Variables Nanodatenmanagement für eingebettete Systeme

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik

der Otto-von-Guericke-Universität Magdeburg

von: Diplom-Wirtschaftsinformatiker Thomas Leich

geboren am 23.09.1976 in Zerbst

Gutachter:

Prof. Dr. Gunter Saake

Prof. Dr. Kai-Uwe Sattler

Prof. Dr. Klaus Turowski

Ort und Datum des Promotionskolloquiums: Magdeburg, den 30.11.2012

Leich, Thomas:
Variables Nanodatenmanagement für eingebettete Systeme
Dissertation, Otto-von-Guericke-Universität
Magdeburg, 2012.

Zusammenfassung

Weltweit sind ca. 98 Prozent aller hergestellten Rechnersysteme als eingebettete Systeme im Einsatz. In diesem Kontext sind die Kosten für die Hardware ein sehr wichtiger Faktor. Deshalb kommt es bei der Softwareentwicklung darauf an, den Ressourcenbedarf zu minimieren, um so preisgünstige Hardware einsetzen zu können. Beispielanwendungen aus dem Bereich der Sensornetzwerke zeigen, dass derartig eingebettete Rechnersysteme stark variierende Elemente klassischer Infrastruktursoftware zur Datenhaltung benötigen. Eine Adaption vorhandener Mehrzwecksystemlösungen für diesen Bereich des Datenmanagements aus dem Großrechner- oder dem PC-Bereich ist auf Grund der Heterogenität der Hard- und Software sowie der extremen Ressourcenbeschränkungen nicht möglich. Das Resultat in der Praxis ist die wiederkehrende Entwicklung ähnlicher Datenhaltungskomponenten als Teil der Anwendungssoftware.

Eine im Rahmen dieser Dissertation angefertigte Analyse von vorhandenen Forschungsprototypen und kommerziellen DBMS beziehungsweise Datenhaltungskomponenten für eingebettete Systeme zeigt, dass der Großteil der komplexen Basis dieser DBMS auf Forschungsergebnissen der späten achtziger beziehungsweise frühen neunziger Jahre basiert. Eine Anpassung auf die heutigen Entwicklungen in der Softwaretechnik oder der Programmiersprachenentwicklung hat nur in Ansätzen stattgefunden. Die Adaption der verwendeten Methoden und Techniken ist aber auf Grund der zunehmenden Heterogenität des Anwendungsspektrums in diesem Bereich unumgänglich.

Neue Ansätze aus dem Software-Engineering, insbesondere aus den Bereichen der Komponententechniken, Produktlinien, Programmfamilien und der Merkmalsorientierten Softwareentwicklung versprechen Hilfe, um die beschriebenen Schwierigkeiten zu überwinden, ohne die Vorteile einer Spezialzwecksoftware wie zum Beispiel Laufzeiteffizienz und geringer Speicherverbrauch zu verlieren. Allerdings wurden diese Methoden bislang nur in einem sehr begrenzten Maße an Infrastruktursoftware für eingebettete

Systeme erprobt beziehungsweise sind sie derzeitig nur sehr beschränkt einsetzbar.

Ziel dieser Dissertation ist es, den Bereich der DBMS beziehungsweise der Datenhaltungskomponenten für eingebettete Systeme zu untersuchen und auf der Basis einer hochkonfigurierbaren DBMS-Produktlinien zu ergründen, inwieweit der Entwicklungsprozess und die eingesetzten Methoden und Techniken verbessert werden können. Dabei liegt aus datenbanktechnischer Sicht der Hauptforschungsaspekt auf dem Überdenken vorhandener Implementierungsstrukturen beziehungsweise der Anpassung von vorhandenen, modularen Strukturen von konfigurierbaren DBMS auf moderne Programmfamilientechnologien.

