand why some stakeholders think three individual architectures exist.
- Investigate whether the interconnection of assets can be standardized.
- Investigate whether the variability management can be standardized, since it is scattered across source code, build processes, and configuration files, among others.
- Analyze whether the notion of features can be established instead of writing variant names on assets.

We can see that it may be problematic to align the answers of different stakeholders and assign them to a specific level.

Synthesizing Results and Reporting

Writing reports

Based on our interview results, we wrote reports for all platforms. We structured the reports according to the BAPO concerns, starting each of the resulting sections with a short summary of the interview answers. Then, we described the assessment according to the family evaluation framework and proposed actionable items for advancing to the next level. We evaluated these reports during a meeting with all interviewees. Instead of directly editing the reports, we documented the feedback in an additional document. The idea was to keep the reports as they were, since they described the interviewer's perception of each platform. We identified no major errors in the reports, and most comments were supplementary information. Interestingly, some evaluations led to discussions between our interviewees, which highlights different views on a platform's properties. We concluded each report with a diagram that indicated the level a platform achieved in each domain (cf. Figure 6.8)

Interviews after report

After publishing the reports internally in the Simulation Center, we received feedback from the readers. All of them agreed with our reports, and sometimes added further remarks and information on a platform. Moreover, we conducted two more interviews with stakeholders who requested to meet and anwer our questions. Based on their answers, we wrote an addendum to the original reports, which we sent only to the managers of the Simulation Center.

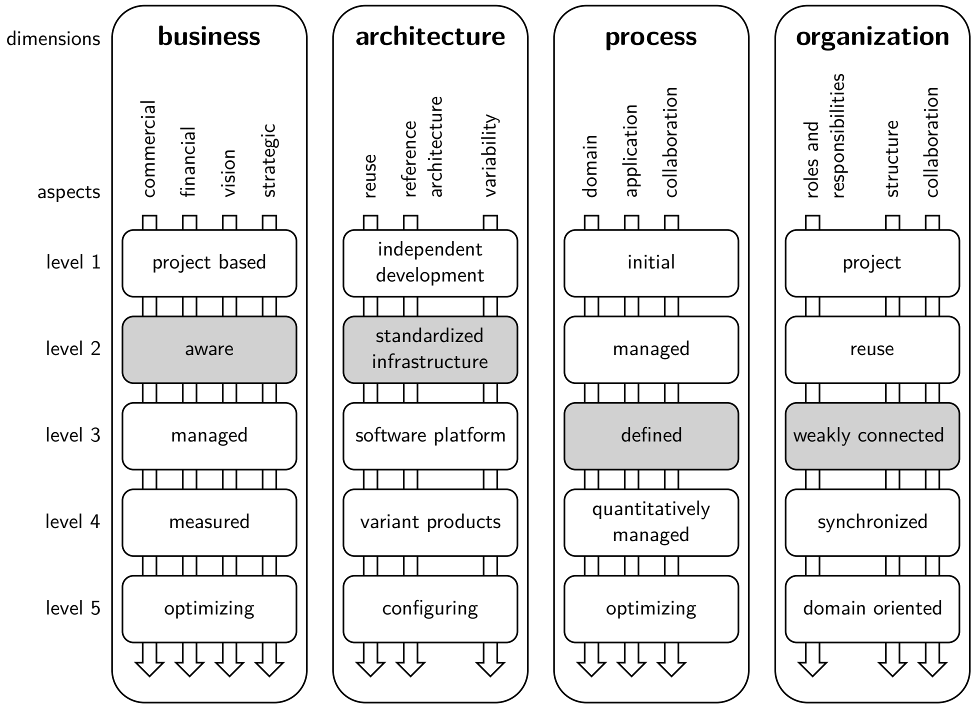


Figure 6.8: Example assessment of one platform based on the family evaluation framework.

Breaking Down Suggested Actions

Defined actions

The reports themselves cannot change anything in the Simulation Center. To establish the suggested actions, the Simulation Center conducted a series of workshops and meetings to:

- Break down the suggested actions into Scrum or Kanban stories,
- find budget to finance the suggested actions,
- prioritize all actions, and
- define which levels in the family evaluation framework each platform should achieve.

Obviously, there are trade offs between these goals. For instance, the levels a platform should achieve can immediately rule out some actions that aim for an even higher level or cannot be financed. Finally, the workshops led to a set of actions that will be performed on each platform to achieve the specified levels in the family evaluation framework.

Repeating

Repeat

The Simulation Center intends to establish the family evaluation framework as an integral part of its regular work. Achieving this integration requires that these assessments are repeated regularly, which means that they are adopted for monitoring. The Simulation Center declared that one year would be a suitable interval until the next round of assessments. These following rounds will allow to perform a retrospective analysis on the assessment we described as well as on the reports, the derived actions, and the whole methodology.

Expected Versus Obtained Results

Expectations and reality

During informal meetings with the technical management, we compared the expectations of using the family evaluation framework to the actual results, which revealed:

- For most platforms, there was a clear and mostly correct perception on the maturity in the architecture dimension. The possibilities and limitations in this dimensions are well-understood within the Simulation Center.
- For most platforms, there were overestimates regarding the maturity in the process dimension. We found that the amount of tacit knowledge compared to defined and documented processes was higher than anticipated.
- For one platform, we assessed the levels of all dimension with a 1 or 2. As a result, the management of that platform questioned the family evaluation framework, since they always finished their projects in time However, after explaining and reasoning on the results, the assessment was accepted with the mindset that corners may have been cut or that technical debt was acquired.
- For one platform, the corresponding managers had a clear impression of their abilities, which proved to be quite similar to the assessment, which indicated high levels in most dimensions. The management of the platform was not surprised, but the remaining Simulation Center had considerably lower expectations for that platform before the assessment. In this case, the results of the family evaluation framework provided justification for previous investments into developing the platform.

We can see that stakeholders may have different perceptions regarding the maturity and goals of a platform. The family evaluation framework can help to reveal mismatches between stakeholders' perceptions and the current state or goal. Additionally, assessing the maturity of a platform is a means to justify investments into that platform.

RO-P$_7$: Adapting and Using the Family Evaluation Framework

By using the family evaluation framework for real-world platforms, we learned:

- *To adapt and operationalize the family evaluation framework, it helps to*
  - *specify the scope of all relevant platforms before the assessment.*
  - *consider the organization's domain and needs while adapting questions.*
  - *derive a unified terminology and explain relevant concepts to stakeholders.*
  - *involve different stakeholder roles in the assessment.*
- *To elicit information, it helps to*
  - *conduct semi-structured interviews that involve different stakeholders.*
  - *select questions based on an interviewee's role and expertise.*
  - *document all answers and display them to the corresponding interviewee.*
- *To analyze information, it helps to*
  - *synthesize level assessments based on all available interviews.*
  - *identify particularly discrepancies between answers.*
  - *assess and update results with relevant stakeholders.*
- *To derive actions from the assessment, it helps to*
  - *involve stakeholders and derive smaller stories.*
  - *prioritize the actions based on goals and responsibilities.*
  - *deliver feedback on how implemented actions perform.*

6.3.3 RO-P$_8$: Challenges of the Family Evaluation Framework

Challenges

While using the family evaluation framework at the Simulation Center, we faced several challenges, of which we discuss six in this section. Note that these challenges are related to each other (following the BAPO principle) and quite explicitly to some of our other research objectives. For instance, justifying investments can benefit the handling of change requests, and is directly related to our economics objective (cf. Chapter 3). Still, these two challenges stem from different dimensions, namely business and process, respectively.

Separating Domain and Application Engineering

Domain and application engineering

Our interviewees had problems to describe the benefits of platform engineering for the Simulation Center. Moreover, they were unaware of activities outside of their own projects. As a consequence, particularly long-running projects tended to perform more and more domain engineering, which should actually be outside of an individual project. This underpins our findings in Section 3.2.3 and Section 6.1 that developers seem to care less about domain and application engineering than for adopting a platform and evolving variants. Besides the fact that proper domain engineering is still required to establish a successful platform, the missing distinction of both phases challenges the application of the family evaluation framework. Namely, if the interviewees do not understand the differences between domain and application engineering, any assessment is likely doomed to fail, since it will not reveal any helpful insights.

Focusing on the Assessment

Focusing

Sometimes, our interviewees wanted to discuss problems that were unrelated to the maturity assessment, and in a few cases it was problematic to refocus the interview. Such situations occurred because it was not obvious what information was unrelated to the family evaluation framework until we dug deeper into the discussion. If we found that a topic was not relevant to our assessment, we propagated it to the right forums. Despite such situations, it was more often useful to not interrupt an interviewee to keep them motivated. The main challenge was to allocate enough time for every interview, plan the recording, and balance to what degree to guide the interview.

Aligning Software-Engineering Practices

Agile practices

The Simulation Center uses agile methods, for example, Kanban and Scrum, which were often considered to oppose platform engineering. More precisely, platform engineering and the family evaluation framework suggest a more formal methodology in the form of domain and application engineering. Agile methods usually assume a high degree of volatility in requirements, whereas platform engineering prefers fixed, but configurable, requirements. So, a challenge for using the family evaluation framework in practice is how to integrate platform engineering with modern software-engineering practices, and to communicate that this can be achieved. Notably, we constructed promote-pl to solve such problems, and discussed its integration with different practices — including agility (cf. Section 6.1).

Identifying Issues Outside the Platform

Task forces

A specific issue we identified at the Simulation Center is the misuse of the platform, that is its modification based on a variant without considering dependencies to other variants. Particularly, this issue occurs in the context of task forces, which have the sole purpose of solving a certain problem as fast as possible, typically driven by an urgent deadline.

Consequently, task forces often disregard the boundaries of projects, processes, teams, or the platform and its dependencies. While they are helpful to quickly resolve a problem, task forces easily introduce unintended dependencies into the platform. With respect to the family evaluation framework, we found it challenging to identify this established practice, despite the considerably impact on the platform. Precisely, since task forces act independently of other organizational structures, few of our interviewees did know about them.

Justifying Investments

Justifications

Conducting the assessment itself and implementing changes based on the results causes costs (cf. Chapter 3). Such changes are hard to justify, particularly if everything seems to work or if the platform involves few variants. Often, everything works well because tacit knowledge is incorporated into the stakeholders' minds, without documenting it (cf. Chapter 4). While such a mindset helps to critically reflect the assessment results, it can also prevent important change. Ideally, we can provide a reasonable justification for the investments to convince stakeholders that the proposed changes are valuable.

Managing Change Requests

Change requests

Change requests represent the actionable items defined in our reports. Unfortunately, change requests can be challenging to manage if they do not align to customer requests or provide no immediate value. Change requests are mostly project-based, which prioritize, plan, finance, and implement only those requests relevant to them. In contrast, most changes that are proposed based on the assessment are related to domain engineering, which is usually out of the scope of a project. Without organizational changes, each project may accept or decline a request, depending on the available budget (cf. Chapter 3). Consequently, it is challenging for any organization to manage change requests that stem from the family evaluation framework without adapting its structure first (e.g., introducing a platform team, allocating separate platform budget). Moreover, for the family evaluation framework, it is key to identify the stakeholders to whom change requests must be propagated.

RO-P$_8$: Challenges of the Family Evaluation Framework

By applying the family evaluation framework, we learned that it is challenging

- *to separate domain and application engineering, since these may be tangled.*
- *to keep interviewees' focus on the assessment instead of other problems.*
- *to integrate platform engineering with other software-engineering practices.*
- *to find problems that are caused by factors outside of the platform.*
- *to justify that applying the assessment and proposed actions is beneficial.*
- *to manage change requests if the responsible stakeholder is unknown.*

6.3.4 RO-P$_9$: Benefits of the Family Evaluation Framework

Benefits

The platforms we analyzed have been evolved for more than ten years, resulting in consequent histories and legacy artifacts. The previous challenges are mainly intended to close differences between legacy and modern platform engineering, and to coordinate the necessary changes. However, using the family evaluation framework as a standardized assessment yielded immense benefits for the Simulation Center, too. Actually, the management of the Simulation Center agreed that the benefits outweighed the challenges. Moreover, all stakeholders support the improved focus on domain engineering and the long-term planning that comes with it. Based on this agreement, the Simulation Center decided to repeat the assessments, essentially using them for monitoring the platforms.

Disseminating Knowledge

Knowledge

One of our first experiences while using the family evaluation framework was the improved dissemination of knowledge on platform engineering (cf. Chapter 4). In fact, assessing different BAPO dimensions raised our interviewees' awareness for issues besides the architecture. Previously, the Simulation Center was not able to advance its platform engineering. By introducing a factory metaphor, we were able to help our interviewees understand that a platform requires standardized interfaces, tooling, well-defined roles, clear processes, and a business model. Arguably, the dissemination of knowledge is a major benefits that improves expertise, eases communication, and establishes a common knowledge base.

Connecting Stakeholders

Common understanding

The reports that combined the perspectives of managers, technical leads, and engineers established a common understanding between stakeholders. Before the assessment, most of them communicated only to resolve issues or in specific forums. While applying the family evaluation framework, every interviewee could mention problems, ranging from small technical issues to concerns about the strategical orientation of the platform. The reports defined a common ground for discussions that was not available before, which is immensely beneficial.

Identifying Shortcomings

Assessment framework

The main purpose of the family evaluation framework is to assess the maturity of a platform and identify potential for improvements. We experienced that interviews were a suitable method to elicit information and to identify shortcomings in the platform engineering. In our opinion, the main benefit of using the family evaluation framework is the well-defined structure along four dimensions. This defined setting helped us to identify, manage, and document shortcomings that the Simulation Center did not know about or that were somewhat vague (e.g., considering who would be responsible).

Defining Road-Maps for Platforms

Platform road-maps

Our reports about the interviews provided first road-maps for guiding the domain engineering of each platform. Before, several of the platforms had no defined domain-engineering activities. Such activities were still executed, but usually in a single process that integrated domain and application engineering. The assessment highlighted that separating the road-map for domain engineering, the feature model (cf. Section 6.2), the finance model, and the organizational units would be beneficial for the Simulation Center. Additionally, separating the domain engineering led to more long-term thinking about technological advances and the inclusion of state-of-the-art methodologies and tools.

Setting Goals for the Platforms

Defined goals

Using the levels in the family evaluation framework provided a novel way for defining goals for a platform. Historically, the Simulation Center used somewhat standardized goals, such as costs, technological levels, or time plans — which could hardly be compared between different platforms. The family evaluation framework defined a common ground for defining and discussing goals for and between platforms. This also benefited the communication between stakeholders, who now could more easily refer to the goals of their platforms.

Lowering Maintenance and Development Costs

Economic impact

In alignment with our previous findings in Chapter 3, we experienced that higher levels of maturity reduced development and maintenance costs — particularly for platforms with more variability, and thus a larger number of variants. The Simulation Center established its first platform in 2010, and data shows that development and maintenance costs went down by at least 50 %. At the same time, new variants can be priced more precisely. Using the family evaluation framework helped to raise the awareness for these benefits, and can lead to even more savings by improving the platform engineering further.

RO-P$_9$: Benefits of the Family Evaluation Framework

Employing the family evaluation framework had the benefits of

- *disseminating platform-engineering knowledge among stakeholders.*
- *connecting stakeholders by establishing the unified reports as knowledge base.*
- *identifying shortcomings based on a structured assessment.*
- *defining road-maps for platforms with missing domain-engineering activities.*
- *specifying comparable goals for platforms.*
- *planning systematically how to lower development and maintenance costs.*

6.3.5 Threats to Validity

Threats to validity

In the following, we discuss potential threats to the validity of our study. As we can see, most of these threats relate to the fact that we studied only a single organization.

Construct Validity

Knowledge on platform engineering

As aforementioned, some stakeholders we interviewed were not familiar with the concepts of platform engineering. We mitigated this threat by unifying the domain terms of the Simulation Center and mapping them to the family evaluation framework. Additionally, we explained all necessary concepts during the interviews to establish a comparable knowledge base and common understanding among our interviewees. To verify that our data was reliable and to collect additional feedback, we allowed all involved stakeholders to review the reports and conducted several workshops. These means improve our confidence in our data and its synthesis, but we cannot fully avoid this threat.

Internal Validity

Subject selection

We aimed to limit potential threats originating from the selected platforms, involved stakeholders, and adaptions of the family evaluation framework. As a first means, we conducted a cross-case analysis to synthesize results from all nine cases. In addition, we invited stakeholders with different roles to obtain complementary data, and ensured that all of them were experts regarding their respective platform. While we could not mitigate potential bias of adapting our questions, we argue that these are actually part of our contributions.

External Validity

Single organization

We aimed to improve the external validity by conducting nine cases. Still, all cases are based on a single organization, which means that our results may not be transferable to other organizations. Since we provide the first report on how to perform a maturity assessment with the family evaluation framework on large-scale real-world platforms, we argue that our study is nonetheless valuable.

Subject organization

Some properties of the Simulation Center threaten the external validity of our study. For instance, the Simulation Center faces requirements that are not representative of other domains, which we cannot resolve by any means. We aimed to mitigate such threats by considering a set of platforms that should exhibit comparable properties to those in other domains. The main differences to other organizations may be the business dimension. Since the Simulation Center mainly delivers to internal customers of its parent organization, our interview data on this dimension may not be representative. However, this should not change how the family evaluation framework can be used, which was our primary goal.

Conclusion Validity

Confidentiality

We designed our methodology within a team of four researchers and used action research to adapt it to new findings. However, for confidentially reasons, we could neither study nor publish all of our data, for instance, regarding the tools, customers, guidelines, or processes of the Simulation Center. This prevents exact replications of our study and means that we must be careful while interpreting our findings. Such problems threaten any study in industrial settings and cannot be fully resolved. We still published as much of our data as possible and argue that other researchers or organizations can understand as well as replicate our study.

6.4 Summary

Chapter summary

In this chapter, we integrated our findings into processes and recommendations that support particularly the planning and monitoring of (re-)engineering projects. At first, we proposed a novel process model for platform engineering that involves contemporary processes and provides a better integration of modern software-engineering practices. Afterwards, we presented feature-modeling principles, which are relevant for all other planning activities, too. We concluded with a multi-case study that details how a variant-rich system can be assessed in practice, which supports decision making, monitoring, and the definition of goals.

Summarizing contributions

The contributions in this chapter help especially practitioners: They can build on promote-pl to organize their projects. Furthermore, they can use our feature-modeling principles as well as experiences on maturity assessments as practices within promote-pl. Also, we highlighted how our results are connected to contemporary software-engineering practices, which helps integrate these. Regarding research, we collected a large body-of-knowledge from empirical data that helps address several research gaps, and highlights directions for future research. For instance, many contemporary software-engineering practices are still not well-supported in existing tools for platform engineering. Since our data shows that integrating such practices with platform engineering is needed, new techniques in this direction can immediately benefit practice. Moreover, our results clearly show that new monitoring tools (e.g., for automated maturity assessments) can help organizations achieve immense benefits from their variant-rich systems. Abstractly, our results suggest the following core finding:

RO-P: Practices
Our processes and recommendations for planning, initiating, steering, and monitoring a variant-rich system can help organizations to utilize the system's full benefits.

Connection to other research objectives

As already mentioned, this chapter builds upon and combines findings we reported in previous chapters. Namely, the development process we elicited in Chapter 3 is part of promote-pl, and the economic data in that chapter is important to reason on decisions while planning and monitoring a variant-rich system (**RO-E**). Similarly, the knowledge and documentation issues that we discussed in Chapter 4 are highly relevant in this chapter, for instance, considering recommendations to rely on experts' knowledge (**RO-K**). Finally, the feature traceability we discussed in Chapter 5 has been mentioned less often explicitly, but has been highlighted as essential in our sources for promote-pl (i.e., "map artifact") and is ideally combined with a feature model (**RO-T**).

7. Conclusion

Concluding remarks

In the following, we briefly summarize the chapters of this dissertation. Moreover, we point out the most important contributions regarding each of our research objectives. Finally, we define directions for future research that build on our contributions.

7.1 Summary

Chapter 3

In Chapter 3, we studied the economics of software reuse. At first, we analyzed existing cost models for software product-line engineering and discussed how these relate to the re-engineering of variant-rich systems. Our insights highlighted that we require systematically elicited data on software reuse to understand the pros and cons of different reuse strategies as well as the relevant cost factors. Then, we explored two research directions to tackle this problem: On the one hand, we elicited empirical data on the costs of developing new variants via clone & own and platform engineering — building on the literature and our collaboration with a large organization. Our data helped confirm and refute established hypotheses on software reuse, providing actual evidence that can support an organization's decision-making beyond educated guesses. On the other hand, we conducted a multi-case study with five cases to understand the challenges and economics of re-engineering cloned variants into a platform. By recording experiences and costs, we provided details on the conduct of re-engineering projects and their economical impact.