Aus software- beziehungsweise programmiertechnischer Sicht werden Methoden, Werkzeuge und Sprachen auf die speziellen Belange des Bereiches der eingebetteten Systeme adaptiert. Im Rahmen der Arbeit erzielte Ergebnisse aus der Entwicklung einer DBMS-Produktlinie zeigen, dass speziell das Konzept der Merkmalsorientierten Softwareentwicklung erfolgversprechend ist. Um die Merkmalsorientierte Softwareentwicklung für den Bereich der eingebetteten Systeme zu nutzen, ist eine Spracherweiterung für C++ entwickelt worden, die die Merkmalsorientierte Softwareentwicklung auch für den Bereich der eingebetteten Systeme ermöglicht. Neben der Spracherweiterung wurden ebenfalls der Entwicklungsprozess einer solchen DBMS-Familie und die nötige Werkzeugunterstützung weiterentwickelt. Eine vergleichende Untersuchung zu einem aspektorientierten Ansatz aus dem Bereich der eingebetteten Echtzeit-DBMS zeigt, dass sowohl hinsichtlich der Modularisierung, Variabilität als auch hinsichtlich der Granularität der Module und des Entwicklungskomforts Vorteile erzielt werden konnten.

Stichworte: Erweiterbares Datenmanagement, Datenmanagement für eingebettete Systeme, Nanodatenmanagement, eingebettetes Datenmanagement, FEATUREC++, FEATUREIDE

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich beim Erstellen dieser Dissertation unterstützt haben. Mein besonderer Dank gilt meinem Doktorvater Herrn Prof. Dr. Gunter Saake, der mich durch alle „Höhen und Tiefen“ beim Anfertigen dieser Dissertation begleitete und immer mit Rat und Tat zur Seite stand. Insbesondere in der schwierigen Anfangszeit, als zahlreiche Publikationsversuche scheiterten und die Thematik bei vielen Gutachtern als zu ambitioniert oder wenig zielführend abgestempelt wurde, hat er mir immer wieder Mut zugesprochen, an die Thematik geglaubt und dieses Forschungsfeld unterstützt. In der Schlussphase konnte er mich dann hinsichtlich der lang hinausgezögerten Fertigstellung immer wieder anspornen, die Arbeitsintensität und Fokussierung zu verstärken.
In diesem Zusammenhang gilt auch Herrn Dr. Sven Apel ein besonderer Dank. Mit ihm habe ich einen langen Weg in meinem „Forscherleben“ gemeinsam beschritten. Der „Nicht-Datenbänkler“ Dr. Apel hat mir, „dem Datenbänkler“ insbesondere das Thema Software-Engineering näher gebracht und sich viele Stunden mit mir über die Domäne Datenbanken wissenschaftlich auseinandergesetzt. Unsere gemeinsamen Forschungsthemen waren im Umfeld der Arbeitsgruppe zunächst etwas exotisch, haben aber im Laufe der Zeit immer mehr Mitstreiter gefunden.
Ein großes Dankeschön auch an die Arbeitsgruppe Datenbanken und an die Kollegen, die thematisch viele Diskussionen ermöglichten und Anregungen gaben. In diesem Zusammenhang möchte ich mich ganz besonders bei Prof. Dr. Christian Kästner, Dr. Marko Rosenmüller, Dr. Martin Kuhlemann und Thomas Thüm bedanken.
Dank gilt auch dem COMET DBMS- und dem *Real-Time Systems Laboratory*-Team der Universität Linköping, die mich für mehrere Monate in ihr Team als Gastwissenschaftler aufgenommen haben. Hier möchte ich besonders Frau Dr. Aleksandra Tešanovic Dank aussprechen, die mir wirklich engagiert die Probleme im Bereich der tief eingebetteten Systeme näherbrachte.

Ein weiterer Dank gilt meinen ehemaligen Diplomanden im Generellen und im Speziellen denen, die mit mir gemeinsam auf dem Gebiet meiner Dissertation mitgearbeitet haben und sich für die Thematik begeistern ließen. Dies sind im Einzelnen Rene Bärecke (FAMEDBMS 1.0 Storagemanager), Dr. Marko Rosenmüller (FEATUREC++), Laura Marnitz (FEATUREIDE 1.0), Dr. Mario Pukall (FAMEDBMS 2.0 Transaktionsverwaltung), Marco Gittler (Refaktorisierung Berkeley DB), Jörg Liebig (ROBBYDBMS), Janet Feigenspan (FEATUREIDE 2.0 und Programmverständnis) und Sara Kunze (ROBBYDBMS).
Des Weiteren möchte ich mich auch sehr bei meinen Kollegen der METOP, ganz besonders meines Bereiches „Angewandte Informatik“ bedanken.
Die Werkzeugunterstützung für die Merkmalsorientierte Programmierung durch FEATUREIDE war in der ersten Version ein typischer Forschungsprototyp mit vielen Schwächen im praktischen Einsatz. Doch unsere intensiven Bemühungen, gemeinsam mit Prof. Dr. Christian Kästner, Thomas Thüm, Dr. Sven Apel und Prof. Dr. Don Batory und einer Vielzahl an Studenten, haben unseren Vorschlag von einer Tool-Unterstützung für die Merkmalsorientierte Programmierung zum weltweiten Standardwerkzeug entwickelt. Für die Implementierung von Teilfunktionalitäten möchte ich mich hierfür bei Constanze Adler, Christian Becker, Stephan Besecke, David Broneske, Tom Brosch, Alexander Dreiling, Janet Feigenspan, Christoph Giesel, David Halm, Sebastian Henneberg, Marcus Kamieth, Stephan Kauschka, Dariusz Krolikowski, Maik Lampe, Laura Marnitz, Cyrill Meier, Marcus Leich, Melanie Pflaume, Eric Schubert, Hannes Smuracsky, Torsten Stöter, Patrick Sulkowski, Patrick Venohr, Jan Wedding und Fabian Wielgorz bedanken.

Für die zahlreichen Korrekturanregungen bei der Erstellung meiner Arbeit gilt mein Dank Prof. Dr. Gunter Saake, Prof. Dr. Kai-Uwe Sattler, Prof. Dr. Klaus Turowski, Dr. Sven Apel, Gisela Ahrens, Prof. Dr. Christian Kästner, Andy Kenner, Marcus Leich, Sandra Leich, Matthias Ritter, Martin Schäler, Ivonne Schröter und Thomas Thüm.
Zum Schluss möchte ich mich noch bei meiner Familie für die großartige Unterstützung bedanken. Meine Frau Sandra, meine Kinder Maja und Annina mussten leider viele Wochenenden auf den „Papa“ verzichten. Dennoch standen sie immer hinter mir und haben mir viel Kraft gegeben. Gleiches gilt auch für meine Eltern, die mir zudem das Studium ermöglichten und mir immer mit Rat und Tat zur Seite standen. Ein weiterer großer Dank gilt meinen Brüdern Andreas und Marcus sowie meinen Schwiegereltern, die mir immer den Rücken frei gehalten haben.

Liste der Veröffentlichungen

Apel, S.; Kästner, C.; Kuhlemann, M.; Leich, T.: Objektorientierte Grenzen - Modularität von Softwarebausteinen: Aspekte versus Merkmale. *iX Magazin für Professionelle Informationstechnik*, Nr. 10, S. 116–122, 2006.

Apel, S.; Kästner, C.; Kuhlemann, M.; Leich, T.: Pointcuts, Advice, Refinements, and Collaborations: Similarities, Differences, and Synergies. *Innovations in Systems and Software Engineering (ISSE) – A NASA Journal*, Band 3, Nr. 3-4, 2007.

Apel, S.; Kuhlemann, M.; Leich, T.: Generic Feature Modules: Two-Staged Program Customization. In *Proceedings of International Conference on Software and Data Technologies (ICSOFT'06)*, S. 127–132, 2006.

Apel, S.; Kolesnikov, S.; Liebig, J.; Kästner, C.; Kuhlemann, M.; Leich, T.: Access Control in Feature-Oriented Programming. *Science of Computer Programming*, Band Special Issue on Feature-Oriented Software Development, 2010.

Apel, S.; Kästner, C.; Leich, T.; Saake, G.: Aspect Refinement. Technischer Bericht Nr. 10, School of Computer Science, University of Magdeburg, Germany, 2006.

Apel, S.; Kästner, C.; Leich, T.; Saake, G.: Aspect Refinement - Unifying AOP and Stepwise Refinement. *Journal of Object Technology (JOT)*, Band 6, Nr. 9, S. 13–33, 2007.