Chapter 4

In Chapter 4, we studied developers' knowledge needs, which we found to be among the cost factors with most impact on the economics of (re-)engineering variant-rich systems. Initially, we investigated what types of knowledge developers intend to memorize and consider important. Based on a systematic literature review and an interview survey, we showed that developers are quite good at recalling domain abstractions (e.g., features), while it seems that they do not intend to memorize implementation details (e.g., assets). We continued with a survey on developers' memory regarding their code, which revealed that we can somewhat adopt existing forgetting curves from psychology to understand and measure how developers forget. Seeing the need for recovering knowledge, we conducted a multi-case study on two variant-rich systems in which we recovered various feature facets (i.e., knowledge about a feature's properties) that are relevant to evolve (e.g., re-engineering) an variant-rich system. Our results show what information sources can help recover what feature facets, guiding developers during their re-engineering projects and the design of specialized reverse-engineering techniques.

Chapter 5 In Chapter 5, we studied feature traceability as a technique that can help mitigate the knowledge problem, and thus reduce costs by preventing expensive recovery activities. In the beginning, we built on related work and a set of different studies to define dimensions of feature traceability and discuss how these impact developers' knowledge — and thus the economics of re-engineering a variant-rich system. Since the empirical evidence regarding these dimensions has been unconvincing, we conducted two experiments. First, we compared different representations of feature traces to each other (i.e., none, virtual, physical), which indicated that a virtual representation (i.e., feature annotations) seems to be the most helpful and least intrusive technique. Second, we studied how the use of feature traces (i.e., documenting versus configuring) can impact developers' program comprehension. Besides revealing an interesting dilemma regarding the discipline of configurable directives, our results also suggest that the configurability adds a layer of complexity to feature traces that should ideally be avoided.

Chapter 6 In Chapter 6, we synthesized our previous findings into processes and recommendations. Namely, we derived a novel process model, promote-pl, which provides an updated perspective on platform engineering and its relations to modern software-engineering practices. In particular, promote-pl incorporates re-engineering processes and activities to a great extent, since these are far more commonly applied when adopting or evolving a platform. Then, we synthesized 34 feature-modeling principles that help an organization construct a feature model and plan its platform-engineering. Not surprisingly, we also found that these principles are largely related to the re-engineering of variants. Finally, we reported a multi-case study on our experiences of using the family evaluation framework to assess the maturity of nine variant-rich system. Our findings suggest that periodical maturity assessments help not only to define goals, but can have immediate benefits on their own.

7.2 Contributions

***RO-E**: economics* Our key contribution regarding **RO-E** is our systematically elicited empirical data on the economics of (re-)engineering variant-rich systems. We used our data to confirm as well as refute established hypotheses on software reuse, which helps organizations to reason about their reuse strategy and opens new directions for research. Moreover, we contributed concrete insights into how to re-engineer cloned variants into a platform, highlighting pitfalls as well as benefits. Considering all of our results, we found that any organization should strive towards systematizing its software reuse, for example, by incrementally adopting a platform. Overall, our data contributes to addressing the long-standing problem of providing decision support for the (re-)engineering of variant-rich systems based on empirical data.

***RO-K**: knowledge* Our key contribution regarding **RO-K** is the improved understanding of how developers forget. In particular, our data suggests that developers are good at memorizing domain concepts like features, but require support for mapping these to the concrete assets — and consequent information must be recorded. To help organizations during the re-engineering of a variant-rich system, we further contributed insights into what information sources can be exploited to recover knowledge on features. Such information is fundamental to plan and initiate a re-engineering project. So, we contributed to a better understanding of how to elicit the knowledge that is required to re-engineer a variant-rich system.

***RO-T**: traceability* Our key contribution regarding **RO-T** is the insight that feature traceability should ideally be separated from variability. Thus, feature locations can be easily identified in the source code without the need for developers to understand how the traces change the behavior of a variant. To this end, our data suggests to rely on light-weight feature annotations that seem to facilitate developers' program comprehension. In summary, we contribute empirical evidence that feature traceability is a helpful means for (re-)engineering a variant-rich system and provide concrete recommendations on how to implement it.

__RO-P__: practices

Our key contribution regarding **RO-P** is promote-pl, which synthesizes our findings into a contemporary process model. Consequently, it can guide organizations by defining processes and activities for the (re-)engineering of variant-rich systems. We further defined concrete recommendations in the form of feature-modeling principles and a process for employing maturity assessments. These three contributions form a framework to help organizations plan, initiate, steer, and monitor the (re-)engineering of a variant-rich system. So, we contributed to the problem that established software product-line process models do not reflect contemporary software-engineering practices and miss concrete recommendations for adopting a variant-rich system.

The big picture

In this dissertation, we aimed at *understanding the re-engineering of variant-rich systems* based on empirical studies. Such an understanding is essential to decide on, plan, initiate, steer, and monitor re-engineering projects. We argue that we achieved this goal, by contributing, in a nutshell:

1. economical data that helps estimate costs and justify decisions;
2. insights into how to elicit and record information;
3. recommendations on how to decide on and implement feature traceability; and
4. processes and recommendations for planning a platform and defining its goals

to the existing body-of-knowledge on re-engineering variant-rich systems. However, not surprisingly, many of our insights exceed this scope.

7.3 Future Work

Design of decision-support systems

As we have shown, numerous cost models have been proposed to support the planning of a variant-rich system. However, it is largely unclear to what extent which model reflects properly on the real world and can actually guide organizations. We improved on this situation by contributing empirical data on the economics of variant-rich systems. Still, we cannot (yet) design a cost model based on our data, which may also not be that helpful. Instead, we argue that our insights can serve as a foundation for more light-weight and scope-specific decision-support systems before advancing to full-fledged cost models. Precisely, we envision systems that reason on certain sub-problems of (re-)engineering a variant-rich system that can be explored in greater detail, such as deciding whether it is economically beneficial to integrate a feature of a cloned variant into a platform. Collecting the required data and designing such (more focused) systems seems more feasible and helpful to support organizations.

Developers' memory

Our research on developers' knowledge needs is obviously not complete. There are numerous paths that should be explored to provide better tools to developers that help record, manage, and recover information. For instance, it would be helpful to improve our understanding of developers' memory further to derive a forgetting model for software engineering and understand how knowledge needs arise. Based on such insights, new tools for eliciting relevant information from different source could be designed. Moreover, we could establish recommendations for recording information and onboarding newcomers based on empirical data.

Management framework for feature traces

We found that feature traces help developers by making feature locations explicit in the source code. Still, the problem of maintaining these annotations remains, and they should ideally be also traced to other artifacts, such as a feature model. Unfortunately, developers mainly trust the actual source code of a system, but not additional constructs that may co-evolve. As a result, our data also shows that many developers have an aversion of using such constructs like feature annotations. Thus, we argue that we have to design frameworks that help manage feature traces, for instance, by providing (semi-)automation for intruding and maintaining (e.g., assuring consistency) them. For this purpose, it seems most promising to first design such frameworks for annotations that are part of a programming language (e.g., Java annotations) or a widely used tool (e.g., JUnit annotations, C preprocessor directives) to promote their benefits and convince developers.

Employing recommendations in practice

We constructed our processes and recommendations from recent publications and our collaborations with industry. Consequently, they should represent contemporary practices of engineering variant-rich systems. An important direction to refine, extend, and combine these contributions is to employ them in practice. Precisely, instantiating them in different organizations can help to unveil potential shortcomings that may be caused by varying properties of different domains or organizations. Moreover, such an operationalization can provide experiences that guide other organizations, suggest best practices for individual steps, and provide insights into the costs as well as benefits of our recommendations — similar to our findings on the family evaluation framework.

Replication studies

The research in this dissertation is based on empirical studies. While we aimed to improve the validity of our findings by synthesizing from various sources, there remain several threats. For instance, our experiments represent single data points that require further confirmation and our multi-case studies are limited to smaller systems or a single organization. To improve the confidence in our results, explore them in more detail, and consider different contexts, replication studies are required. Consequently, conducting replications is an important future research direction.

New technologies and tools

A severe problem we faced while conducting our research are the limited capabilities of existing tools for re-engineering variant-rich systems. For instance, during our multi-case study on re-engineering, we had to adapt existing or even implement own tools. While some adaptations may be highly domain dependent, and thus not relevant for many variant-rich systems, we argue that others demand for further research. Concretely, we had no suitable support for the actual re-engineering of cloned variants, which is a severe problem seeing that this is the most common adoption strategy. Arguably, clone-management or synchronization frameworks can be adopted for this purpose, but require novel techniques and adaptations. Furthermore, Android and web-services represent domains with large numbers of variant-rich systems of any maturity. Usually, variant-rich systems in such domains build on a combination of technologies and programming languages, which is not well-supported by existing tools. So, a direction for future work is to explore how techniques and tools for variant-rich systems can be designed to support multiple technologies (and potentially platforms) simultaneously

Bibliography

Iago Abal, Jean Melo, Ștefan Stănciulescu, Claus Brabrand, Márcio de Medeiros Ribeiro, and Andrzej Wąsowski. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *ACM Transactions on Software Engineering and Methodology*, 26(3):10:1–34, 2018. doi:10.1145/3149119. (cited on Page 94 and 113)

Muhammad Abbas, Robbert Jongeling, Claes Lindskog, Eduard P. Enoiu, Mehrdad Saadatmand, and Daniel Sundmark. Product Line Adoption in Industry: An Experience Report from the Railway Domain. In *International Systems and Software Product Line Conference (SPLC)*, pages 3:1–11. ACM, 2020. doi:10.1145/3382025.3414953. (cited on Page 1)

Hadil Abukwaik, Andreas Burger, Berima K. Andam, and Thorsten Berger. Semi-Automated Feature Traceability with Embedded Annotations. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 529–533. IEEE, 2018. doi:10.1109/icsme.2018.00049. (cited on Page 96)

Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming*, 78(6):657–681, 2013. doi:10.1016/j.scico.2012.12.004. (cited on Page 175)

Faheem Ahmed and Luiz F. Capretz. An Organizational Maturity Model of Software Product Line Engineering. *Software Quality Journal*, 18(2):195–225, 2010a. doi:10.1007/s11219-009-9088-5. (cited on Page 177)

Faheem Ahmed and Luiz F. Capretz. A Business Maturity Model of Software Product Line Engineering. *Information Systems Frontiers*, 13(4):543–560, 2010b. doi:10.1007/s10796-010-9230-8. (cited on Page 177)

Faheem Ahmed, Luiz F. Capretz, and Shahbaz A. Sheikh. Institutionalization of Software Product Line: An Empirical Investigation of Key Organizational Factors. *Journal of Systems and Software*, 80(6):836–849, 2007. doi:10.1016/j.jss.2006.09.010. (cited on Page 177)

Jonas Åkesson, Sebastian Nilsson, Jacob Krüger, and Thorsten Berger. Migrating the Android Apo-Games into an Annotation-Based Software Product Line. In *International Systems and Software Product Line Conference (SPLC)*, pages 103–107. ACM, 2019. doi:10.1145/3336294.3342362. (cited on Page 19, 49, 51, and 53)

Abdullah Al-Nayeem, Krzysztof Ostrowski, Sebastian Pueblas, Christophe Restif, and Sai Zhang. Information Needs for Validating Evolving Software Systems: An Exploratory Study at Google. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 544–545. IEEE, 2017. doi:10.1109/icst.2017.64. (cited on Page 69)

Muhammad S. Ali, Muhammad A. Babar, and Klaus Schmid. A Comparative Survey of Economic Models for Software Product Lines. In *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 275–278. IEEE, 2009. doi:10.1109/seaa.2009.89. (cited on Page 4, 21, 22, and 23)

Muhammad S. Ali, Muhammad A. Babar, Lianping Chen, and Klaas-Jan Stol. A Systematic Review of Comparative Evidence of Aspect-Oriented Programming. *Information and Software Technology*, 52(9):871–887, 2010. doi:10.1016/j.infsof.2010.05.003. (cited on Page 117)

Vander R. Alves, Nan Niu, Carina Alves, and George Valença. Requirements Engineering for Software Product Lines: A Systematic Literature Review. *Information and Software Technology*, 52(8):806–820, 2010. doi:10.1016/j.infsof.2010.03.014. (cited on Page 4)

Valentin Amrhein, Sander Greenland, and Blake McShane. Retire Statistical Significance: Scientists Rise Up against Statistical Significance. *Nature*, 567(7748):305–307, 2019. doi:10.1038/d41586-019-00857-9. (cited on Page 86, 122, 133, and 144)

Berima Andam, Andreas Burger, Thorsten Berger, and Michel R. V. Chaudron. FLOrIDA: Feature LOcatIon DAshboard for Extracting and Visualizing Feature Traces. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 100–107. ACM, 2017. doi:10.1145/3023956.3023967. (cited on Page 4, 5, 96, and 108)

Henric Andersson. *Variability and Customization of Simulator Products: A Product Line Approach in Model Based Systems Engineering*. PhD thesis, Linköping University, 2012. (cited on Page 179)

Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Ștefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. Flexible Product Line Engineering with a Virtual Platform. In *International Conference on Software Engineering (ICSE)*, pages 532–535. ACM, 2014. doi:10.1145/2591062.2591126. (cited on Page 159 and 160)

Giuliano Antoniol, Jane Cleland-Huang, Jane H. Hayes, and Michael Vierhauser, editors. *Grand Challenges of Traceability: The Next Ten Years*. CoRR, 2017. URL https://arxiv.org/abs/1710.03129v1. (cited on Page 5 and 108)

John Anvik, Lyndon Hiew, and Gail C. Murphy. Who Should Fix This Bug? In *International Conference on Software Engineering (ICSE)*, pages 361–370. ACM, 2006. doi:10.1145/1134285.1134336. (cited on Page 66)

Sven Apel and Christian Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–48, 2009a. doi:10.5381/jot.2009.8.5.c5. (cited on Page 11, 117, and 131)

Sven Apel and Christian Kästner. Virtual Separation of Concerns – A Second Chance for Preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009b. doi:10.5381/jot.2009.8.6.c5. (cited on Page 110)

Sven Apel and Christian Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *International Conference on Software Composition (SC)*, pages 20–35. Springer, 2008. doi:10.1007/978-3-540-78789-1_2. (cited on Page 17)

Sven Apel, Christian Kästner, and Christian Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *International Conference on Software Engineering (ICSE)*, pages 221–231. IEEE, 2009. doi:10.1109/icse.2009.5070523. (cited on Page 17 and 53)

Sven Apel, William Cook, Krzysztof Czarnecki, and Oscar Nierstrasz, editors. *Feature-Oriented Software Development (FOSD)*. Schloss Dagstuhl, 2011. (cited on Page 131 and 148)

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013a. doi:10.1007/978-3-642-37521-7. (cited on Page 1, 2, 5, 11, 13, 14, 15, 16, 18, 56, 62, 107, 108, 109, 110, 112, 124, and 153)

Sven Apel, Christian Kästner, and Christian Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013b. doi:10.1109/tse.2011.120. (cited on Page 17, 52, and 53)

Wesley K. G. Assunção and Silvia Regina Vergilio. Feature Location for Software Product Line Migration: A Mapping Study. In *International Software Product Line Conference (SPLC)*, pages 52–59. ACM, 2014. doi:10.1145/2647908.2655967. (cited on Page 4, 28, 29, and 56)

Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. Reengineering Legacy Applications into Software Product Lines: A Systematic Mapping. *Empirical Software Engineering*, 22(6):2972–3016, 2017. doi:10.1007/s10664-017-9499-z. (cited on Page 2, 3, 4, 5, 32, 49, 56, 148, 149, 152, and 153)

Wesley K. G. Assunção, Jacob Krüger, and Willian D. F. Mendonça. Variability Management meets Microservices: Six Challenges of Re-Engineering Microservice-Based Webshops. In *International Systems and Software Product Line Conference (SPLC)*, pages 22:1–6. ACM, 2020. doi:10.1145/3382025.3414942. (cited on Page 160)

Lee Averell and Andrew Heathcote. The Form of the Forgetting Curve and the Fate of Memories. *Journal of Mathematical Psychology*, 55(1):25–35, 2011. doi:10.1016/j.jmp.2010.08.009. (cited on Page 68, 83, and 92)

Felix Bachmann and Paul C. Clements. Variability in Software Product Lines. Technical Report CMU/SEI-2005-TR-012, Carnegie Mellon University, 2005. (cited on Page 15)

Noor Hasrina Bakar, Zarinah M. Kasirun, and Norsaremah Salleh. Feature Extraction Approaches from Natural Language Requirements for Reuse in Software Product Lines: A Systematic Literature Review. *Journal of Systems and Software*, 106:132–149, 2015. doi:10.1016/j.jss.2015.05.006. (cited on Page 4)

Monya Baker. Statisticians Issue Warning over Misuse of P Values. *Nature*, 531(7593): 151–151, 2016. doi:10.1038/nature.2016.19503. (cited on Page 86, 122, 133, and 144)

Sebastian Baltes and Stephan Diehl. Worse Than Spam: Issues In Sampling Software Developers. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 52:1–6. ACM, 2016. doi:10.1145/2961111.2962628. (cited on Page 138)

José L. Barros-Justo, Fernando Pinciroli, Santiago Matalonga, and Nelson Martínez-Araujo. What Software Reuse Benefits Have Been Transferred to the Industry? A Systematic Mapping Study. *Information and Software Technology*, 103:1–21, 2018. doi:10.1016/j.infsof.2018.06.003. (cited on Page 32 and 37)

Anton Barua, Stephen W. Thomas, and Ahmed E. Hassan. What are Developers Talking About? An Analysis of Topics and Trends in Stack Overflow. *Empirical Software Engineering*, 19(3):619–654, 2014. doi:10.1007/s10664-012-9231-y. (cited on Page 115)

Julian M. Bass, Sarah Beecham, and John Noll. Experience of Industry Case Studies: A Comparison of Multi-Case and Embedded Case Study Methods. In *International Workshop on Conducting Empirical Studies in Industry (CESI)*, pages 13–20. ACM, 2018. doi:10.1145/3193965.3193967. (cited on Page 50, 63, 93, and 177)

Len Bass, Paul C. Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1999. (cited on Page 36, 44, 46, and 47)

Don Batory. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In *International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 3–35. Springer, 2006. doi:10.1007/11877028_1. (cited on Page 17)

Don Batory, Jacob N. Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004. doi:10.1109/tse.2004.23. (cited on Page 17)

Veronika Bauer and Antonio Vetrò. Comparing Reuse Practices in Two Large Software-Producing Companies. *Journal of Systems and Software*, 117:545–582, 2016. doi:10.1016/j.jss.2016.03.067. (cited on Page 2, 10, 36, 41, 42, 43, 45, and 47)

Veronika Bauer, Jonas Eckhardt, Benedikt Hauptmann, and Manuel Klimek. An Exploratory Study on Reuse at Google. In *International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, pages 14–23. ACM, 2014. doi:10.1145/2593850.2593854. (cited on Page 1, 36, 41, 46, and 47)

Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean-Marc DeBaud. PuLSE: A Methodology to Develop Software Product Lines. In *Symposium on Software Reusability (SSR)*, pages 122–131. ACM, 1999. doi:10.1145/303008.303063. (cited on Page 4, 20, and 177)

Benjamin Behringer. *Projectional Editing of Software Product Lines - The PEoPL Approach*. PhD thesis, University of Luxembourg, 2017. (cited on Page 111)

Benjamin Behringer and Moritz Fey. Implementing Delta-Oriented SPLs using PEoPL: An Example Scenario and Case Study. In *International Workshop on Feature-Oriented Software Development (FOSD)*, pages 28–38. ACM, 2016. doi:10.1145/3001867.3001871. (cited on Page 111)

Benjamin Behringer, Jochen Palz, and Thorsten Berger. PEoPL: Projectional Editing of Product Lines. In *International Conference on Software Engineering (ICSE)*, pages 563–574. IEEE, 2017. doi:10.1109/icse.2017.58. (cited on Page 111)

Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007. doi:10.1109/tse.2007.70725. (cited on Page 2 and 52)

Sana Ben Abdallah Ben Lamine, Lamia L. Jilani, and Henda H. Ben Ghezala. A Software Cost Estimation Model for a Product Line Engineering Approach: Supporting Tool and UML Modeling. In *International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 383–390. IEEE, 2005a. doi:10.1109/sera.2005.16. (cited on Page 22)

Sana Ben Abdallah Ben Lamine, Lamia L. Jilani, and Henda H. Ben Ghezala. Cost Estimation for Product Line Engineering Using COTS Components. In *International Software Product Line Conference (SPLC)*, pages 113–123. Springer, 2005b. doi:10.1007/11554844_13. (cited on Page 22)