Apel, S.; Leich, T.: Einsatz von Aspektorientierung und Programmfamilien bei der Entwicklung von Datenbank-Management-Systemen. In Höpfner, H.; Saake, G.; Schallehn, E. (Hrsg.): *Tagungsband zum 15. GI-Workshop Grundlagen von Datenbanken 10.-13. Juni 2003, Preprint Nr. 06/2003*, S. 103–107. Fakultät für Informatik, Universität Magdeburg, Deutschland, 2003.

Apel, S.; Liebig, J.; Kästner, C.; Kuhlemann, M.; Leich, T.: An Orthogonal Access Modifier Model for Feature-Oriented Programming. In *Proceedings of the First Workshop on Feature-Oriented Software Development (FOSD)*, S. 27–34. ACM Press, 2009.

Apel, S.; Leich, T.; Rosenmüller, M.; Saake, G.: Combining Feature-Oriented and Aspect-Oriented Programming to Support Software Evolution. In *Proceedings of the 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05)*, in 19th European Conference on Object-Oriented Programming (ECOOP'05). Glasgow, Scotland, 2005.

Apel, S.; Leich, T.; Rosenmüller, M.; Saake, G.: FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++. Technischer Bericht Nr. 3, Fakultät für Informatik, Universität Magdeburg, 2005.

Apel, S.; Leich, T.; Rosenmüller, M.; Saake, G.: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In Glueck, R.; Lowry, M. (Hrsg.): *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005*, Lecture Notes on Computer Science, Band 3676. Springer, 2005.

Apel, S.; Leich, T.; Saake, G.: Aspect Refinement and Bounding Quantification in Incremental Designs. In *Proceedings of the Asia-Pacific Conference on Software Engineering (held at AOASIA'05)*, 2005.

Apel, S.; Leich, T.; Saake, G.: Aspect Refinement in Software Product Lines. In *Aspects and Software Product Lines: An Early Aspects Workshop at SPLC-Europe'05*. Rennes, France, 2005.

Apel, S.; Leich, T.; Saake, G.: Aspectual Mixin Layers. Technischer Bericht Nr. 8, Fakultät für Informatik, Universität Magdeburg, 2005.

Apel, S.; Leich, T.; Saake, G.: Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of IEEE and ACM SIGSOFT 28th International Conference on Software Engineering (ICSE'06)*, 2006.

Apel, S.; Leich, T.; Saake, G.: Aspectual Feature Modules. *IEEE Transactions on Software Engineering*, Band 34, Nr. 2, S. 162–180, 2008.

Apel, S.; Sichting, H.; Leich, T.; Plack, M.: The FATIMA Middleware for Mobile and Pervasive Computing. In Callaos, N.; Lesso, W. (Hrsg.): *Proceedings of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI'05)*, S. 374–379. Orlando, Florida, 2005.

Apel, S.; Sichting, H.; Leich, T.; Plack, M.: On Implementation Techniques for Mobile and Pervasive Middleware Families. In Callaos, N.; Lesso, W. (Hrsg.): *Proceedings of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI'05)*, S. 368–373. Orlando, Florida, 2005.

Feigenspan, J.; Kästner, C.; Apel, S.; Liebig, J.; Schulze, M.; Dachselt, R.; Papendieck, M.; Leich, T.; Saake, G.: Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empirical Software Engineering Journal*, 2012.

Feigenspan, J.; Kästner, C.; Apel, S.; Leich, T.: How to Compare Program Comprehension in FOSD Empirically – An Experience Report. In *Proceedings of the First Workshop on Feature-Oriented Software Development (FOSD)*, S. 55–62. ACM Press, 2009.

Kuhlemann, M.; Apel, S.; Leich, T.: Streamlining Feature-Oriented Designs. In *Software Composition*, S. 168–175, 2007.

Kuhlemann, M.; Leich, T.; Apel, S.: Merkmalorientierte Architekturen für eingebettete Datenmanagementsysteme. In *Workshop Maßgeschneidertes Datenmanagement*, Aachener Informatik-Berichte. RWTH Aachen, 2007.