Sana Ben Abdallah Ben Lamine, Lamia L. Jilani, and Henda H. Ben Ghezala. A Software Cost Estimation Model for Product Line Engineering: SoCoEMo-PLE. In *International Conference on Software Engineering Research and Practice (SERP)*, pages 649–655. CSREA, 2005c. (cited on Page 22)

David Benavides and José A. Galindo. Automated Analysis of Feature Models. Current State and Practices. In *International Systems and Software Product Line Conference (SPLC)*, pages 298–298. ACM, 2018. doi:10.1145/3233027.3233055. (cited on Page 162)

David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–636, 2010. doi:10.1016/j.is.2010.01.001. (cited on Page 14, 15, 18, 162, and 174)

Fabian Benduhn, Reimar Schröter, Andy Kenner, Christopher Kruczek, Thomas Leich, and Gunter Saake. Migration from Annotation-Based to Composition-Based Product Lines: Towards a Tool-Driven Process. In *International Conference on Advances and Trends in Software Engineering (SOFTENG)*, pages 102–109. IARIA, 2016. (cited on Page 52 and 111)

Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In *International Conference on Automated Software Engineering (ASE)*, pages 73–82. ACM, 2010. doi:10.1145/1858996.1859010. (cited on Page 165 and 173)

Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A Survey of Variability Modeling in Industrial Practice. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 1–8. ACM, 2013a. doi:10.1145/2430502.2430513. (cited on Page 2, 11, 12, 14, 148, 159, 162, and 165)

Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640, 2013b. doi:10.1109/tse.2013.34. (cited on Page 15, 165, and 173)

Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. Three Cases of Feature-Based Variability Modeling in Industry. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 302–319. Springer, 2014a. doi:10.1007/978-3-319-11653-2_19. (cited on Page 148, 165, and 171)

Thorsten Berger, Rolf-Helge Pfeiffer, Reinhard Tartler, Steffen Dienst, Krzysztof Czarnecki, Andrzej Wąsowski, and Steven She. Variability Mechanisms in Software Ecosystems. *Information and Software Technology*, 56(11):1520–1535, 2014b. doi:10.1016/j.infsof.2014.05.005. (cited on Page 15, 165, and 170)

Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *International Software Product Line Conference (SPLC)*, pages 16–25. ACM, 2015. doi:10.1145/2791060.2791108. (cited on Page 1, 4, 11, 15, 62, 94, and 112)

Thorsten Berger, Marsha Chechik, Timo Kehrer, and Manuel Wimmer, editors. *Software Evolution in Time and Space: Unifying Version and Variability Management*. Schloss Dagstuhl, 2019. (cited on Page 152)

Thorsten Berger, Jan-Philipp Steghöfer, Tewfik Ziadi, Jacques Robin, and Jabier Martinez. The State of Adoption and the Challenges of Systematic Variability Management in Industry. *Empirical Software Engineering*, 25:1755–1797, 2020. doi:10.1007/s10664-019-09787-6. (cited on Page 1, 2, 4, 5, 9, 10, 12, 20, 45, 49, 148, 149, 159, 160, 174, and 177)

John Bergey, Sholom G. Cohen, Lawrence G. Jones, and Dennis Smith. Software Product Lines: Experiences from the Sixth DoD Software Product Line Workshop. Technical Report CMU/SEI-2004-TN-011, Carnegie Mellon University, 2004. (cited on Page 36, 42, and 47)

Danilo Beuche. Modeling and Building Software Product Lines with pure::variants. In *International Software Product Line Conference (SPLC)*, pages 255–255. ACM, 2012. doi:10.1145/2364412.2364457. (cited on Page 2, 61, and 176)

Stefan Biffl, Aybüke Aurum, Barry W. Boehm, Hakan Erdogmus, and Paul Grünbacher, editors. *Value-Based Software Engineering*. Springer, 2006. doi:10.1007/3-540-29263-2. (cited on Page 20)

Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. The Concept Assignment Problem in Program Understanding. In *Working Conference on Reverse Engineering (WCRE)*, pages 482–498. IEEE, 1993. doi:10.1109/wcre.1993.287781. (cited on Page 4, 28, and 62)

BigLever Software, Inc. BigLever Software Case Study: Engenio. Technical Report 2005-06-14-1, BigLever Software, Inc., 2005. (cited on Page 36, 42, 43, and 46)

Andreas Birk, Gerald Heller, Isabel John, Klaus Schmid, Thomas von der Maßen, and Klaus Müller. Product Line Engineering: The State of the Practice. *IEEE Software*, 20 (6):52–60, 2003. doi:10.1109/ms.2003.1241367. (cited on Page 1)

J. Martin Bland and Douglas G. Altman. The Odds Ratio. *BMJ*, 320(7247):1468–1468, 2000. doi:10.1136/bmj.320.7247.1468. (cited on Page 139)

Mario G. Blandón, Paulo P. Cano, and Clifton Clunie. Cost Estimate Methods Comparison in Software Product Lines (SPL). *Informática y Control*, 1(2):18–24, 2013. (cited on Page 21, 22, 23, and 24)

Günter Böckle, Jesús B. Muñoz, Peter Knauber, Charles W. Krueger, Julio C. S. do Prado Leite, Frank J. van der Linden, Linda M. Northrop, Michael Stark, and David M. Weiss. Adopting and Institutionalizing a Product Line Culture. In *International Software Product Line Conference (SPLC)*, pages 49–59. Springer, 2002. doi:10.1007/3-540-45652-x_4. (cited on Page 26)

Günter Böckle, Paul C. Clements, John D. McGregor, Dirk Muthig, and Klaus Schmid. A Cost Model for Software Product Lines. In *International Workshop on Software Product-Family Engineering (PFE)*, pages 310–316. Springer, 2004a. doi:10.1007/978-3-540-24667-1_23. (cited on Page 20, 22, 23, and 24)

Günter Böckle, Paul C. Clements, John D. McGregor, Dirk Muthig, and Klaus Schmid. Calculating ROI for Software Product Lines. *IEEE Software*, 21(3):23–31, 2004b. doi:10.1109/ms.2004.1293069. (cited on Page 2, 10, 20, 22, 23, and 24)

Barry W. Boehm. Software Engineering Economics. *IEEE Transactions on Software Engineering*, SE-10(1):4–21, 1984. doi:10.1109/tse.1984.5010193. (cited on Page 4, 19, 20, 21, 33, and 48)

Barry W. Boehm and LiGuo Huang. Value-Based Software Engineering: Reinventing "Earned Value" Monitoring and Control. *ACM SIGSOFT Software Engineering Notes*, 28 (2):4–11, 2003. doi:10.1145/638750.638776. (cited on Page 20)

Barry W. Boehm, Chris Abts, and Sunita Chulani. Software Development Cost Estimation Approaches - A Survey. *Annals of Software Engineering*, 10(1/4):177–205, 2000. doi:10.1023/a:1018991717352. (cited on Page 21)

Barry W. Boehm, A. Winsor Brown, Ray Madachy, and Ye Yang. A Software Product Line Life Cycle Cost Estimation Model. In *International Symposium on Empirical Software Engineering (ESE)*, pages 156–164. IEEE, 2004. doi:10.1109/isese.2004.1334903. (cited on Page 22 and 23)

Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems. In *International Symposium on the Foundations of Software Engineering (FSE)*, pages 109–120. ACM, 2016. doi:10.1145/2950290.2950325. (cited on Page 25, 36, 43, and 47)

Denise Bombonatti, Miguel Goulão, and Ana Moreira. Synergies and Tradeoffs in Software Reuse – A Systematic Mapping Study. *Software: Practice and Experience*, 47(7):943–957, 2016. doi:10.1002/spe.2416. (cited on Page 32 and 37)

Jan Bosch. Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In *International Software Product Line Conference (SPLC)*, pages 257–271. Springer, 2002. doi:10.1007/3-540-45652-x_16. (cited on Page 177)

Jan Bosch, editor. *Continuous Software Engineering*. Springer, 2014. doi:10.1007/978-3-319-11283-1. (cited on Page 11, 147, and 159)

Jan Bosch and Petra Bosch-Sijtsema. From Integration to Composition: On the Impact of Software Product Lines, Global Development and Ecosystems. *Journal of Systems and Software*, 83(1):67–76, 2010. doi:10.1016/j.jss.2009.06.051. (cited on Page 1, 2, 10, and 11)

Michael F. Bosu and Stephen G. Macdonell. Experience: Quality Benchmarking of Datasets Used in Software Effort Estimation. *Journal of Data and Information Quality*, 11(4): 19:1–38, 2019. doi:10.1145/3328746. (cited on Page 4)

Ekaterina Boutkova. Experience with Variability Management in Requirement Specifications. In *International Software Product Line Conference (SPLC)*, pages 303–312. IEEE, 2011. doi:10.1109/splc.2011.35. (cited on Page 165 and 168)

Gregory M. Bowen. An Organized, Devoted, Project-Wide Reuse Effort. *Ada Letters*, XII (1):43–52, 1992. doi:10.1145/141454.141457. (cited on Page 36, 41, and 47)

O. Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from Applying the Systematic Literature Review Process within the Software Engineering Domain. *Journal of Systems and Software*, 80(4):571–583, 2007. doi:10.1016/j.jss.2006.07.009. (cited on Page 67)

Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Frequently Asked Questions in Bug Reports. Technical Report 2009-03-23T16:05:29Z, University of Calgary, 2009. (cited on Page 68)

Lisa Brownsword and Paul C. Clements. A Case Study in Successful Product Line Development. Technical Report CMU/SEI-96-TR-016, Carnegie Mellon University, 1996. (cited on Page 36 and 42)

Ross Buhrdorf, Dale Churchett, and Charles W. Krueger. Salion's Experience with a Reactive Software Product Line Approach. In *International Workshop on Software Product-Family Engineering (PFE)*, pages 317–322. Springer, 2003. doi:10.1007/978-3-540-24667-1_24. (cited on Page 36 and 42)

John Businge, Moses Openja, Sarah Nadi, Engineer Bainomugisha, and Thorsten Berger. Clone-Based Variability Management in the Android Ecosystem. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 625–634. IEEE, 2018. doi:10.1109/icsme.2018.00072. (cited on Page 1, 63, and 105)

Marimuthu C and K. Chandrasekaran. Systematic Studies in Software Product Lines: A Tertiary Study. In *International Systems and Software Product Line Conference (SPLC)*, pages 143–152. ACM, 2017. doi:10.1145/3106195.3106212. (cited on Page 3, 32, and 149)

Rafael Capilla and Jan Bosch. Dynamic Variability Management Supporting Operational Modes of a Power Plant Product Line. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 49–56. ACM, 2016. doi:10.1145/2866614.2866621. (cited on Page 153 and 160)

Sofia Charalampidou, Apostolos Ampatzoglou, Evangelos Karountzos, and Paris Avgeriou. Empirical Studies on Software Traceability: A Mapping Study. *Journal of Software: Evolution and Process*, 33(2):e2294:1–28, 2021. doi:10.1002/smr.2294. (cited on Page 108 and 112)

Oliver Charles, Markus Schalk, and Steffen Thiel. Kostenmodelle für Softwareproduktlinien. *Informatik-Spektrum*, 34(4):377–390, 2011. doi:10.1007/s00287-010-0478-7. (cited on Page 21, 22, and 23)

Jaime Chavarriaga, Carlos Rangel, Carlos Noguera, Rubby Casallas, and Viviane Jonckers. Using Multiple Feature Models to Specify Configuration Options for Electrical Transformers: An Experience Report. In *International Software Product Line Conference (SPLC)*, pages 216–224. ACM, 2015. doi:10.1145/2791060.2791091. (cited on Page 165 and 169)

Lianping Chen and Muhammad A. Babar. A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. *Information and Software Technology*, 53(4):344–362, 2011. doi:10.1016/j.infsof.2010.12.006. (cited on Page 2 and 14)

Wai T. Cheung, Sukyoung Ryu, and Sunghun Kim. Development Nature Matters: An Empirical Study of Code Clones in JavaScript Applications. *Empirical Software Engineering*, 21(2):517–564, 2016. doi:10.1007/s10664-015-9368-6. (cited on Page 2)

Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a Feature: A Requirements Engineering Perspective. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 16–30. Springer, 2008. doi:10.1007/978-3-540-78743-3_2. (cited on Page 1, 11, and 62)

Jane Cleland-Huang, Orlena C. Z. Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. Software Traceability: Trends and Future Directions. In *Conference on the Future of Software Engineering (FOSE)*, pages 55–69. ACM, 2014. doi:10.1145/2593882.2593891. (cited on Page 108)

Paul C. Clements and John Bergey. The U.S. Army's Common Avionics Architecture System (CAAS) Product Line: A Case Study. Technical Report CMU/SEI-2005-TR-019, Carnegie Mellon University, 2005. (cited on Page 36)

Paul C. Clements and Charles W. Krueger. Point/Counterpoint: Being Proactive Pays Off / Eliminating the Adoption Barrier. *IEEE Software*, 19(4):28–30, 2002. doi:10.1109/ms.2002.1020283. (cited on Page 2, 10, 11, 26, 27, 149, 154, and 160)

Paul C. Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. (cited on Page 2, 36, 41, 42, 44, 46, 47, 56, and 112)

Paul C. Clements and Linda M. Northrop. Salion, Inc.: A Software Product Line Case Study. Technical Report CMU/SEI-2002-TR-038, Carnegie Mellon University, 2002. (cited on Page 36 and 42)

Paul C. Clements, Sholom G. Cohen, Patrick Donohoe, and Linda M. Northrop. Control Channel Toolkit: A Software Product Line Case Study. Technical Report CMU/SEI-2001-TR-030, Carnegie Mellon University, 2001. (cited on Page 36, 44, and 46)

Paul C. Clements, John D. McGregor, and Sholom G. Cohen. The Structured Intuitive Model for Product Line Economics (SIMPLE). Technical Report CMU/SEI-2005-TR-003, Carnegie Mellon University, 2005. (cited on Page 20, 22, 23, and 24)

Paul C. Clements, Susan P. Gregg, Charles W. Krueger, Jeremy Lanman, Jorge Rivera, Rick Scharadin, James T. Shepherd, and Andrew J. Winkler. Second Generation Product Line Engineering Takes Hold in the DoD. *CrossTalk: The Journal of Defense Software Engineering*, 27(1):12–18, 2014. (cited on Page 36, 42, and 46)

Norman Cliff. Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions. *Psychological Bulletin*, 114(3):494–509, 1993. doi:10.1037/0033-2909.114.3.494. (cited on Page 141)

Gillian Cohen and Martin A. Conway, editors. *Memory in the Real World*. Psychology Press, 2007. doi:10.4324/9780203934852. (cited on Page 69)

Sholom G. Cohen. Predicting When Product Line Investment Pays. Technical Report CMU/SEI-2003-TN-017, Carnegie Mellon University, 2003. (cited on Page 22)

Sholom G. Cohen, Jay L. Stanley Jr., A. Spencer Peterson, and Robert W. Krut Jr. Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain. Technical Report CMU/SEI-91-TR-28, Carnegie Mellon University, 1992. (cited on Page 165)

Sholom G. Cohen, Ed Dunn, and Albert Soule. Successful Product Line Development and Sustainment: A DoD Case Study. Technical Report CMU/SEI-2002-TN-018., Carnegie Mellon University, 2002. (cited on Page 36, 41, 42, and 47)

Adrian Colyer and Andrew Clement. Large-Scale AOSD for Middleware. In *International Conference on Aspect-Oriented Software Development (AOSD)*, pages 56–65. ACM, 2004. doi:10.1145/976270.976279. (cited on Page 117)

Alejandro Cortiñas, Miguel R. Luaces, Oscar Pedreira, Ángeles S. Places, and Jennifer Pérez. Web-Based Geographic Information Systems SPLE: Domain Analysis and Experience Report. In *International Systems and Software Product Line Conference (SPLC)*, pages 190–194. ACM, 2017. doi:10.1145/3106195.3106222. (cited on Page 36, 44, and 153)

Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Nematollah Bidokhti. How Bad Can a Bug Get? An Empirical Analysis of Software Failures in the OpenStack Cloud Computing Platform. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 200–211. ACM, 2019. doi:10.1145/3338906.3338916. (cited on Page 25)

Bill Curtis, Marc I. Kellner, and Jim Over. Process Modeling. *Communications of the ACM*, 35(9):75–90, 1992. doi:10.1145/130994.130998. (cited on Page 150)

Krzysztof Czarnecki. Overview of Generative Software Development. In *International Workshop on Unconventional Programming Paradigms (UPP)*, pages 326–341. Springer, 2005. doi:10.1007/11527800_25. (cited on Page 11, 148, and 153)

Krzysztof Czarnecki, Chang H. P. Kim, and Karl T. Kalleberg. Feature Models are Views on Ontologies. In *International Software Product Line Conference (SPLC)*, pages 41–51. IEEE, 2006. doi:10.1109/spline.2006.1691576. (cited on Page 169)

Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 173–182. ACM, 2012. doi:10.1145/2110147.2110167. (cited on Page 2, 14, 162, and 169)

Kostadin Damevski, David Shepherd, and Lori Pollock. A Field Study of how Developers Locate Features in Source Code. *Empirical Software Engineering*, 21(2):724–747, 2016. doi:10.1007/s10664-015-9373-9. (cited on Page 30 and 31)

Robert M. Davison, Maris G. Martinsons, and Ned Kock. Principles of Canonical Action Research. *Information Systems*, 14(1):65–86, 2004. doi:10.1111/j.1365-2575.2004.00162.x. (cited on Page 50 and 177)

Jamel Debbiche, Oskar Lignell, Jacob Krüger, and Thorsten Berger. Migrating Java-Based Apo-Games into a Composition-Based Software Product Line. In *International Systems and Software Product Line Conference (SPLC)*, pages 98–102. ACM, 2019. doi:10.1145/3336294.3342361. (cited on Page 19, 49, 51, and 53)

Florian Deissenboeck, Elmar Juergens, Benjamin Hummel, Stefan Wagner, Benedikt M. y Parareda, and Markus Pizka. Tool Support for Continuous Quality Control. *IEEE Software*, 25(5):60–67, 2008. doi:10.1109/ms.2008.129. (cited on Page 52)

Mahdi Derakhshanmanesh, Joachim Fox, and Jürgen Ebert. Requirements-Driven Incremental Adoption of Variability Management Techniques and Tools: An Industrial Experience Eeport. *Requirements Engineering*, 19(4):333–354, 2014. doi:10.1007/s00766-013-0185-4. (cited on Page 165)

Eric L. Dey. Working with Low Survey Response Rates: The Efficacy of Weighting Adjustments. *Research in Higher Education*, 38(2):215–227, 1997. doi:10.1023/a:1024985704202. (cited on Page 85)

Deepak Dhungana, Paul Grünbacher, Rick Rabiser, and Thomas Neumayer. Structuring the Modeling Space and Supporting Evolution in Software Product Line Engineering. *Journal of Systems and Software*, 83(7):1108–1122, 2010. doi:10.1016/j.jss.2010.02.018. (cited on Page 165 and 171)

Michael Dillon, Jorge Rivera, and Rowland Darbin. A Methodical Approach to Product Line Adoption. In *International Software Product Line Conference (SPLC)*, pages 340–349. ACM, 2014. doi:10.1145/2648511.2648550. (cited on Page 36, 41, 42, 43, and 46)

Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud Computing: Issues and Challenges. In *International Conference on Advanced Information Networking and Applications (AINA)*, pages 27–33. IEEE, 2010. doi:10.1109/aina.2010.187. (cited on Page 160)

Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013. doi:10.1002/smr.567. (cited on Page 4, 20, 28, 29, and 112)

Jose J. Dolado. On the Problem of the Software Cost Function. *Information and Software Technology*, 43(1):61–72, 2001. doi:10.1016/s0950-5849(00)00137-3. (cited on Page 20)

Peter Dorman. *Microeconomics*. Springer, 2014. doi:10.1007/978-3-642-37434-0. (cited on Page 27)

Ekwa Duala-Ekoko and Martin P. Robillard. Asking and Answering Questions about Unfamiliar APIs: An Exploratory Study. In *International Conference on Software Engineering (ICSE)*, pages 266–276. IEEE, 2012. doi:10.1109/icse.2012.6227187. (cited on Page 68)

Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An Exploratory Study of Cloning in Industrial Software Product Lines. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 25–34. IEEE, 2013. doi:10.1109/csmr.2013.13. (cited on Page 1, 9, 36, 39, 41, 42, 46, 49, 147, 159, and 160)

Anh N. Duc, Audris Mockus, Randy Hackbarth, and John Palframan. Forking and Coordination in Multi-Platform Development. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 59:1–10. ACM, 2014. doi:10.1145/2652524.2652546. (cited on Page 36, 41, 43, 46, and 47)

Slawomir Duszynski. Visualizing and Analyzing Software Variability with Bar Diagrams and Occurrence Matrices. In *International Software Product Line Conference (SPLC)*, pages 481–485. Springer, 2010. doi:10.1007/978-3-642-15579-6_41. (cited on Page 20)

Slawomir Duszynski, Jens Knodel, and Martin Becker. Analyzing the Source Code of Multiple Software Variants for Reuse Potential. In *Working Conference on Reverse Engineering (WCRE)*, pages 303–307. IEEE, 2011. doi:10.1109/wcre.2011.44. (cited on Page 2 and 12)

Tore Dybå, Barbara A. Kitchenham, and Magne Jørgensen. Evidence-Based Software Engineering for Practitioners. *IEEE Software*, 22(1):58–65, 2005. doi:10.1109/ms.2005.6. (cited on Page 3)

Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer, 2008. doi:10.1007/978-1-84800-044-5_11. (cited on Page 50)

Hermann Ebbinghaus. *Über das Gedächtnis: Untersuchungen zur experimentellen Psychologie*. Duncker & Humblot, 1885. (cited on Page 82, 83, 87, 89, 90, 91, and 92)

Christof Ebert and Michel Smouts. Tricks and Traps of Initiating a Product Line Concept in Existing Products. In *International Conference on Software Engineering (ICSE)*, pages 520–525. IEEE, 2003. doi:10.1109/icse.2003.1201231. (cited on Page 36, 42, 43, 46, and 47)

Alexander Egyed, Florian Graf, and Paul Grünbacher. Effort and Quality of Recovering Requirements-to-Code Traces: Two Exploratory Experiments. In *International Conference Requirements Engineering (RE)*, pages 221–300. IEEE, 2010. doi:10.1109/re.2010.34. (cited on Page 112)

Wilford J. Eiteman and Glenn E. Guthrie. The Shape of the Average Cost Curve. *The American Economic Review*, 42(5):832–838, 1952. (cited on Page 27)

Emelie Engström and Per Runeson. Software Product Line Testing - A Systematic Mapping Study. *Information and Software Technology*, 53(1):2–13, 2011. doi:10.1016/j.infsof.2010.05.011. (cited on Page 27)

Sina Entekhabi, Anton Solback, Jan-Philipp Steghöfer, and Thorsten Berger. Visualization of Feature Locations with the Tool FeatureDashboard. In *International Systems and Software Product Line Conference (SPLC)*, pages 1–4. ACM, 2019. doi:10.1145/3307630.3342392. (cited on Page 96 and 108)

Ali Erdem, W. Lewis Johnson, and Stacy Marsella. Task Oriented Software Understanding. In *International Conference on Automated Software Engineering (ASE)*, pages 230–239. IEEE, 1998. doi:10.1109/ase.1998.732658. (cited on Page 69 and 73)

Michael D. Ernst, Greg J. Badros, and David Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002. doi:10.1109/tse.2002.1158288. (cited on Page 113)

Davide Falessi, Natalia Juristo, Claes Wohlin, Burak Turhan, Jürgen Münch, Andreas Jedlitschka, and Markku Oivo. Empirical Software Engineering Experts on the Use of Students and Professionals in Experiments. *Empirical Software Engineering*, 23(1): 452–489, 2017. doi:10.1007/s10664-017-9523-3. (cited on Page 120)

David Faust and Chris Verhoef. Software Product Line Migration and Deployment. *Software: Practice and Experience*, 33(10):933–955, 2003. doi:10.1002/spe.530. (cited on Page 36, 41, 42, 43, and 44)

Jean-Marie Favre. Preprocessors from an Abstract Point of View. In *Working Conference on Reverse Engineering (WCRE)*, pages 287–296. IEEE, 1996. doi:10.1109/wcre.1996.558940. (cited on Page 113)

Jean-Marie Favre. Understanding-In-The-Large. In *International Workshop on Program Comprehension (IWPC)*, pages 29–38. IEEE, 1997. doi:10.1109/wpc.1997.601260. (cited on Page 113)

Janet Feigenspan, Maria Papendieck, Christian Kästner, Mathias Frisch, and Raimund Dachselt. FeatureCommander: Colorful #ifdef World. In *International Software Product Line Conference (SPLC)*, pages 48:1–2. ACM, 2011. doi:10.1145/2019136.2019192. (cited on Page 111)

Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring Programming Experience. In *International Conference on Program Comprehension (ICPC)*, pages 73–82. IEEE, 2012. doi:10.1109/icpc.2012.6240511. (cited on Page 79 and 84)

Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empirical Software Engineering*, 18(4):699–745, 2013. doi:10.1007/s10664-012-9208-x. (cited on Page 111, 114, and 121)

Robert Feldt, Lefteris Angelis, Richard Torkar, and Maria Samuelsson. Links Between the Personalities, Views and Attitudes of Software Engineers. *Information and Software Technology*, 52(6):611–624, 2010. doi:10.1016/j.infsof.2010.01.001. (cited on Page 69)

Wolfram Fenske and Sandro Schulze. Code Smells Revisited: A Variability Perspective. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 3–10. ACM, 2015. doi:10.1145/2701319.2701321. (cited on Page 27 and 117)

Wolfram Fenske, Thomas Thüm, and Gunter Saake. A Taxonomy of Software Product Line Reengineering. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 4:1–8. ACM, 2013. doi:10.1145/2556624.2556643. (cited on Page 1, 2, 11, 12, 32, and 149)

Wolfram Fenske, Sandro Schulze, Daniel Meyer, and Gunter Saake. When Code Smells Twice as Much: Metric-Based Detection of Variability-Aware Code Smells. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 171–180. IEEE, 2015. doi:10.1109/scam.2015.7335413. (cited on Page 113 and 134)

Wolfram Fenske, Jens Meinicke, Sandro Schulze, Steffen Schulze, and Gunter Saake. Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 316–326. IEEE, 2017a. doi:10.1109/saner.2017.7884632. (cited on Page 52 and 62)

Wolfram Fenske, Sandro Schulze, and Gunter Saake. How Preprocessor Annotations (Do Not) Affect Maintainability: A Case Study on Change-Proneness. In *International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pages 77–90. ACM, 2017b. doi:10.1145/3136040.3136059. (cited on Page 113 and 117)

Wolfram Fenske, Jacob Krüger, Maria Kanyshkova, and Sandro Schulze. #ifdef Directives and Program Comprehension: The Dilemma between Correctness and Preference. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 255–266. IEEE, 2020. doi:10.1109/ICSME46990.2020.00033. (cited on Page 17, 60, 62, 107, 108, 112, 114, 117, and 133)

Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. Do #ifdefs Influence the Occurrence of Vulnerabilities? An Empirical Study of the Linux Kernel. In *International Systems and Software Product Line Conference (SPLC)*, pages 65–73. ACM, 2016. doi:10.1145/2934466.2934467. (cited on Page 113)

Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 391–400. IEEE, 2014. doi:10.1109/icsme.2014.61. (cited on Page 2, 9, 10, and 160)

Ronald A. Fisher. On the Interpretation of χ^2 from Contingency Tables, and the Calculation of P. *Journal of the Royal Statistical Society*, 85(1):87–94, 1922. doi:10.2307/2340521. (cited on Page 126 and 139)

Brian Fitzgerald and Klaas-Jan Stol. Continuous Software Engineering and Beyond: Trends and Challenges. In *International Workshop on Rapid Continuous Software Engineering (RCoSE)*, pages 1–9. ACM, 2014. doi:10.1145/2593812.2593813. (cited on Page 147 and 159)

Beat Fluri, Michael Würsch, and Harald C. Gall. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. In *Working Conference on*

Reverse Engineering (WCRE), pages 70–79. IEEE, 2007. doi:10.1109/wcre.2007.21. (cited on Page 99 and 129)

Thomas S. Fogdal, Helene Scherrebeck, Juha Kuusela, Martin Becker, and Bo Zhang. Ten Years of Product Line Engineering at Danfoss: Lessons Learned and Way Ahead. In *International Systems and Software Product Line Conference (SPLC)*, pages 252–261. ACM, 2016. doi:10.1145/2934466.2934491. (cited on Page 1, 2, 12, 36, 42, 44, 46, 148, 149, 153, 160, and 165)

William B. Frakes and Kyo C. Kang. Software Reuse Research: Status and Future. *IEEE Transactions on Software Engineering*, 31(7):529–536, 2005. doi:10.1109/tse.2005.85. (cited on Page 9 and 177)

William B. Frakes and Giancarlo Succi. An Industrial Study of Reuse, Quality, and Productivity. *Journal of Systems and Software*, 57(2):99–106, 2001. doi:10.1016/s0164-1212(00)00121-7. (cited on Page 36, 38, and 46)

William B. Frakes and Carol Terry. Software Reuse: Metrics and Models. *ACM Computing Surveys*, 28(2):415–435, 1996. doi:10.1145/234528.234531. (cited on Page 113 and 177)

Gregory A. Fredricks and Roger B. Nelsen. On the Relationship Between Spearman's Rho and Kendall's Tau for Pairs of Continuous Random Variables. *Journal of Statistical Planning and Inference*, 137(7):2143–2150, 2007. doi:10.1016/j.jspi.2006.06.045. (cited on Page 92)

Thomas Fritz and Gail C. Murphy. Using Information Fragments to Answer the Questions Developers Ask. In *International Conference on Software Engineering (ICSE)*, pages 175–184. IEEE, 2010. doi:10.1145/1806799.1806828. (cited on Page 69, 71, and 74)

Thomas Fritz, Gail C. Murphy, and Emily Hill. Does a Programmer's Activity Indicate Knowledge of Code? In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 341–350. ACM, 2007. doi:10.1145/1287624.1287673. (cited on Page 81)

Thomas Fritz, Jingwen Ou, Gail C. Murphy, and Emerson Murphy-Hill. A Degree-of-Knowledge Model to Capture Source Code Familiarity. In *International Conference on Software Engineering (ICSE)*, pages 385–394. ACM, 2010. doi:10.1145/1806799.1806856. (cited on Page 81 and 89)

Cristina Gacek and Michalis Anastasopoules. Implementing Product Line Variabilities. In *Symposium on Software Reusability (SSR)*, pages 109–117. ACM, 2001. doi:10.1145/375212.375269. (cited on Page 2, 15, and 109)

Cristina Gacek, Peter Knauber, Klaus Schmid, and Paul C. Clements. Successful Software Product Line Development in a Small Organization: A Case Study. Technical Report 013.01/E, Fraunhofer IESE, 2001. (cited on Page 26 and 36)

Jesús P. Gaeta and Krzysztof Czarnecki. Modeling Aerospace Systems Product Lines in SysML. In *International Software Product Line Conference (SPLC)*, pages 293–302. ACM, 2015. doi:10.1145/2791060.2791104. (cited on Page 165)

Dharmalingam Ganesan, Dirk Muthig, and Kentaro Yoshimura. Predicting Return-on-Investment for Product Line Generations. In *International Software Product Line Conference (SPLC)*, pages 13–22. IEEE, 2006. doi:10.1109/spline.2006.1691573. (cited on Page 22)

Christopher Ganz and Michael Layes. Modular Turbine Control Software: A Control Software Architecture for the ABB Gas Turbine Family. In *International Workshop on Architectural Reasoning for Embedded Systems (ARES)*, pages 32–38. Springer, 1998. doi:10.1007/3-540-68383-6_6. (cited on Page 36, 46, and 47)

Paul Gazzillo and Robert Grimm. SuperC: Parsing All of C by Taming the Preprocessor. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 323–334. ACM, 2012. doi:10.1145/2254064.2254103. (cited on Page 120)

Yaser Ghanam, Frank Maurer, and Pekka Abrahamsson. Making the Leap to a Software Platform Strategy: Issues and Challenges. *Information and Software Technology*, 54(9): 968–984, 2012. doi:10.1016/j.infsof.2012.03.005. (cited on Page 4, 10, 27, and 103)

Charles Gillan, Peter Kilpatrick, Ivor T. A. Spence, T. John Brown, Rabih Bashroush, and Rachel Gawley. Challenges in the Application of Feature Modelling in Fixed Line Telecommunications. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 141–148. Lero, 2007. (cited on Page 165)

Georgios Gousios, Martin Pinzger, and Arie van Deursen. An Exploratory Study of the Pull-Based Software Development Model. In *International Conference on Software Engineering (ICSE)*, pages 345–355. ACM, 2014. doi:10.1145/2568225.2568260. (cited on Page 2 and 9)

Susan P. Gregg, Rick Scharadin, Eric LeGore, and Paul C. Clements. Lessons from AEGIS: Organizational and Governance Aspects of a Major Product Line in a Multi-Program Environment. In *International Software Product Line Conference (SPLC)*, pages 264–273. ACM, 2014. doi:10.1145/2648511.2648541. (cited on Page 36)

Susan P. Gregg, Rick Scharadin, and Paul C. Clements. The More You Do, the More You Save: The Superlinear Cost Avoidance Effect of Systems Product Line Engineering. In *International Software Product Line Conference (SPLC)*, pages 303–310. ACM, 2015. doi:10.1145/2791060.2791065. (cited on Page 36)

Susan P. Gregg, Denise M. Albert, and Paul C. Clements. Product Line Engineering on the Right Side of the "V". In *International Systems and Software Product Line Conference (SPLC)*, pages 165–174. ACM, 2017. doi:10.1145/3106195.3106219. (cited on Page 153 and 159)

Martin L. Griss. Software Reuse Architecture, Process, and Organization for Business Success. In *Israeli Conference on Computer Systems and Software Engineering (ICCSSE)*, pages 86–89. IEEE, 1997. doi:10.1109/iccsse.1997.599879. (cited on Page 171)

Martin L. Griss, John Favaro, and Massimo d'Alessandro. Integrating Feature Modeling with the RSEB. In *International Conference on Software Reuse (ICSR)*, pages 76–85. IEEE, 1998. doi:10.1109/ICSR.1998.685732. (cited on Page 165 and 171)

MyungJoo Ham and Geunsik Lim. Making Configurable and Unified Platform, Ready for Broader Future Devices. In *International Conference on Software Engineering (ICSE)*, pages 141–150. IEEE, 2019. doi:10.1109/icse-seip.2019.00023. (cited on Page 1 and 36)

Alexander Hars and Shaosong Ou. Working for Free? Motivations for Participating in Open-Source Projects. *International Journal of Electronic Commerce*, 6(3):25–39, 2002. doi:10.1080/10864415.2002.11044241. (cited on Page 92)

Jan Hauke and Tomasz Kossowski. Comparison of Values of Pearson's and Spearman's Correlation Coefficients on the Same Sets of Data. *Quaestiones Geographicae*, 30(2): 87–93, 2011. doi:10.2478/v10117-011-0021-1. (cited on Page 92)

Kengo Hayashi and Mikio Aoyama. A Multiple Product Line Development Method Based on Variability Structure Analysis. In *International Systems and Software Product Line Conference (SPLC)*, pages 160–169. ACM, 2018. doi:10.1145/3233027.3233048. (cited on Page 153)

Kengo Hayashi, Mikio Aoyama, and Keiji Kobata. Agile Tames Product Line Variability: An Agile Development Method for Multiple Product Lines of Automotive Software Systems. In *International Systems and Software Product Line Conference (SPLC)*, pages 180–189. ACM, 2017. doi:10.1145/3106195.3106221. (cited on Page 153 and 160)

Fred J. Heemstra. Software Cost Estimation. *Information and Software Technology*, 34(10): 627–639, 1992. ISSN 0950-5849. doi:10.1016/0950-5849(92)90068-z. (cited on Page 4, 19, 20, 21, and 33)

Andreas Hein, Michael Schlick, and Renato Vinga-Martins. Applying Feature Models in Industrial Settings. In *International Software Product Line Conference (SPLC)*, pages 47–70. Springer, 2000. doi:10.1007/978-1-4615-4339-8_3. (cited on Page 165, 169, and 171)

Emmanuel Henry and Benoît Faller. Large-Scale Industrial Reuse to Reduce Cost and Cycle Time. *IEEE Software*, 12(5):47–53, 1995. doi:10.1109/52.406756. (cited on Page 36, 38, 41, and 46)

Ruben Heradio, David Fernandez-Amoros, Luis Torre-Cubillo, and Alberto Perez Garcia-Plaza. Improving the Accuracy of COPLIMO to Estimate the Payoff of a Software Product Line. *Expert Systems with Applications*, 39(9):7919–7928, 2012. doi:10.1016/j.eswa.2012.01.109. (cited on Page 22)

Ruben Heradio, David Fernandez-Amoros, Jose A. Cerrada, and Ismael Abad. A Literature Review on Feature Diagram Product Counting And its Usage in Software Product Line Economic Models. *International Journal of Software Engineering and Knowledge Engineering*, 23(8):1177–1204, 2013. doi:10.1142/s0218194013500368. (cited on Page 21 and 22)

Ruben Heradio, Hector Perez-Morago, David Fernandez-Amoros, Francisco Javier Cabrerizo, and Enrique Herrera-Viedma. A Bibliometric Analysis of 20 Years of Research on Software Product Lines. *Information and Software Technology*, 72:1–15, 2016. doi:10.1016/j.infsof.2015.11.004. (cited on Page 3, 23, and 32)

Ruben Heradio, David Fernández-Amorós, Carlos Cerrada, Francisco J. Cabrerizo, and Enrique Herrera-Viedma. Integration of Economic Models for Software Product Lines by Means of a Common Lexicon. In *International Conference on New Trends in Intelligent Software Methodologies, Tools and Techniques (SoMeT)*, pages 31–47. IOS, 2018. doi:10.3233/978-1-61499-900-3-31. (cited on Page 21 and 22)

Agneta Herlitz, Lars-Göran Nilsson, and Lars Bäckman. Gender Differences in Episodic Memory. *Memory & Cognition*, 25(6):801–811, 1997. doi:10.3758/bf03211324. (cited on Page 92)

Guido Hertel, Sven Niedner, and Stefanie Herrmann. Motivation of Software Developers in Open Source Projects: An Internet-Based Survey of Contributors to the Linux Kernel.

Research Policy, 32(7):1159–1177, 2003. doi:10.1016/s0048-7333(03)00047-7. (cited on Page 92)

William A. Hetrick, Charles W. Krueger, and Joseph G. Moore. Incremental Return on Incremental Investment: Engenio's Transition to Software Product Line Practice. In *International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 798–804. ACM, 2006. doi:10.1145/1176617.1176726. (cited on Page 36, 42, 43, 46, and 160)

Peter Hofman, Tobias Stenzel, Thomas Pohley, Michael Kircher, and Andreas Bermann. Domain Specific Feature Modeling for Software Product Lines. In *International Software Product Line Conference (SPLC)*, pages 229–238. ACM, 2012. doi:10.1145/2362536.2362568. (cited on Page 165 and 171)

Uwe D. C. Hohenstein and Michael C. Jäger. Using Aspect-Orientation in Industrial Projects: Appreciated or Damned? In *International Conference on Aspect-Oriented Software Development (AOSD)*, pages 213–222. ACM, 2009. doi:10.1145/1509239.1509268. (cited on Page 117)

Gerald Holl, Paul Grünbacher, and Rick Rabiser. A Systematic Review and an Expert Survey on Capabilities Supporting Multi Product Lines. *Information and Software Technology*, 54(8):828–852, 2012. doi:10.1016/j.infsof.2012.02.002. (cited on Page 181)

Sture Holm. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979. URL https://www.jstor.org/stable/4615733. (cited on Page 122)

Reid Holmes and Robert J. Walker. Systematizing Pragmatic Software Reuse. *ACM Transactions on Software Engineering and Methodology*, 21(4):20:1–44, 2012. doi:10.1145/2377656.2377657. (cited on Page 1 and 9)

Helena Holmström Olsson, Hiva Alahyari, and Jan Bosch. Climbing the "Stairway to Heaven" – A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software. In *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 392–399. IEEE, 2012. doi:10.1109/seaa.2012.54. (cited on Page 10 and 103)

Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes. Software Product Line Engineering: A Practical Experience. In *International Systems and Software Product Line Conference (SPLC)*, pages 164–176. ACM, 2019. doi:10.1145/3336294.3336304. (cited on Page 2, 61, and 153)

Siw E. Hove and Bente Anda. Experiences from Conducting Semi-Structured Interviews in Empirical Software Engineering Research. In *International Software Metrics Symposium (METRICS)*, pages 23:1–10. IEEE, 2005. doi:10.1109/metrics.2005.24. (cited on Page 176)

Arnaud Hubaux, Patrick Heymans, and David Benavides. Variability Modeling Challenges from the Trenches of an Open Source Product Line Re-Engineering Project. In *International Software Product Line Conference (SPLC)*, pages 55–64. IEEE, 2008. doi:10.1109/splc.2008.39. (cited on Page 165, 169, and 172)

Arnaud Hubaux, Andreas Classen, Marcílio Mendonça, and Patrick Heymans. A Preliminary Review on the Application of Feature Diagrams in Practice. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 53–59. University of Duisburg-Essen, 2010. (cited on Page 165 and 168)

Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, Dirk Deridder, and Ebrahim K. Abbasi. Supporting Multiple Perspectives in Feature-Based Configuration. *International Journal on Software & Systems Modeling*, 12(3):641–663, 2011. doi:10.1007/s10270-011-0220-1. (cited on Page 175)

Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. A User Survey of Configuration Challenges in Linux and eCos. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 149–155. ACM, 2012. doi:10.1145/2110147.2110164. (cited on Page 18)

Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley, 2010. (cited on Page 18 and 159)

Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering*, 21(2):449–482, 2016. doi:10.1007/s10664-015-9360-1. (cited on Page 16, 49, 105, 113, 120, 132, and 144)

Takahiro Iida, Masahiro Matsubara, Kentaro Yoshimura, Hideyuki Kojima, and Kimio Nishino. PLE for Automotive Braking System with Management of Impacts from Equipment Interactions. In *International Systems and Software Product Line Conference (SPLC)*, pages 232–241. ACM, 2016. doi:10.1145/2934466.2934490. (cited on Page 153 and 160)

Hoh P. In, Jongmoon Baik, Sangsoo Kim, Ye Yang, and Barry W. Boehm. A Quality-Based Cost Estimation Model for the Product Line Life Cycle. *Communications of the ACM*, 49(12):85–88, 2006. doi:10.1145/1183236.1183273. (cited on Page 22)

Angelo J. Incorvaia, Alan M. Davis, and Richard E. Fairley. Case Studies in Software Reuse. In *Annual Computer Software and Applications Conference (COMPSAC)*, pages 301–306. IEEE, 1990. doi:10.1109/cmpsac.1990.139373. (cited on Page 36, 38, and 41)

Takashi Iwasaki, Makoto Uchiba, Jun Ohtsuka, Koji Hachiya, Tsuneo Nakanishi, Kenji Hisazumi, and Akira Fukuda. An Experience Report of Introducing Product Line Engineering across the Board. In *International Software Product Line Conference (SPLC)*, pages 255–258. Lancaster University, 2010. (cited on Page 165)

Khaled Jaber, Bonita Sharif, and Chang Liu. A Study on the Effect of Traceability Links in Software Maintenance. *IEEE Access*, 1:726–741, 2013. doi:10.1109/access.2013.2286822. (cited on Page 112)

Mohamad Y. Jaber and Sverker Sikström. A Numerical Comparison of Three Potential Learning and Forgetting Models. *International Journal of Production Economics*, 92(3): 281–294, 2004a. doi:10.1016/j.ijpe.2003.10.019. (cited on Page 83)

Mohamad Y. Jaber and Sverker Sikström. A Note on "An Empirical Comparison of Forgetting Models". *IEEE Transactions on Engineering Management*, 51(2):233–234, 2004b. doi:10.1109/tem.2004.826017. (cited on Page 83)

Slinger Jansen, Sjaak Brinkkemper, Ivo Hunink, and Cetin Demir. Pragmatic and Opportunistic Reuse in Innovative Start-Up Companies. *IEEE Software*, 25(6):42–49, 2008. doi:10.1109/ms.2008.155. (cited on Page 10, 36, 38, 46, and 47)

Paul Jensen. Experiences with Product Line Development of Multi-Discipline Analysis Software at Overwatch Textron Systems. In *International Software Product Line Conference (SPLC)*, pages 35–43. IEEE, 2007. doi:10.1109/spline.2007.25. (cited on Page 36 and 47)

Paul Jensen. Experiences With Software Product Line Development. *CrossTalk: The Journal of Defense Software Engineering*, 22(1):11–14, 2009. (cited on Page 36 and 47)

Hans P. Jepsen and Danilo Beuche. Running a Software Product Line — Standing Still Is Going Backwards. In *International Software Product Line Conference (SPLC)*, pages 101–110. ACM, 2009. (cited on Page 148)

Hans P. Jepsen and Flemming Nielsen. A Two-Part Architectural Model as Basis for Frequency Converter Product Families. In *International Workshop on Software Architectures for Product Families (IW-SAPF)*, pages 30–38. Springer, 2000. doi:10.1007/978-3-540-44542-5_4. (cited on Page 148)

Hans P. Jepsen, Jan G. Dall, and Danilo Beuche. Minimally Invasive Migration to Software Product Lines. In *International Software Product Line Conference (SPLC)*, pages 203–211. IEEE, 2007. doi:10.1109/spline.2007.30. (cited on Page 36, 43, 46, and 148)

Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. Maintaining Feature Traceability with Embedded Annotations. In *International Software Product Line Conference (SPLC)*, pages 61–70. ACM, 2015. doi:10.1145/2791060.2791107. (cited on Page 5, 28, 96, and 104)

Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011. doi:10.1109/tse.2010.62. (cited on Page 124)

Howell Jordan, Jacek Rosik, Sebastian Herold, Goetz Botterweck, and Jim Buckley. Manually Locating Features in Industrial Source Code: The Search Actions of Software Nomads. In *International Conference on Program Comprehension (ICPC)*, pages 174–177. IEEE, 2015. doi:10.1109/icpc.2015.26. (cited on Page 30)

Magne Jørgensen. A Review of Studies on Expert Estimation of Software Development Effort. *Journal of Systems and Software*, 70(1-2):37–60, 2004. doi:10.1016/s0164-1212(02)00156-5. (cited on Page 23 and 33)

Magne Jørgensen. Forecasting of Software Development Work Effort: Evidence on Expert Judgment and Formal Models. *International Journal of Forecasting*, 23(3):449–462, 2007. doi:10.1016/j.ijforecast.2007.05.008. (cited on Page 23)

Magne Jørgensen. What We Do and Don't Know about Software Development Effort Estimation. *IEEE Software*, 31(2):37–40, 2014. doi:10.1109/ms.2014.49. (cited on Page 4, 20, 21, 23, 33, and 48)

Magne Jørgensen and Barry W. Boehm. Software Development Effort Estimation: Formal Models or Expert Judgment? *IEEE Software*, 26(2):14–19, 2009. doi:10.1109/ms.2009.47. (cited on Page 20, 21, and 33)

Magne Jørgensen and Kjetil Moløkken-Østvold. Reasons for Software Effort Estimation Error: Impact of Respondent Role, Information Collection Approach, and Data Analysis Method. *IEEE Transactions on Software Engineering*, 30(12):993–1007, 2004. doi:10.1109/tse.2004.103. (cited on Page 33)

Magne Jørgensen and Martin Shepperd. A Systematic Review of Software Development Cost Estimation Studies. *IEEE Transactions on Software Engineering*, 33(1):33–53, 2007. doi:10.1109/tse.2007.256943. (cited on Page 21 and 23)

Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. CloneDetective - A Workbench for Clone Detection Research. In *International Conference on Software Engineering (ICSE)*, pages 603–606. IEEE, 2009. doi:10.1109/icse.2009.5070566. (cited on Page 52)

Mert E.in Kalender, Eray Tüzün, and Bedir Tekinerdogan. Decision Support for Adopting SPLE with Transit-PL. In *International Software Product Line Conference (SPLC)*, pages 150–153. ACM, 2013. doi:10.1145/2499777.2499783. (cited on Page 177)

Keumseok Kang and Jungpil Hahn. Learning and Forgetting Curves in Software Development: Does Type of Knowledge Matter? In *International Conference on Information Systems (ICIS)*, pages 194:1–15. AIS, 2009. (cited on Page 4, 69, 81, and 130)

Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, 1990. (cited on Page 14, 160, 162, and 176)

Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering*, 5:143–168, 1998. doi:10.1023/a:1018980625587. (cited on Page 165, 171, and 176)

Kyo C. Kang, Sajoong Kim, Jaejoon Lee, and Kwanwoo Lee. Feature-Oriented Engineering of PBX Software for Adaptability and Reuseability. *Software: Practice and Experience*, 29(10):875–896, 1999. doi:10.1002/(SICI)1097-024X(199908)29:10<875::AID-SPE262>3.0.CO;2-W. (cited on Page 165 and 171)

Kyo C. Kang, Jaejoon Lee, and Patrick Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, 19(4):58–65, 2002. doi:10.1109/ms.2002.1020288. (cited on Page 148, 153, and 165)

Kyo C. Kang, Kwanwoo Lee, Jaejoon Lee, and Sajoong Kim. Feature Oriented Product Line Software Engineering: Principles and Guidelines. In *Domain Oriented Systems Development: Perspectives and Practices*, pages 29–46. Taylor & Francis, 2003. (cited on Page 165)

Cory J. Kapser and Michael W. Godfrey. "Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software. *Empirical Software Engineering*, 13(6):645–692, 2008. doi:10.1007/s10664-008-9076-6. (cited on Page 36, 41, 43, and 142)

Christian Kästner and Sven Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, pages 35–40, 2008. (cited on Page 109 and 111)

Christian Kästner and Sven Apel. Feature-Oriented Software Development: A Short Tutorial on Feature-Oriented Programming, Virtual Separation of Concerns, and Variability-Aware Analysis. In *International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 346–382. Springer, 2013. doi:10.1007/978-3-642-35992-7_10. (cited on Page 110)

Christian Kästner, Sven Apel, and Don Batory. A Case Study Implementing Features Using AspectJ. In *International Software Product Line Conference (SPLC)*, pages 223–232. IEEE, 2007. doi:10.1109/spline.2007.12. (cited on Page 52)

Christian Kästner, Sven Apel, and Martin Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pages 157–166. ACM, 2009. doi:10.1145/1621607.1621632. (cited on Page 111)

Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 805–824. ACM, 2011. doi:10.1145/2048066.2048128. (cited on Page 120)

Christian Kästner, Alexander Dreiling, and Klaus Ostermann. Variability Mining: Consistent Semi-Automatic Detection of Product-Line Features. *IEEE Transactions on Software Engineering*, 40(1):67–82, 2014. doi:10.1109/tse.2013.45. (cited on Page 28)

Maurice G. Kendall. A New Measure of Rank Correlation. *Biometrika*, 30(1/2):81–93, 1938. doi:10.2307/2332226. (cited on Page 79)

Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. TypeChef: Toward Type Checking #ifdef Variability in C. In *International Workshop on Feature-Oriented Software Development (FOSD)*, pages 25–32. ACM, 2010. doi:10.1145/1868688.1868693. (cited on Page 120)

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978. (cited on Page 16)

Mahvish Khurum, Tony Gorschek, and Kent Pettersson. Systematic Review of Solutions Proposed for Product Line Economics. In *International Software Product Line Conference (SPLC)*, pages 277–284. Lero, 2008. (cited on Page 21, 22, and 23)

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242. Springer, 1997. doi:10.1007/bfb0053381. (cited on Page 115)

Barbara A. Kitchenham, Tore Dybå, and Magne Jørgensen. Evidence-Based Software Engineering. In *International Conference on Software Engineering (ICSE)*, pages 273–281. IEEE, 2004. doi:10.1109/icse.2004.1317449. (cited on Page 3)

Barbara A. Kitchenham, David Budgen, and O. Pearl Brereton. *Evidence-Based Software Engineering and Systematic Reviews*. CRC Press, 2015. doi:10.1201/b19467. (cited on Page 3, 36, 38, 67, 81, 151, and 163)

Peter Knauber, Jesus Bermejo, Günter Böckle, Julio C. S. do Prado Leite, Frank J. van der Linden, Linda M. Northrop, Michael Stark, and David M. Weiss. Quantifying Product Line Benefits. In *International Workshop on Software Product-Family Engineering (PFE)*, pages 155–163. Springer, 2002. doi:10.1007/3-540-47833-7_15. (cited on Page 4, 12, and 33)

Amy J. Ko, Robert DeLine, and Gina Venolia. Information Needs in Collocated Software Development Teams. In *International Conference on Software Engineering (ICSE)*, pages 344–353. IEEE, 2007. doi:10.1109/icse.2007.45. (cited on Page 69 and 75)

Amy J. Ko, Thomas D. LaToza, and Margaret M. Burnett. A Practical Guide to Controlled Experiments of Software Engineering Tools with Human Participants. *Empirical Software Engineering*, 20(1):110–141, 2015. doi:10.1007/s10664-013-9279-3. (cited on Page 79 and 84)

Ronny Kolb, Isabel John, Jens Knodel, Dirk Muthig, Uwe Haury, and Gerald Meier. Experiences with Product Line Development of Embedded Systems at Testo AG. In *International Software Product Line Conference (SPLC)*, pages 172–181. IEEE, 2006. doi:10.1109/spline.2006.1691589. (cited on Page 36, 44, 46, and 47)

Heiko Koziolek, Thomas Goldschmidt, Thijmen de Gooijer, Dominik Domis, Stephan Sehestedt, Thomas Gamer, and Markus Aleksy. Assessing Software Product Line Potential: An Exploratory Industrial Case Study. *Empirical Software Engineering*, 21(2):411–448, 2016. doi:10.1007/s10664-014-9358-0. (cited on Page 4, 20, 23, 24, 153, and 177)

Sebastian Krieter. Enabling Efficient Automated Configuration Generation and Management. In *International Systems and Software Product Line Conference (SPLC)*, pages 215–221. ACM, 2019. doi:10.1145/3307630.3342705. (cited on Page 18)

Sebastian Krieter, Jacob Krüger, and Thomas Leich. Don't Worry About it: Managing Variability On-The-Fly. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 19–26. ACM, 2018a. doi:10.1145/3168365.3170426. (cited on Page 107, 108, and 109)

Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. Propagating Configuration Decisions with Modal Implication Graphs. In *International Conference on Software Engineering (ICSE)*, pages 898–909. ACM, 2018b. doi:10.1145/3180155.3180159. (cited on Page 18)

Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, 1992. doi:10.1145/130844.130856. (cited on Page 1, 3, and 9)

Charles W. Krueger. Easing the Transition to Software Mass Customization. In *International Workshop on Software Product-Family Engineering (PFE)*, pages 282–293. Springer, 2002. doi:10.1007/3-540-47833-7_25. (cited on Page 2, 12, and 155)

Charles W. Krueger. BigLever Software Gears and the 3-Tiered SPL Methodology. In *International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 844–845. ACM, 2007. doi:10.1145/1297846.1297918. (cited on Page 2 and 61)

Charles W. Krueger and Paul C. Clements. Systems and Software Product Line Engineering. In *Encyclopedia of Software Engineering*, pages 1–14. Taylor & Francis, 2013. (cited on Page 37)

Charles W. Krueger, Dale Churchett, and Ross Buhrdorf. HomeAway's Transition to Software Product Line Practice: Engineering and Business Results in 60 Days. In *International Software Product Line Conference (SPLC)*, pages 297–306. IEEE, 2008. doi:10.1109/splc.2008.36. (cited on Page 36 and 44)

Jacob Krüger. A Cost Estimation Model for the Extractive Software-Product-Line Approach. Master's thesis, Otto-von-Guericke University Magdeburg, 2016. (cited on Page 4, 20, 21, 22, 23, 24, 26, 149, and 150)

Jacob Krüger. Lost in Source Code: Physically Separating Features in Legacy Systems. In *International Conference on Software Engineering (ICSE)*, pages 461–462. IEEE, 2017. doi:10.1109/icse-c.2017.46. (cited on Page 1, 3, 107, and 151)

Jacob Krüger. When to Extract Features: Towards a Recommender System. In *International Conference on Software Engineering (ICSE)*, pages 518–520. ACM, 2018a. doi:10.1145/3183440.3190328. (cited on Page 1, 3, 19, 20, 28, and 32)

Jacob Krüger. Separation of Concerns: Experiences of the Crowd. In *Symposium on Applied Computing (SAC)*, pages 2076–2077. ACM, 2018b. doi:10.1145/3167132.3167458. (cited on Page 1, 3, 62, 107, 108, 113, and 115)

Jacob Krüger. Tackling Knowledge Needs during Software Evolution. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1244–1246. ACM, 2019a. doi:10.1145/3338906.3342505. (cited on Page 1, 3, 4, 5, and 65)

Jacob Krüger. Are You Talking about Software Product Lines? An Analysis of Developer Communities. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 11:1–9. ACM, 2019b. doi:10.1145/3302333.3302348. (cited on Page 2, 10, 12, 148, and 159)

Jacob Krüger and Thorsten Berger. Activities and Costs of Re-Engineering Cloned Variants Into an Integrated Platform. In *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 21:1–10. ACM, 2020a. doi:10.1145/3377024.3377044. (cited on Page 2, 3, 4, 5, 10, 19, 23, 49, 53, 147, 148, 151, and 153)

Jacob Krüger and Thorsten Berger. An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 432–444. ACM, 2020b. doi:10.1145/3368089.3409684. (cited on Page 1, 2, 3, 4, 5, 6, 9, 10, 19, 23, 32, 51, 66, 147, 148, 150, 151, 153, 158, 159, 160, and 177)

Jacob Krüger and Regina Hebig. What Developers (Care to) Recall: An Interview Survey on Smaller Systems. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 46–57. IEEE, 2020. doi:10.1109/ICSME46990.2020.00015. (cited on Page 23, 28, 65, 66, and 130)

Jacob Krüger, Wolfram Fenske, Jens Meinicke, Thomas Leich, and Gunter Saake. Extracting Software Product Lines: A Cost Estimation Perspective. In *International Systems and Software Product Line Conference (SPLC)*, pages 354–361. ACM, 2016a. doi:10.1145/2934466.2962731. (cited on Page 2, 10, 19, 20, 23, 24, 26, 27, 149, and 160)

Jacob Krüger, Ivonne Schröter, Andy Kenner, Christopher Kruczek, and Thomas Leich. FeatureCoPP: Compositional Annotations. In *International Workshop on Feature-Oriented Software Development (FOSD)*, pages 74–84. ACM, 2016b. doi:10.1145/3001867.3001876. (cited on Page 107, 108, 109, and 111)

Jacob Krüger, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. Finding Lost Features in Cloned Systems. In *International Systems and Software Product Line Conference (SPLC)*, pages 65–72. ACM, 2017a. doi:10.1145/3109729.3109736. (cited on Page 19, 49, 51, 52, 65, 147, 153, and 160)

Jacob Krüger, Sebastian Nielebock, Sebastian Krieter, Christian Diedrich, Thomas Leich, Gunter Saake, Sebastian Zug, and Frank Ortmeier. Beyond Software Product Lines: Variability Modeling in Cyber-Physical Systems. In *International Systems and Software Product Line Conference (SPLC)*, pages 237–241. ACM, 2017b. doi:10.1145/3106195.3106217. (cited on Page 160)

Jacob Krüger, Ivonne Schröter, Andy Kenner, and Thomas Leich. Empirical Studies in Question-Answering Systems: A Discussion. In *International Workshop on Conducting Empirical Studies in Industry (CESI)*, pages 23–26. IEEE, 2017c. doi:10.1109/cesi.2017.6. (cited on Page 115 and 120)

Jacob Krüger, Wolfram Fenske, Thomas Thüm, Dirk Aporius, Gunter Saake, and Thomas Leich. Apo-Games - A Case Study for Reverse Engineering Variability from Cloned Java Variants. In *International Systems and Software Product Line Conference (SPLC)*, pages 251–256. ACM, 2018a. doi:10.1145/3233027.3236403. (cited on Page 1, 19, and 51)

Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. Towards a Better Understanding of Software Features and Their Characteristics. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 105–112. ACM, 2018b. doi:10.1145/3168365.3168371. (cited on Page 1, 28, 30, 39, 65, 93, and 113)

Jacob Krüger, Kai Ludwig, Bernhard Zimmermann, and Thomas Leich. Physical Separation of Features: A Survey with CPP Developers. In *Symposium on Applied Computing (SAC)*, pages 2042–2049. ACM, 2018c. doi:10.1145/3167132.3167351. (cited on Page 107, 108, 113, and 115)

Jacob Krüger, Marcus Pinnecke, Andy Kenner, Christopher Kruczek, Fabian Benduhn, Thomas Leich, and Gunter Saake. Composing Annotations Without Regret? Practical Experiences Using FeatureC. *Software: Practice and Experience*, 48(3):402–427, 2018d. doi:10.1002/spe.2525. (cited on Page 19, 49, 51, 52, 55, 107, and 111)

Jacob Krüger, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. Do You Remember This Source Code? In *International Conference on Software Engineering (ICSE)*, pages 764–775. ACM, 2018e. doi:10.1145/3180155.3180215. (cited on Page 4, 7, 28, 65, 69, 82, and 130)

Jacob Krüger, Thorsten Berger, and Thomas Leich. Features and How to Find Them - A Survey of Manual Feature Location. In *Software Engineering for Variability Intensive Systems*, pages 153–172. CRC Press, 2019a. doi:10.1201/9780429022067-9. (cited on Page 4, 19, 20, 28, 112, 128, 148, 150, and 153)

Jacob Krüger, Gül Çalıklı, Thorsten Berger, Thomas Leich, and Gunter Saake. Effects of Explicit Feature Traceability on Program Comprehension. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 338–349. ACM, 2019b. doi:10.1145/3338906.3338968. (cited on Page 5, 62, 96, 107, 108, 112, 114, and 121)

Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. Where is My Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software*, 152:239–253, 2019c. doi:10.1016/j.jss.2019.01.057. (cited on Page 2, 9, 11, 30, 39, 65, 93, 110, 112, 147, 148, 151, 153, and 160)

Jacob Krüger, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. Program Comprehension and Developers' Memory. In *INFORMATIK*, pages 99–100. GI, 2019d. doi:10.18420/inf2019_13. (cited on Page 65)

Jacob Krüger, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. Understanding How Programmers Forget. In *Software Engineering and Software Management (SE/SWM)*, pages 85–86. GI, 2019e. doi:10.18420/se2019-23. (cited on Page 65)

Jacob Krüger, Sofia Ananieva, Lea Gerling, and Eric Walkingshaw. Third International Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution 2020). In *International Systems and Software Product Line Conference (SPLC)*, page 34:1. ACM, 2020a. doi:10.1145/3382025.3414944. (cited on Page 152)

Jacob Krüger, Sebastian Krieter, Gunter Saake, and Thomas Leich. EXtracting Product Lines from vAriaNTs (EXPLANT). In *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 13:1–2. ACM, 2020b. doi:10.1145/3377024.3377046. (cited on Page 1, 3, and 9)

Jacob Krüger, Christian Lausberger, Ivonne von Nostitz-Wallwitz, Gunter Saake, and Thomas Leich. Search. Review. Repeat? An Empirical Study of Threats to Replicating SLR Searches. *Empirical Software Engineering*, 25(1):627–677, 2020c. doi:10.1007/s10664-019-09763-0. (cited on Page 38 and 67)

Jacob Krüger, Wardah Mahmood, and Thorsten Berger. Promote-pl: A Round-Trip Engineering Process Model for Adopting and Evolving Product Lines. In *International Systems and Software Product Line Conference (SPLC)*, pages 2:1–12. ACM, 2020d. doi:10.1145/3382025.3414970. (cited on Page 5, 9, 39, 56, 147, and 149)

Jacob Krüger, Sebastian Nielebock, and Robert Heumüller. How Can I Contribute? A Qualitative Analysis of Community Websites of 25 Unix-Like Distributions. In *International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 324–329. ACM, 2020e. doi:10.1145/3383219.3383256. (cited on Page 65, 94, 104, 147, 148, 151, and 158)

Jacob Krüger, Gul Calikli, Thorsten Berger, and Thomas Leich. How Explicit Feature Traces Did Not Impact Developers' Memory. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021. (cited on Page 107, 114, and 121)

William H. Kruskal and W. Allen Wallis. Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association*, 47(260):583–621, 1952. doi:10.1080/01621459.1952.10483441. (cited on Page 127)

Juraj Kubelka, Alexandre Bergel, and Romain Robbes. Asking and Answering Questions during a Programming Change Task in Pharo Language. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, pages 1–11. ACM, 2014. doi:10.1145/2688204.2688212. (cited on Page 69 and 71)

Juraj Kubelka, Romain Robbes, and Alexandre Bergel. Live Programming and Software Evolution: Questions During a Programming Change Task. In *International Conference on Program Comprehension (ICPC)*, pages 30–41. IEEE, 2019. doi:10.1109/icpc.2019.00017. (cited on Page 69)

Elias Kuiter, Sebastian Krieter, Jacob Krüger, Kai Ludwig, Thomas Leich, and Gunter Saake. PCLocator: A Tool Suite to Automatically Identify Configurations for Code Locations. In *International Systems and Software Product Line Conference (SPLC)*, pages 284–288. ACM, 2018a. doi:10.1145/3233027.3236399. (cited on Page 116 and 120)

Elias Kuiter, Jacob Krüger, Sebastian Krieter, Thomas Leich, and Gunter Saake. Getting Rid of Clone-And-Own: Moving to a Software Product Line for Temperature Monitoring. In *International Systems and Software Product Line Conference (SPLC)*, pages 189–189. ACM, 2018b. doi:10.1145/3233027.3233050. (cited on Page 1, 2, 10, 12, 19, 33, 36, 43, 47, 49, 51, 53, 148, and 153)

Elias Kuiter, Sebastian Krieter, Jacob Krüger, Thomas Leich, and Gunter Saake. Foundations of Collaborative, Real-Time Feature Modeling. In *International Systems and Software Product Line Conference (SPLC)*, pages 257–264. ACM, 2019. doi:10.1145/3336294.3336308. (cited on Page 175)

Elias Kuiter, Sebastian Krieter, Jacob Krüger, Gunter Saake, and Thomas Leich. VariED: An Editor for Collaborative, Real-Time Feature Modeling. *Empirical Software Engineering*, 26(24):1–47, 2021. doi:10.1007/s10664-020-09892-x. (cited on Page 175)

Naveen Kulkarni and Vasudeva Varma. Perils of Opportunistically Reusing Software Module. *Software: Practice and Experience*, 47(7):971–984, 2016. doi:10.1002/spe.2439. (cited on Page 1, 9, and 38)

Miguel A. Laguna and Yania Crespo. A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring. *Science of Computer Programming*, 78(8):1010–1034, 2013. doi:10.1016/j.scico.2012.05.003. (cited on Page 2, 3, 4, 5, 32, and 149)

Jeremy Lanman, Rowland Darbin, Jorge Rivera, Paul C. Clements, and Charles W. Krueger. The Challenges of Applying Service Orientation to the U.S. Army's Live Training Software Product Line. In *International Software Product Line Conference (SPLC)*, pages 244–253. ACM, 2013. doi:10.1145/2491627.2491649. (cited on Page 36)

Thomas D. LaToza and Brad A. Myers. Hard-to-Answer Questions about Code. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, pages 8:1–6. ACM, 2010a. doi:10.1145/1937117.1937125. (cited on Page 69, 71, and 74)

Thomas D. LaToza and Brad A. Myers. Developers Ask Reachability Questions. In *International Conference on Software Engineering (ICSE)*, pages 185–194. ACM, 2010b. doi:10.1145/1806799.1806829. (cited on Page 66, 69, and 71)

Duc Le, Eric Walkingshaw, and Martin Erwig. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 143–150. IEEE, 2011. doi:10.1109/vlhcc.2011.6070391. (cited on Page 114 and 121)

Kwanwoo Lee, Kyo C. Kang, Eunman Koh, Wonsuk Chae, Bokyoung Kim, and Byoung W. Choi. Domain-Oriented Engineering of Elevator Control Software. In *Software Product Lines*, pages 3–22. Springer, 2000. doi:10.1007/978-1-4615-4339-8_1. (cited on Page 36 and 165)

Kwanwoo Lee, Kyo C. Kang, and Jaejoon Lee. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In *International Conference on Software Reuse (ICSR)*, pages 62–77. Springer, 2002. doi:10.1007/3-540-46020-9_5. (cited on Page 162, 165, 167, 168, 169, 172, and 174)

Dorothy Leonard-Barton. A Dual Methodology for Case Studies: Synergistic Use of a Longitudinal Single Site with Replicated Multiple Sites. *Organization Science*, 1(3): 248–266, 1990. doi:10.1287/orsc.1.3.248. (cited on Page 50, 63, 93, and 177)

Nicholas Lesiecki. Applyinq AspectJ to J2EE Application Development. *IEEE Software*, 23(1):24–32, 2006. doi:10.1109/ms.2006.1. (cited on Page 117)

Stanley Letovsky. Cognitive Processes in Program Comprehension. *Journal of Systems and Software*, 7(4):325–339, 1987. doi:10.1016/0164-1212(87)90032-x. (cited on Page 68)

Daniela Lettner, Florian Angerer, Herbert Prähofer, and Paul Grünbacher. A Case Study on Software Ecosystem Characteristics in Industrial Automation Software. In *International Conference on Software and Systems Process (ICSSP)*, pages 40–49. ACM, 2014. doi:10.1145/2600821.2600826. (cited on Page 11)

Daniela Lettner, Klaus Eder, Paul Grünbacher, and Herbert Prähofer. Feature Modeling of Two Large-Scale Industrial Software Systems: Experiences and Lessons Learned. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 386–395. IEEE, 2015. doi:10.1109/models.2015.7338270. (cited on Page 14, 165, 168, and 174)

Hareton Leung and Zhang Fan. Software Cost Estimation. In *Handbook of Software Engineering and Knowledge Engineering*, pages 307–324. World Scientific, 2002. doi:10.1142/9789812389701_0014. (cited on Page 19, 20, 21, and 23)

Dong Li and Carl K. Chang. Initiating and Institutionalizing Software Product Line Engineering: From Bottom-Up Approach to Top-Down Practice. In *Annual Computer Software and Applications Conference (COMPSAC)*, pages 53–60. IEEE, 2009. doi:10.1109/compsac.2009.17. (cited on Page 36, 43, 44, 46, and 47)

Dong Li and David M. Weiss. Adding Value through Software Product Line Engineering: The Evolution of the FISCAN Software Product Lines. In *International Software Product Line Conference (SPLC)*, pages 213–222. IEEE, 2011. doi:10.1109/splc.2011.43. (cited on Page 36, 44, 46, and 47)

Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006. doi:10.1109/tse.2006.28. (cited on Page 2)

Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *International Conference on Software Engineering (ICSE)*, pages 105–114. ACM, 2010. doi:10.1145/1806799.1806819. (cited on Page 1, 16, 52, 113, 116, and 138)

Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *International Conference on Aspect-Oriented Software Development (AOSD)*, pages 191–202. ACM, 2011. doi:10.1145/1960275.1960299. (cited on Page 1, 16, 60, 108, 113, 116, and 138)

Max Lillack, Ştefan Stănciulescu, Wilhelm Hedman, Thorsten Berger, and Andrzej Wąsowski. Intention-Based Integration of Software Variants. In *International Conference on Software Engineering (ICSE)*, pages 831–842. IEEE, 2019. doi:10.1109/icse.2019.00090. (cited on Page 2 and 9)

Wayne C. Lim. Effects of Reuse on Quality, Productivity, and Economics. *IEEE Software*, 11(5):23–30, 1994. doi:10.1109/52.311048. (cited on Page 36, 42, 46, and 47)

Wayne C. Lim. Reuse Economics: A Comparison of Seventeen Models and Directions for Future Research. In *International Conference on Software Reuse (ICSR)*, pages 41–50. IEEE, 1996. doi:10.1109/icsr.1996.496112. (cited on Page 21)

Crescencio Lima, Ivan do Carmo Machado, Eduardo S. de Almeida, and Christina von Flach G. Chavez. Recovering the Product Line Architecture of the Apo-Games. In *International Systems and Software Product Line Conference (SPLC)*, pages 289–293. ACM, 2018. doi:10.1145/3233027.3236398. (cited on Page 52)

Robert Lindohf, Jacob Krüger, Erik Herzog, and Thorsten Berger. Software Product-Line Evaluation in the Large. *Empirical Software Engineering*, 26(30):1–41, 2021. doi:10.1007/s10664-020-09913-9. (cited on Page 2, 4, 9, 10, 13, 19, 24, 33, 147, 148, 151, 160, 161, 177, 178, and 185)

Lukas Linsbauer. A Variability Aware Configuration Management and Revision Control Platform. In *International Conference on Software Engineering (ICSE)*, pages 803–806. ACM, 2016. doi:10.1145/2889160.2889262. (cited on Page 2)

Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. In *European Conference on Computer Systems (EuroSys)*, pages 191–204. ACM, 2006. doi:10.1145/1217935.1217954. (cited on Page 112 and 133)

John Long. Software Reuse Antipatterns. *ACM SIGSOFT Software Engineering Notes*, 26 (4):68–76, 2001. doi:10.1145/505482.505492. (cited on Page 1, 2, 9, 10, and 12)

Roberto E. Lopez-Herrejon and Don Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *International Symposium on Generative and Component-Based Software Engineering (GCSE)*, pages 10–24. Springer, 2001. doi:10.1007/3-540-44800-4_2. (cited on Page 14)

Roberto E. Lopez-Herrejon, Leticia Montalvillo-Mendizabal, and Alexander Egyed. From Requirements to Features: An Exploratory Study of Feature-Oriented Refactoring. In *International Software Product Line Conference (SPLC)*, pages 181–190. IEEE, 2011. doi:10.1109/splc.2011.52. (cited on Page 14)

Daniel Lucrédio, Kellyton dos Santos Brito, Alexandre Alvaro, Vinicius C. Garcia, Eduardo S. de Almeida, Renata P. de Mattos Fortes, and Silvio L. Meira. Software Reuse: The Brazilian Industry Scenario. *Journal of Systems and Software*, 81(6):996–1013, 2008. doi:10.1016/j.jss.2007.08.036. (cited on Page 36 and 43)

Kai Ludwig, Jacob Krüger, and Thomas Leich. Covert and Phantom Features in Annotations: Do They Impact Variability Analysis? In *International Systems and Software Product Line Conference (SPLC)*, pages 218–230. ACM, 2019. doi:10.1145/3336294.3336296. (cited on Page 107, 108, 113, and 115)

Kai Ludwig, Jacob Krüger, and Thomas Leich. FeatureCoPP: Unfolding Preprocessor Variability. In *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 24:1–9. ACM, 2020. doi:10.1145/3377024.3377039. (cited on Page 107, 109, 111, and 116)

Patrick Mäder and Alexander Egyed. Do Developers Benefit from Requirements Traceability When Evolving and Maintaining a Software System? *Empirical Software Engineering*, 20 (2):413–441, 2014. doi:10.1007/s10664-014-9314-z. (cited on Page 112)

Romero Malaquias, Márcio Ribeiro, Rodrigo Bonifácio, Eduardo Monteiro, Flávio Medeiros, Alessandro Garcia, and Rohit Gheyi. The Discipline of Preprocessor-Based Annotations - Does #ifdef TAG n't #endif Matter. In *International Conference on Program Comprehension (ICPC)*, pages 297–307. IEEE, 2017. doi:10.1109/icpc.2017.41. (cited on Page 114, 133, and 135)

Henry B. Mann and Donald R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947. doi:10.1214/aoms/1177730491. (cited on Page 140)

Jason Mansell. Experiences and Expectations Regarding the Introduction of Systematic Reuse in Small- and Medium-Sized Companies. In *Software Product Lines*, pages 91–124. Springer, 2006. doi:10.1007/978-3-540-33253-4_3. (cited on Page 26)

Christian Manz, Michael Stupperich, and Manfred Reichert. Towards Integrated Variant Management in Global Software Engineering: An Experience Report. In *International Conference on Global Software Engineering (ICGSE)*, pages 168–172. IEEE, 2013. doi:10.1109/icgse.2013.29. (cited on Page 165 and 173)

Luciano Marchezan, Elder Macedo Rodrigues, Maicon Bernardino, and Fábio Paulo Basso. PAxSPL: A Feature Retrieval Process for Software Product Line Reengineering. *Software: Practice and Experience*, 49(8):1278–1306, 2019. doi:10.1002/spe.2707. (cited on Page 153)

Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. (cited on Page 129)

Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Bottom-Up Adoption of Software Product Lines - A Generic and Extensible Approach. In *International Software Product Line Conference (SPLC)*, pages 101–110. ACM, 2015. doi:10.1145/2791060.2791086. (cited on Page 20)

Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *International Systems and Software Product Line Conference (SPLC)*, pages 38–41. ACM, 2017. doi:10.1145/3109729.3109748. (cited on Page 37)

Jabier Martinez, Xhevahire Tërnava, and Tewfik Ziadi. Software Product Line Extraction from Variability-Rich Systems: The Robocode Case Study. In *International Systems and Software Product Line Conference (SPLC)*, pages 132–142. ACM, 2018. doi:10.1145/3233027.3233038. (cited on Page 36, 153, and 160)

Yoshihiro Matsumoto. A Guide for Management and Financial Controls of Product Lines. In *International Software Product Line Conference (SPLC)*, pages 163–170. IEEE, 2007. doi:10.1109/spline.2007.26. (cited on Page 21 and 22)

David W. McDonald and Mark S. Ackerman. Expertise Recommender: A Flexible Recommendation System and Architecture. In *Conference on Computer Supported Cooperative Work (CSCW)*, pages 231–240. ACM, 2000. doi:10.1145/358916.358994. (cited on Page 81)

John D. McGregor, Linda M. Northrop, Salah Jarrad, and Klaus Pohl. Initiating Software Product Lines. *IEEE Software*, 19(4):24–27, 2002. doi:10.1109/ms.2002.1020282. (cited on Page 27)

Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. Investigating Preprocessor-Based Syntax Errors. In *International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pages 75–84. ACM, 2013. doi:10.1145/2517208.2517221. (cited on Page 113)

Flávio Medeiros, Christian Kästner, Márcio de Medeiros Ribeiro, Sarah Nadi, and Rohit Gheyi. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 495–518. Schloss Dagstuhl, 2015. doi:10.4230/LIPIcs.ECOOP.2015.495. (cited on Page 16, 60, 62, 113, 114, 117, 133, 135, and 138)

Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Transactions on Software Engineering*, 44(5):453–469, 2018. doi:10.1109/tse.2017.2688333. (cited on Page 114, 133, and 135)

Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017. doi:10.1007/978-3-319-61443-4. (cited on Page 2, 17, 18, and 52)

Jean Melo, Claus Brabrand, and Andrzej Wąsowski. How Does the Degree of Variability Affect Bug Finding? In *International Conference on Software Engineering (ICSE)*, pages 679–690. ACM, 2016. doi:10.1145/2884781.2884831. (cited on Page 114 and 115)

Bertrand Meyer. *Agile!* Springer, 2014. doi:10.1007/978-3-319-05155-0. (cited on Page 11 and 160)

Marc H. Meyer and Alvin P. Lehnerd. *The Power of Product Platforms: Building Value and Cost Leadership*. Simon and Schuster, 1997. (cited on Page 2 and 10)

Ali Mili, S. Fowler Chmiel, Ravi Gottumukkala, and Lisa Zhang. An Integrated Cost Model for Software Reuse. In *International Conference on Software Engineering (ICSE)*, pages 157–166. ACM, 2000. doi:10.1145/337180.337199. (cited on Page 1, 9, and 21)

Shawn Minto and Gail C. Murphy. Recommending Emergent Teams. In *International Workshop on Mining Software Repositories (MSR)*, pages 5:1–8. IEEE, 2007. doi:10.1109/msr.2007.27. (cited on Page 81)

Audris Mockus and James D. Herbsleb. Expertise Browser: A Quantitative Approach to Identifying Expertise. In *International Conference on Software Engineering (ICSE)*, pages 503–512. ACM, 2002. doi:10.1145/581339.581401. (cited on Page 81)

Parastoo Mohagheghi and Reidar Conradi. Quality, Productivity and Economic Benefits of Software Reuse: A Review of Industrial Studies. *Empirical Software Engineering*, 12(5): 471–516, 2007. doi:10.1007/s10664-007-9040-x. (cited on Page 37)

Kjetil Moløkken and Magne Jørgensen. A Review of Surveys on Software Effort Estimation. In *International Symposium on Empirical Software Engineering (ESE)*, pages 223–223. ACM, 2003. (cited on Page 33)

Leticia Montalvillo and Oscar Díaz. Tuning GitHub for SPL Development: Branching Models & Repository Operations for Product Engineers. In *International Software Product Line Conference (SPLC)*, pages 111–120. ACM, 2015. doi:10.1145/2791060.2791083. (cited on Page 159)

Alan Moran. *Managing Agile*. Springer, 2015. doi:10.1007/978-3-319-16262-1. (cited on Page 11 and 160)

Tim P. Moran. Anxiety and Working Memory Capacity: A Meta-Analysis and Narrative Review. *Psychological Bulletin*, 142(8):831–864, 2016. doi:10.1037/bul0000051. (cited on Page 69)

Mukelabai Mukelabai, Benjamin Behringer, Moritz Fey, Jochen Palz, Jacob Krüger, and Thorsten Berger. Multi-View Editing of Software Product Lines with PEoPL. In *International Conference on Software Engineering (ICSE)*, pages 81–84. ACM, 2018a. doi:10.1145/3183440.3183499. (cited on Page 107, 108, 109, and 111)

Mukelabai Mukelabai, Damir Nešić, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *International Conference on Automated Software Engineering (ASE)*, pages 155–166. ACM, 2018b. doi:10.1145/3238147.3238201. (cited on Page 27)

Raphael Muniz, Larissa Braz, Rohit Gheyi, Wilkerson Andrade, Baldoino Fonseca, and Márcio Ribeiro. A Qualitative Analysis of Variability Weaknesses in Configurable Systems with #ifdefs. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 51–58. ACM, 2018. doi:10.1145/3168365.3168382. (cited on Page 113, 114, and 115)

Jaap M. J. Murre and Joeri Dros. Replication and Analysis of Ebbinghaus' Forgetting Curve. *PLoS ONE*, 10(7):e0120644:1–23, 2015. doi:10.1371/journal.pone.0120644. (cited on Page 83 and 92)

Emtinan I. Mustafa and Rasha Osman. SEERA: A Software Cost Estimation Dataset for Constrained Environments. In *International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, pages 61–70. ACM, 2020. doi:10.1145/3416508.3417119. (cited on Page 4 and 23)

Motoi Nagamine, Tsuyoshi Nakajima, and Noriyoshi Kuno. A Case Study of Applying Software Product Line Engineering to the Air Conditioner Domain. In *International Systems and Software Product Line Conference (SPLC)*, pages 220–226. ACM, 2016. doi:10.1145/2934466.2934489. (cited on Page 36 and 153)

Sunil Nair, Jose L. de la Vara, and Sagar Sen. A Review of Traceability Research at the Requirements Engineering Conference$^{RE@21}$. In *International Conference Requirements Engineering (RE)*, pages 222–229. IEEE, 2013. doi:10.1109/re.2013.6636722. (cited on Page 5, 108, and 112)

Tsuneo Nakanishi, Kenji Hisazumi, and Akira Fukuda. Teaching Software Product Lines as a Paradigm to Engineers: An Experience Report in Education Programs and Seminars for Senior Engineers in Japan. In *International Systems and Software Product Line Conference (SPLC)*, pages 46–47. ACM, 2018. doi:10.1145/3236405.3237204. (cited on Page 165 and 170)

Krishna Narasimhan and Christoph Reichenbach. Copy and Paste Redeemed. In *International Conference on Automated Software Engineering (ASE)*, pages 630–640. IEEE, 2015. doi:10.1109/ase.2015.39. (cited on Page 2)

David A. Nembhard and Napassavong Osothsilp. An Empirical Comparison of Forgetting Models. *IEEE Transactions on Engineering Management*, 48(3):283–291, 2001. doi:10.1109/17.946527. (cited on Page 83)

Damir Nešić, Jacob Krüger, Ștefan Stănciulescu, and Thorsten Berger. Principles of Feature Modeling. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 62–73. ACM, 2019. doi:10.1145/3338906.3338974. (cited on Page 2, 9, 14, 33, 147, 148, 150, 151, 153, and 162)

Michael Nieke, Lukas Linsbauer, Jacob Krüger, and Thomas Leich. Second International Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution 2019). In *International Systems and Software Product Line Conference (SPLC)*, pages 320–320. ACM, 2019. doi:10.1145/3336294.3342367. (cited on Page 152)

Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. Commenting Source Code: Is It Worth It for Small Programming Tasks? *Empirical Software Engineering*, 24(3):1418–1457, 2019. doi:10.1007/s10664-018-9664-z. (cited on Page 65, 94, 99, 107, and 129)

Eila Niemelä, Mari Matinlassi, and Anne Taulavuori. Practical Evaluation of Software Product Family Architectures. In *International Software Product Line Conference (SPLC)*, pages 130–145. Springer, 2004. doi:10.1007/978-3-540-28630-1_8. (cited on Page 177)

Lars-Göran Nilsson. Memory Function in Normal Aging. *Acta Neurologica Scandinavica*, 107(179):7–13, 2003. doi:10.1034/j.1600-0404.107.s179.5.x. (cited on Page 92)

Jarley P. Nóbrega. An Integrated Cost Model for Product Line Engineering. Master's thesis, Federal University of Pernambuco, 2008. (cited on Page 22 and 23)

Jarley P. Nóbrega, Eduardo S. de Almeida, and Silvio R. de Lemos Meira. A Cost Framework Specification for Software Product Lines Scenarios. In *Workshop on Component-Based Development (WCBD)*, pages 1–8, 2006. (cited on Page 22 and 23)

Jarley P. Nóbrega, Eduardo S. de Almeida, and Sílvio R. L. Meira. InCoME: Integrated Cost Model for Product Line Engineering. In *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 27–34. IEEE, 2008. doi:10.1109/seaa.2008.41. (cited on Page 22 and 23)

Andy J. Nolan and Silvia Abrahão. Dealing with Cost Estimation in Software Product Lines: Experiences and Future Directions. In *International Software Product Line Conference (SPLC)*, pages 121–135. Springer, 2010. doi:10.1007/978-3-642-15579-6_9. (cited on Page 1, 4, 23, and 24)

Makoto Nonaka, Liming Zhu, Muhammad A. Babar, and Mark Staples. Impact of Architecture and Quality Investment in Software Product Line Development. In *International Software Product Line Conference (SPLC)*, pages 63–73. IEEE, 2007a. doi:10.1109/spline.2007.35. (cited on Page 21 and 22)

Makoto Nonaka, Liming Zhu, Muhammad A. Babar, and Mark Staples. Project Cost Overrun Simulation in Software Product Line Development. In *International Conference on Product Focused Software Process Improvement (PROFES)*, pages 330–344. Springer, 2007b. doi:10.1007/978-3-540-73460-4_29. (cited on Page 21 and 22)

Linda M. Northrop. SEI's Software Product Line Tenets. *IEEE Software*, 19(4):32–40, 2002. doi:10.1109/ms.2002.1020285. (cited on Page 5, 11, 20, 26, 37, 148, 150, 153, and 155)

Renato Novais, Creidiane Brito, and Manoel Mendonça. What Questions Developers Ask During Software Evolution? An Academic Perspective. In *Workshop on Software Visualization, Evolution, and Maintenance (VEM)*, pages 14–21, 2014. (cited on Page 69 and 71)

Object Management Group. *OMG® Unified Modeling Language® (OMG UML®)*, 2017. Version 2.5.1. (cited on Page 14 and 155)

Johnatan Oliveira, Markos Viggiato, and Eduardo Figueiredo. How Well Do You Know This Library? Mining Experts from Source Code Analysis. In *Brazilian Symposium on Software Quality (SBQS)*, pages 49–58. ACM, 2019. doi:10.1145/3364641.3364648. (cited on Page 81)

Jun Otsuka, Kouichi Kawarabata, Takashi Iwasaki, Makoto Uchiba, Tsuneo Nakanishi, and Kenji Hisazumi. Small Inexpensive Core Asset Construction for Large Gainful Product Line Development. In *International Software Product Line Conference (SPLC)*, pages 1–5. ACM, 2011. doi:10.1145/2019136.2019159. (cited on Page 36, 38, 46, and 47)

Ditmar Parmeza. Cost and Efforts in Product Lines for Developing Safety Critical Products – An Empirical Study. Master's thesis, Mälardalen University, 2015. (cited on Page 21 and 22)

David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972. doi:10.1145/361598.361623. (cited on Page 110 and 113)

Chris Parnin. A Cognitive Neuroscience Perspective on Memory for Programming Tasks. In *Annual Workshop of the Psychology of Programming Interest Group (WPPIG)*, pages 1–15. PPIG, 2010. (cited on Page 65)

Chris Parnin and Spencer Rugaber. Programmer Information Needs after Memory Failure. In *International Conference on Program Comprehension (ICPC)*, pages 123–132. IEEE, 2012. doi:10.1109/icpc.2012.6240479. (cited on Page 28, 65, and 69)

Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. Information Needs in Contemporary Code Review. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):135:1–27, 2018. doi:10.1145/3274404. (cited on Page 68)

Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. Feature-Oriented Software Evolution. In *International Workshop on Feature-Oriented Software Development (FOSD)*, pages 17:1–8. ACM, 2013. doi:10.1145/2430502.2430526. (cited on Page 131)

Dale R. Peterson. Economics of Software Product Lines. In *International Workshop on Software Product-Family Engineering (PFE)*, pages 381–402. Springer, 2004. doi:10.1007/978-3-540-24667-1_29. (cited on Page 22 and 23)

Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. Synchronizing Software Variants with VariantSync. In *International Systems and Software Product Line Conference (SPLC)*, pages 329–332. ACM, 2016. doi:10.1145/2934466.2962726. (cited on Page 2, 9, 10, 39, 159, and 177)

Gustavo Pinto, Weslley Torres, and Fernando Castor. A Study on the Most Popular Questions about Concurrent Programming. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, pages 39–46. ACM, 2015. doi:10.1145/2846680.2846687. (cited on Page 68)

Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering*. Springer, 2005. doi:10.1007/3-540-28901-1. (cited on Page 1, 2, 5, 11, 12, 13, 14, 22, 23, 26, 27, 36, 37, 44, 56, 148, 153, 154, 157, and 178)

Richard Pohl, Mischa Höchsmann, Philipp Wohlgemuth, and Christian Tischer. Variant Management Solution for Large Scale Software Product Lines. In *International Conference on Software Engineering (ICSE)*, pages 85–94. ACM, 2018. doi:10.1145/3183519.3183523. (cited on Page 153, 159, and 165)

Denys Poshyvanyk, Yann-Gaël Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Václav Rajlich. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Transactions on Software Engineering*, 33(6): 420–432, 2007. doi:10.1109/tse.2007.1016. (cited on Page 28)

Jeffrey S. Poulin. The Economics of Product Line Development. *International Journal of Applied Software Technology*, 3(1):20–34, 1997. (cited on Page 22, 23, and 24)

Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443. Springer, 1997. doi:10.1007/bfb0053389. (cited on Page 15, 17, 53, and 110)

Rodrigo Queiroz, Leonardo Passos, Marco T. Valente, Claus Hunsen, Sven Apel, and Krzysztof Czarnecki. The Shape of Feature Code: An Analysis of Twenty C-Preprocessor-Based Systems. *International Journal on Software & Systems Modeling*, 16(1):77–96, 2017. doi:10.1007/s10270-015-0483-z. (cited on Page 113 and 116)

Gerard Quilty and Mel Ó. Cinnéide. Experiences with Software Product Line Development in Risk Management Software. In *International Software Product Line Conference (SPLC)*, pages 251–260. IEEE, 2011. doi:10.1109/splc.2011.30. (cited on Page 36, 42, and 46)

R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, 2018–2020. URL https://www.R-project.org. (cited on Page 78, 86, 122, and 134)

Rick Rabiser, Klaus Schmid, Martin Becker, Goetz Botterweck, Matthias Galster, Iris Groher, and Danny Weyns. A Study and Comparison of Industrial vs. Academic Software Product Line Research Published at SPLC. In *International Systems and Software Product Line Conference (SPLC)*, pages 14–24. ACM, 2018. doi:10.1145/3233027.3233028. (cited on Page 3, 4, 5, 32, 149, 152, and 156)

Rick Rabiser, Klaus Schmid, Martin Becker, Goetz Botterweck, Matthias Galster, Iris Groher, and Danny Weyns. Industrial and Academic Software Product Line Research at SPLC: Perceptions of the Community. In *International Systems and Software Product Line Conference (SPLC)*, pages 189–194. ACM, 2019. doi:10.1145/3336294.3336310. (cited on Page 4 and 152)

D. Paul Ralph. Toward Methodological Guidelines for Process Theories and Taxonomies in Software Engineering. *IEEE Transactions on Software Engineering*, 45(7):712–735, 2019. doi:10.1109/tse.2018.2796554. (cited on Page 150, 151, and 161)

Awais Rashid, Thomas Cottenier, Phil Greenwood, Ruzanna Chitchyan, Regine Meunier, Roberta Coelho, Mario Südholt, and Wouter Joosen. Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe. *Computer*, 43(2):19–26, 2010. doi:10.1109/mc.2010.30. (cited on Page 115)

Abdul Razzaq, Asanka Wasala, Chris Exton, and Jim Buckley. The State of Empirical Evaluation in Static Feature Location. *ACM Transactions on Software Engineering and Methodology*, 28(1):2:1–58, 2018. doi:10.1145/3280988. (cited on Page 4, 29, 104, and 112)

Iris Reinhartz-Berger and Sameh Abbas. A Variability-Driven Analysis Method for Automatic Extraction of Domain Behaviors. In *International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 467–481. Springer, 2020. doi:10.1007/978-3-030-49435-3_29. (cited on Page 52)

Patrick Rempel and Patrick Mäder. Preventing Defects: The Impact of Requirements Traceability Completeness on Software Quality. *IEEE Transactions on Software Engineering*, 43(8):777–797, 2017. doi:10.1109/tse.2016.2622264. (cited on Page 112)

Meghan Revelle, Tiffany Broadbent, and David Coppit. Understanding Concerns in Software: Insights Gained from Two Case Studies. In *International Workshop on Program Comprehension (IWPC)*, pages 23–32. IEEE, 2005. doi:10.1109/wpc.2005.43. (cited on Page 30)

Luisa Rincón, Raúl Mazo, and Camille Salinesi. APPLIES: A Framework for EvaluAting Organization's Motivation and Preparation for AdoPting Product LInEs. In *International Conference on Research Challenges in Information Science (RCIS)*, pages 1–12. IEEE, 2018. doi:10.1109/rcis.2018.8406641. (cited on Page 4 and 177)

Luisa Rincón, Raúl Mazo, and Camille Salinesi. Analyzing the Convenience of Adopting a Product Line Engineering Approach: An Industrial Qualitative Evaluation. In *International Systems and Software Product Line Conference (SPLC)*, pages 90–97. ACM, 2019. doi:10.1145/3307630.3342418. (cited on Page 177)

David C. Rine and Robert M. Sonnemann. Investments in Reusable Software. A Study of Software Reuse Investment Success Factors. *Journal of Systems and Software*, 41(1): 17–32, 1998. doi:10.1016/s0164-1212(97)10003-6. (cited on Page 36 and 43)

Alcemir Rodrigues Santos, Ivan do Carmo Machado, Eduardo Santana de Almeida, Janet Siegmund, and Sven Apel. Comparing the Influence of Using Feature-Oriented Programming and Conditional Compilation on Comprehending Feature-Oriented Software. *Empirical Software Engineering*, 24(3):1226–1258, 2019. doi:10.1007/s10664-018-9658-x. (cited on Page 114, 121, and 122)

Marko Rosenmüller. *Towards Flexible Feature Composition: Static and Dynamic Binding in Software Product Lines*. PhD thesis, Otto-von-Guericke University Magdeburg, 2011. (cited on Page 15 and 160)

Marko Rosenmüller, Sven Apel, Thomas Leich, and Gunter Saake. Tailor-Made Data Management for Embedded Systems: A Case Study on Berkeley DB. *Data & Knowledge Engineering*, 68(12):1493–1512, 2009. doi:10.1016/j.datak.2009.07.013. (cited on Page 52)

Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming*, 74(7):470–495, 2009. doi:10.1016/j.scico.2009.02.007. (cited on Page 2 and 52)

Julia Rubin and Marsha Chechik. A Framework for Managing Cloned Product Variants. In *International Conference on Software Engineering (ICSE)*, pages 1233–1236. IEEE, 2013a. doi:10.1109/icse.2013.6606686. (cited on Page 9, 159, and 177)

Julia Rubin and Marsha Chechik. A Survey of Feature Location Techniques. In *Domain Engineering*, pages 29–58. Springer, 2013b. doi:10.1007/978-3-642-36654-3_2. (cited on Page 4, 20, 28, 29, and 112)

Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. Managing Forked Product Variants. In *International Software Product Line Conference (SPLC)*, pages 156–160. ACM, 2012. doi:10.1145/2362536.2362558. (cited on Page 2, 9, 10, 159, and 177)

Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Managing Cloned Variants: A Framework and Experience. In *International Software Product Line Conference (SPLC)*, pages 101–110. ACM, 2013. doi:10.1145/2491627.2491644. (cited on Page 2, 9, 159, and 177)

Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Cloned Product Variants: From Ad-Hoc to Managed Software Product Lines. *International Journal on Software Tools for Technology Transfer*, pages 627–646, 2015. doi:10.1007/s10009-014-0347-9. (cited on Page 1, 2, 9, 10, 39, and 153)

Per Runeson. Using Students as Experiment Subjects – An Analysis on Graduate and Freshmen Student Data. In *International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 95–102, 2003. (cited on Page 120)

Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell, editors. *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley, 2012. (cited on Page 50, 93, and 177)

Ioana Rus and Mikael Lindvall. Knowledge Management in Software Engineering. *IEEE Software*, 19(3):26–38, 2002. doi:10.1109/ms.2002.1003450. (cited on Page 66)

Mohammed Sayagh, Noureddine Kerzazi, Bram Adams, and Fabio Petrillo. Software Configuration Engineering in Practice: Interviews, Survey, and Systematic Literature Review. *IEEE Transactions on Software Engineering*, 46(6):646–673, 2020. doi:10.1109/tse.2018.2867847. (cited on Page 153)

Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software Diversity: State of the Art and Perspectives. *International Journal on Software Tools for Technology Transfer*, 14(5):477–495, 2012. doi:10.1007/s10009-012-0253-y. (cited on Page 2, 3, and 14)

Klaus Schmid. An Initial Model of Product Line Economics. In *International Workshop on Software Product-Family Engineering (PFE)*, pages 38–50. Springer, 2002. doi:10.1007/3-540-47833-7_5. (cited on Page 21, 22, and 23)

Klaus Schmid. Integrated Cost- and Investmentmodels for Product Family Development. Technical Report 067.03/E, Fraunhofer IESE, 2003. (cited on Page 22 and 23)

Klaus Schmid. A Quantitative Model of the Value of Architecture in Product Line Adoption. In *International Workshop on Software Product-Family Engineering (PFE)*, pages 32–43. Springer, 2004. doi:10.1007/978-3-540-24667-1_4. (cited on Page 22)

Klaus Schmid and Martin Verlage. The Economic Impact of Product Line Adoption and Evolution. *IEEE Software*, 19(4):50–57, 2002. doi:10.1109/ms.2002.1020287. (cited on Page 2, 10, 12, 20, 26, 149, 155, and 160)

Klaus Schmid, Isabel John, Ronny Kolb, and Gerald Meier. Introducing the PuLSE Approach to an Embedded System Population at Testo AG. In *International Conference on Software Engineering (ICSE)*, pages 544–552. ACM, 2005. doi:10.1145/1062455.1062552. (cited on Page 177)

Klaus Schmid, Rick Rabiser, and Paul Grünbacher. A Comparison of Decision Modeling Approaches in Product Lines. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 119–126. ACM, 2011. doi:10.1145/1944892.1944907. (cited on Page 14)

Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *International Conference Requirements Engineering (RE)*, pages 139–148. IEEE, 2006. doi:10.1109/re.2006.23. (cited on Page 14 and 162)

David Schuler and Thomas Zimmermann. Mining Usage Expertise from Version Archives. In *International Working Conference on Mining Software Repositories (MSR)*, pages 121–124. ACM, 2008. doi:10.1145/1370750.1370779. (cited on Page 82)

Alexander Schultheiß, Paul M. Bittner, Timo Kehrer, and Thomas Thüm. On the Use of Product-Line Variants as Experimental Subjects for Clone-and-Own Research: A Case Study. In *International Systems and Software Product Line Conference (SPLC)*, pages 27:1–6. ACM, 2020. doi:10.1145/3382025.3414972. (cited on Page 51 and 62)

Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. Does the Discipline of Preprocessor Annotations Matter? A Controlled Experiment. In *International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pages 65–74. ACM, 2013. doi:10.1145/2517208.2517215. (cited on Page 60, 62, and 114)

Christa Schwanninger, Iris Groher, Christoph Elsner, and Martin Lehofer. Variability Modelling throughout the Product Line Lifecycle. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 685–689. Springer, 2009. doi:10.1007/978-3-642-04425-0_55. (cited on Page 165 and 172)

Carolyn B. Seaman. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999. doi:10.1109/32.799955. (cited on Page 76 and 176)

Todd Sedano, D. Paul Ralph, and Cecile Peraire. The Product Backlog. In *International Conference on Software Engineering (ICSE)*, pages 200–211. IEEE, 2019. doi:10.1109/icse.2019.00036. (cited on Page 11)

Pranab K. Sen. Estimates of the Regression Coefficient Based on Kendall's Tau. *Journal of the American Statistical Association*, 63(324):1379–1389, 1968. doi:10.1080/01621459.1968.10480934. (cited on Page 79)

Kanwarpreet Sethi, Yuanfang Cai, Sunny Wong, Alessandro Garcia, and Claudio Sant'Anna. From Retrospect to Prospect: Assessing Modularity and Stability from Software Architecture. In *Joint Working Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA)*, pages 269–272. IEEE, 2009. doi:10.1109/wicsa.2009.5290817. (cited on Page 122)

Yusra Shakeel, Jacob Krüger, Ivonne von Nostitz-Wallwitz, Christian Lausberger, Gabriel C. Durand, Gunter Saake, and Thomas Leich. (Automated) Literature Analysis - Threats and Experiences. In *International Workshop on Software Engineering for Science (SE4Science)*, pages 20–27. ACM, 2018. doi:10.1145/3194747.3194748. (cited on Page 38 and 67)

Devesh Sharma, Aybüke Aurum, and Barbara Paech. Business Value through Product Line Engineering – A Case Study. In *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 167–174. IEEE, 2008. doi:10.1109/seaa.2008.19. (cited on Page 36, 42, 43, 46, and 47)

Vibhu S. Sharma, Rohit Mehra, and Vikrant Kaulgud. What Do Developers Want? An Advisor Approach for Developer Priorities. In *International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 78–81. IEEE, 2017. doi:10.1109/chase.2017.14. (cited on Page 69)

Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors. *Guide to Advanced Empirical Software Engineering*. Springer, 2008. doi:10.1007/978-1-84800-044-5. (cited on Page 3, 76, and 176)

Janet Siegmund. *Framework for Measuring Program Comprehension*. PhD thesis, Otto-von-Guericke University Magdeburg, 2012. (cited on Page 79 and 84)

Janet Siegmund and Jana Schumann. Confounding Parameters on Program Comprehension: A Literature Survey. *Empirical Software Engineering*, 20(4):1159–1192, 2015. doi:10.1007/s10664-014-9318-8. (cited on Page 69, 122, 134, and 136)

Janet Siegmund, Christian Kästner, Jörg Liebig, and Sven Apel. Comparing Program Comprehension of Physically and Virtually Separated Concerns. In *International Workshop on Feature-Oriented Software Development (FOSD)*, pages 17–24. ACM, 2012. doi:10.1145/2377816.2377819. (cited on Page 62, 114, 121, and 122)

Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring and Modeling Programming Experience. *Empirical Software Engineering*, 19(5): 1299–1334, 2014. doi:10.1007/s10664-013-9286-4. (cited on Page 79, 84, 123, and 136)

Janet Siegmund, Norbert Siegmund, and Sven Apel. Views on Internal and External Validity in Empirical Software Engineering. In *International Conference on Software Engineering (ICSE)*, pages 9–19. IEEE, 2015. doi:10.1109/icse.2015.24. (cited on Page 131, 137, 144, and 181)

Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions Programmers Ask During Software Evolution Tasks. In *International Symposium on the Foundations of Software Engineering (FSE)*, pages 23–34. ACM, 2006. doi:10.1145/1181775.1181779. (cited on Page 69, 71, and 74)

Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering*, 34(4): 434–451, 2008. doi:10.1109/tse.2008.26. (cited on Page 69)

Marco Sinnema, Sybren Deelstra, Jos Nijhuis, and Jan Bosch. COVAMOF: A Framework for Modeling Variability in Software Product Families. In *International Software Product Line Conference (SPLC)*, pages 197–213. Springer, 2004. doi:10.1007/978-3-540-28630-1_12. (cited on Page 165)

Dag I. K. Sjøberg, Tore Dybå, Bente C. D. Anda, and Jo E. Hannay. Building Theories in Software Engineering. In *Guide to Advanced Empirical Software Engineering*, pages 312–336. Springer, 2008. doi:10.1007/978-1-84800-044-5_12. (cited on Page 150)

Dag I. K. Sjøberg, Aiko Yamashita, Bente C. D. Anda, Audris Mockus, and Tore Dybå. Quantifying the Effect of Code Smells on Maintenance Effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2013. doi:10.1109/tse.2012.89. (cited on Page 142)

Odd P. N. Slyngstad, Anita Gupta, Reidar Conradi, Parastoo Mohagheghi, Harald Rønneberg, and Einar Landre. An Empirical Study of Developers Views on Software Reuse in Statoil ASA. In *International Symposium on Empirical Software Engineering (ISESE)*, pages 242–251. ACM, 2006. doi:10.1145/1159733.1159770. (cited on Page 36, 42, 45, 46, and 47)

Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather R. Lipford. Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 248–259. ACM, 2015. doi:10.1145/2786805.2786812. (cited on Page 69, 71, and 74)

Hannah Snyder. Literature Review as a Research Methodology: An Overview and Guidelines. *Journal of Business Research*, 104:333–339, 2019. doi:10.1016/j.jbusres.2019.07.039. (cited on Page 150 and 151)

Software Engineering Institute. SEI Product Line Bibliography. Technical Report REV-03.18.2016.0, Carnegie Mellon University, 2018. (cited on Page 37)

Henry Spencer and Geoff Collyer. #ifdef Considered Harmful, or Portability Experience With C News. In *USENIX Conference (USENIX)*, pages 185–197. USENIX, 1992. (cited on Page 112, 113, 117, and 133)

Webb Stacy and Jean MacMillan. Cognitive Bias in Software Engineering. *Communications of the ACM*, 38(6):57–63, 1995. doi:10.1145/203241.203256. (cited on Page 69)

Ștefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. Forked and Integrated Variants in an Open-Source Firmware Project. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 151–160. IEEE, 2015. doi:10.1109/icsm.2015.7332461. (cited on Page 1, 2, 9, 39, 92, 94, 147, 148, and 159)

Thomas A. Standish. An Essay on Software Reuse. *IEEE Transactions on Software Engineering*, SE-10(5):494–497, 1984. doi:10.1109/tse.1984.5010272. (cited on Page 1, 9, 66, and 113)

Mark Staples and Derrick Hill. Experiences Adopting Software Product Line Development without a Product Line Architecture. In *Asia-Pacific Software Engineering Conference (APSEC)*, pages 176–183. IEEE, 2004. doi:10.1109/apsec.2004.50. (cited on Page 36, 42, 43, 46, 159, and 160)

Miroslaw Staron. *Action Research in Software Engineering*. Springer, 2020. doi:10.1007/978-3-030-32610-4. (cited on Page 50 and 177)

Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *International Systems and Software Product Line Conference (SPLC)*, pages 177–188. ACM, 2019. doi:10.1145/3336294.3336302. (cited on Page 9, 19, 20, 27, 28, 50, 62, 147, 148, 152, 153, 158, and 159)

Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A Taxonomy of Variability Realization Techniques. *Software: Practice and Experience*, 35(8):705–754, 2005. doi:10.1002/spe.652. (cited on Page 2, 15, and 109)

Antony Tang, Wim Couwenberg, Erik Scheppink, Niels A. de Burgh, Sybren Deelstra, and Hans van Vliet. SPL Migration Tensions: An Industry Experience. In *Workshop on Knowledge-Oriented Product Line Engineering (KOPLE)*, pages 3:1–6. ACM, 2010. doi:10.1145/1964138.1964141. (cited on Page 4 and 24)

Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How Do Software Engineers Understand Code Changes? - An Exploratory Study in Industry. In *International Symposium on the Foundations of Software Engineering (FSE)*, pages 51:1–11. ACM, 2012. doi:10.1145/2393596.2393656. (cited on Page 69 and 75)

Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *International Conference on Software Engineering (ICSE)*, pages 107–119. ACM, 1999. doi:10.1145/302405.302457. (cited on Page 110 and 113)

Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *European Conference on Computer Systems (EuroSys)*, pages 47–60. ACM, 2011. doi:10.1145/1966445.1966451. (cited on Page 112 and 133)

Aleksandra Tešanović, Ke Sheng, and Jörgen Hansson. Application-Tailored Database Systems: A Case of Aspects in an Embedded Database. In *International Database Engineering and Applications Symposium (IDEAS)*, pages 291–301. IEEE, 2004. doi:10.1109/ideas.2004.1319803. (cited on Page 52)

Johannes Thönes. Microservices. *IEEE Software*, 32(1):116–116, 2015. doi:10.1109/ms.2015.11. (cited on Page 160)

Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):1–45, 2014a. doi:10.1145/2580950. (cited on Page 14)

Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 79:70–85, 2014b. doi:10.1016/j.scico.2012.06.002. (cited on Page 2 and 17)

Thomas Thüm, Sebastian Krieter, and Ina Schaefer. Product Configuration in the Wild: Strategies for Conflicting Decisions in Web Configurators. In *International Configuration Workshop (ConfWS)*, pages 1–8. CEUR-WS.org, 2018. (cited on Page 18, 166, and 174)

Anil K. Thurimella and T. Maruthi Padmaja. Economic Models and Value-Based Approaches for Product Line Architectures. In *Economics-Driven Software Architecture*, pages 11–36. Elsevier, 2014. doi:10.1016/b978-0-12-410464-8.00002-7. (cited on Page 4, 21, 22, and 23)

Rebecca Tiarks. What Maintenance Programmers Really Do: An Observational Study. In *Workshop on Software Reengineering (WSR)*, pages 36–37. University of Siegen, 2011. (cited on Page 4, 28, and 66)

Amir Tomer, Leah Goldin, Tsvi Kuflik, Esther Kimchi, and Stephen R. Schach. Evaluating Software Reuse Alternatives: A Model and Its Application to an Industrial Case Study. *IEEE Transactions on Software Engineering*, 30(9):601–612, 2004. doi:10.1109/tse.2004.50. (cited on Page 1, 9, and 22)

Richard Torkar, Tony Gorschek, Robert Feldt, Mikael Svahnberg, Uzair A. Raja, and Kashif Kamran. Requirements Traceability: A Systematic Review and Industry Case Study. *International Journal of Software Engineering and Knowledge Engineering*, 22 (03):385–433, 2012. doi:10.1142/s021819401250009x. (cited on Page 108 and 112)

Adam Trendowicz. *Software Cost Estimation, Benchmarking, and Risk Assessment*. Springer, 2013. doi:10.1007/978-3-642-30764-5. (cited on Page 20 and 33)

Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and Why Your Code Starts to Smell Bad. In *International Conference on Software Engineering (ICSE)*, pages 403–414. IEEE, 2015. doi:10.1109/icse.2015.59. (cited on Page 142)

Eray Tüzün and Bedir Tekinerdogan. Analyzing Impact of Experience Curve on ROI in the Software Product Line Adoption Process. *Information and Software Technology*, 59: 136–148, 2015. doi:10.1016/j.infsof.2014.09.008. (cited on Page 21, 22, and 23)

Eray Tüzün, Bedir Tekinerdogan, Mert E. Kalender, and Semih Bilgen. Empirical Evaluation of a Decision Support Model for Adopting Software Product Line Engineering. *Information and Software Technology*, 60:77–101, 2015. doi:10.1016/j.infsof.2014.12.007. (cited on Page 177)

Muhammad Usman, Muhammad Z. Iqbal, and Muhammad U. Khan. A Product-Line Model-Driven Engineering Approach for Generating Feature-Based Mobile Applications. *Journal of Systems and Software*, 123:1–32, 2017. doi:10.1016/j.jss.2016.09.049. (cited on Page 153)

Tassio Vale, Eduardo S. de Almeida, Vander R. Alves, Uirá Kulesza, Nan Niu, and Ricardo de Lima. Software Product Lines Traceability: A Systematic Mapping Study. *Information and Software Technology*, 84:1–18, 2017. doi:10.1016/j.infsof.2016.12.004. (cited on Page 5, 107, 108, 109, 110, and 112)

Frank J. van der Linden. Software Product Families in Europe: The Esaps & Café Projects. *IEEE Software*, 19(4):41–49, 2002. doi:10.1109/ms.2002.1020286. (cited on Page 1, 12, and 26)

Frank J. van der Linden. Family Evaluation Framework: Overview & Introduction. Technical Report PH-0503-01, Philips, 2005. (cited on Page 2, 4, 10, 12, 26, and 177)

Frank J. van der Linden. Philips Healthcare Compositional Diversity Case. In *Systems and Software Variability Management*, pages 185–202. Springer, 2013. doi:10.1007/978-3-642-36583-6_13. (cited on Page 36, 42, 46, and 47)

Frank J. van der Linden, Jan Bosch, Erik Kamsties, Kari Känsälä, and Henk Obbink. Software Product Family Evaluation. In *International Workshop on Software Product-Family Engineering (PFE)*, pages 110–129. Springer, 2004. doi:10.1007/978-3-540-28630-1_7. (cited on Page 1, 2, 10, 12, 26, 177, and 178)

Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action*. Springer, 2007. doi:10.1007/978-3-540-71437-8. (cited on Page 1, 2, 4, 5, 9, 10, 11, 12, 14, 27, 36, 37, 43, 47, 148, 177, 179, and 184)

Julia Varnell-Sarjeant, Anneliese Amschler Andrews, Joe Lucente, and Andreas Stefik. Comparing Development Approaches and Reuse Strategies: An Empirical Evaluation of Developer Views from the Aerospace Industry. *Information and Software Technology*, 61: 71–92, 2015. doi:10.1016/j.infsof.2015.01.002. (cited on Page 39)

Karina Villela, Adeline Silva, Tassio Vale, and Eduardo Santana de Almeida. A Survey on Software Variability Management Approaches. In *International Software Product Line Conference (SPLC)*, pages 147–156. ACM, 2014. doi:10.1145/2648511.2648527. (cited on Page 9)

Jacob Viner. Cost Curves and Supply Curves. *Zeitschrift für Nationalökonomie*, 3(1):23–46, 1932. doi:10.1007/978-3-662-39842-5_1. (cited on Page 27)

Anneliese von Mayrhauser and A. Marie Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995. doi:10.1109/2.402076. (cited on Page 66)

Robert Walker and Rylan Cottrell. Pragmatic Software Reuse: A View from the Trenches. Technical Report 2016-1088-07, University of Calgary, 2016. (cited on Page 36, 41, 46, and 47)

Eric Walkingshaw and Klaus Ostermann. Projectional Editing of Variational Software. In *International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pages 29–38. ACM, 2014. doi:10.1145/2658761.2658766. (cited on Page 111)

Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. An Exploratory Study of Feature Location Process: Distinct Phases, Recurring Patterns, and Elementary Actions. In *International Conference on Software Maintenance (ICSM)*, pages 213–222. IEEE, 2011. doi:10.1109/icsm.2011.6080788. (cited on Page 30 and 100)

Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study. *Journal of Software: Evolution and Process*, 25(11):1193–1224, 2013. doi:10.1002/smr.1593. (cited on Page 4, 28, 30, 31, 100, 112, and 128)

Ronald L. Wasserstein and Nicole A. Lazar. The ASA Statement on p-Values: Context, Process, and Purpose. *The American Statistician*, 70(2):129–133, 2016. doi:10.1080/00031305.2016.1154108. (cited on Page 86, 122, 133, and 144)

Ronald L. Wasserstein, Allen L. Schirm, and Nicole A. Lazar. Moving to a World Beyond "p $<$ 0.05". *The American Statistician*, 73(sup1):1–19, 2019. doi:10.1080/00031305.2019.1583913. (cited on Page 86, 122, 133, and 144)

Jens H. Weber, Anita Katahoire, and Morgan Price. Uncovering Variability Models for Software Ecosystems from Multi-Repository Structures. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 103–108. ACM, 2015. doi:10.1145/2701319.2701333. (cited on Page 153)

Jacco H. Wesselius. Strategic Scenario-Based Valuation of Product Line Roadmaps. In *Software Product Lines*, pages 53–89. Springer, 2006. doi:10.1007/978-3-540-33253-4_2. (cited on Page 22)

Frank Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6): 80–83, 1945. doi:10.2307/3001968. (cited on Page 79)

Norman Wilde, Michelle Buckellew, Henry Page, and Vaclav Rajlich. A Case Study of Feature Location in Unstructured Legacy Fortran Code. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 68–76. IEEE, 2001. doi:10.1109/csmr.2001.914970. (cited on Page 28)

Norman Wilde, Michelle Buckellew, Henry Page, Vaclav Rajlich, and LaTreva Pounds. A Comparison of Methods for Locating Features in Legacy Software. *Journal of Systems and Software*, 65(2):105–114, 2003. doi:10.1016/s0164-1212(02)00052-3. (cited on Page 4, 28, 30, 31, and 104)

James Withey. Investment Analysis of Software Assets for Product Lines. Technical Report CMU/SEI-96-TR-010, Carnegie Mellon University, 1996. (cited on Page 22 and 23)

Claes Wohlin. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 38:1–10. ACM, 2014. doi:10.1145/2601248.2601268. (cited on Page 21, 67, 81, and 163)

Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer, 2012. doi:10.1007/978-3-642-29044-2. (cited on Page 68, 69, 72, and 81)

Wayne Wolf. Cyber-Physical Systems. *Computer*, 42(3):88–89, 2009. doi:10.1109/mc.2009.81. (cited on Page 160)

Yinxing Xue. Reengineering Legacy Software Products into Software Product Line Based on Automatic Variability Analysis. In *International Conference on Software Engineering (ICSE)*, pages 1114–1117. ACM, 2011. doi:10.1145/1985793.1986009. (cited on Page 56)

Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. Feature Location in a Collection of Product Variants. In *Working Conference on Reverse Engineering (WCRE)*, pages 145–154. IEEE, 2012. doi:10.1109/wcre.2012.24. (cited on Page 28 and 56)

Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig. Defining a Strategy to Introduce a Software Product Line Using Existing Embedded Systems. In *International Software Product Line Conference (SPLC)*, pages 63–72. ACM, 2006a. doi:10.1145/1176887.1176897. (cited on Page 23)

Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig. Assessing Merge Potential of Existing Engine Control Systems into a Product Line. In *International Workshop on Software Engineering for Automotive Systems (SEAS)*, pages 61–67. ACM, 2006b. doi:10.1145/1138474.1138485. (cited on Page 1, 2, 10, and 12)

Bobbi Young, Judd Cheatwood, Todd Peterson, Rick Flores, and Paul C. Clements. Product Line Engineering Meets Model Based Engineering in the Defense and Automotive Industries. In *International Systems and Software Product Line Conference (SPLC)*, pages 175–179. ACM, 2017. doi:10.1145/3106195.3106220. (cited on Page 153)

Trevor J. Young. Using AspectJ to Build a Software Product Line for Mobile Devices. Master's thesis, University of British Columbia, 2005. (cited on Page 122)

Tao Yue, Shaukat Ali, and Bran Selic. Cyber-Physical System Product Line Engineering: Comprehensive Domain Analysis and Experience Report. In *International Software Product Line Conference (SPLC)*, pages 338–347. ACM, 2015. doi:10.1145/2791060.2791067. (cited on Page 153 and 160)

Bo Zhang, Slawomir Duszynski, and Martin Becker. Variability Mechanisms and Lessons Learned in Practice. In *International Workshop on Conducting Empirical Studies in Industry (CESI)*, pages 14–20. ACM, 2016. doi:10.1145/2897045.2897048. (cited on Page 15)

Gang Zhang, Liwei Shen, Xin Peng, Zhenchang Xing, and Wenyun Zhao. Incremental and Iterative Reengineering towards Software Product Line: An Industrial Case Study. In *International Conference on Software Maintenance (ICSM)*, pages 418–427. IEEE, 2011. doi:10.1109/icsm.2011.6080809. (cited on Page 36)

Shurui Zhou, Ștefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. Identifying Features in Forks. In *International Conference on Software Engineering (ICSE)*, pages 106–116. ACM, 2018. doi:10.1145/3180155.3180205. (cited on Page 94)

Thomas Zimmermann. Card-Sorting: From Text to Themes. In *Perspectives on Data Science for Software Engineering*, pages 137–141. Elsevier, 2016. doi:10.1016/b978-0-12-804206-9.00027-1. (cited on Page 29, 38, 68, 81, 120, 128, and 152)

Ehrenerklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; verwendete fremde und eigene Quellen sind als solche kenntlich gemacht. Insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Ich habe insbesondere nicht wissentlich:

- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht Unterlassungs- und Schadensersatzansprüche des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen kann. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht.

Magdeburg, den 10.09.2021

Jacob Krüger