Kuhlemann, M.; Rosenmüller, M.; Apel, S.; Leich, T.: On the Duality of Aspect-Oriented and Feature-Oriented Design Patterns. In *AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2007.

Kästner, C.; Thüm, T.; Saake, G.; Feigenspan, J.; Leich, T.; Wielgorz, F.; Apel, S.: FeatureIDE: A Tool Framework for Feature-Oriented Software Development. In *Proceedings of the 31th International Conference on Software Engineering (ICSE)*, S. 611–614. IEEE Computer Society, 2009.

Leich, T.; Apel, S.: Ein merkmalsorientierter Speichermanager für eingebettete Systeme. In *17. Workshop "Grundlagen von Datenbanken", Wörlitz (17.05.–20.05.05)*, S. 73–77. Institut für Informatik, Universität Halle-Wittenberg, 2005.

Liebig, J.; Apel, S.; Lengauer, C.; Leich, T.: RobbyDBMS - A case study on Hardware/Software Product Line Engineering. In *Proceedings of the First Workshop on Feature-Oriented Software Development (FOSD)*, S. 60–65. ACM Press, 2009.

Leich, T.; Apel, S.; Marnitz, L.; Saake, G.: Tool Support for Feature-Oriented Software Development - FeatureIDE: An Eclipse-Based Approach. In *Proceedings of OOPSLA Eclipse Technology eXchange (ETX) Workshop*, S. 55–59. San Diego, USA, 2005.

Leich, T.; Apel, S.; Rosenmüller, M.; Saake, G.: Handling Optional Features in Software Product Lines. In *Proceedings of OOPSLA Workshop on Managing Variabilities consistently in Design and Code*. San Diego, USA, 2005.

Leich, T.; Apel, S.; Saake, G.: Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In Eder, J.; Haav, H.-M.; Kalja, A.; Penjam, J. (Hrsg.): *Proceedings of the 9th East-European Conference on Advances in Databases and Information Systems (ADBIS 2005), Tallinn, Estonia, September 12-15, 2005*, Lecture Notes in Computer Science (LNCS). Springer-Verlag, Berlin/Heidelberg, 2005.

Leich, T.; Höpfner, H.: Konzeption eines Anfragesystems für leichtgewichtige, erweiterbare DBMS. In Höpfner, H.; Saake, G. (Hrsg.): *Tagungsband zum Workshop "Mobile Datenbanken und Informationssysteme - Datenbanktechnologie überall und jederzeit"*, Nr. 1, S. 33–37. Fakultät für Informatik, Universität Magdeburg, 2002.

Plack, M.; Leich, T.: Eine Middleware-Architektur für mobile Informationssysteme. In Schubert, S.; Reusch, B.; Jesse, N. (Hrsg.): *Informatik bewegt: Informatik 2002 - 32. Jahrestagung der Gesellschaft für Informatik e.v. (GI), 30. September - 3.Oktober 2002 in Dortmund*, Lecture Notes in Informatics, Band 19, S. 594–598. GI, 2002.

Pukall, M.; Leich, T.; Kuhlemann, M.; Rosenmüller, M.: Highly Configurable Transaction Management for Embedded Systems. In *AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, S. 1–8, 2007.

Rosenmüller, M.; Apel, S.; Leich, T.; Saake, G.: Tailor-Made Data Management for Embedded Systems: A Case Study on Berkeley DB. *Data and Knowledge Engineering (DKE)*, Band 68, Nr. 12, S. 1493–1512, 2009.

Rosenmüller, M.; Kästner, C.; Siegmund, N.; Sunkle, S.; Apel, S.; Leich, T.; Saake, G.: SQL à la Carte – Toward Tailor-Made Data Management. In *Datenbanksysteme in Business, Technologie und Web (BTW)*, S. 117–136, 2009.

Rosenmüller, M.; Leich, T.; Apel, S.: Konfigurierbarkeit für ressourceneffiziente Datenhaltung in eingebetteten Systemen am Beispiel von Berkeley DB. In *Datenbanksysteme in Business Technologie und Web - Workshop*, 2007.

Rosenmüller, M.; Leich, T.; Apel, S.; Saake, G.: Von Mini- über Micro- bis zu Nano-DBMS: Datenhaltung in eingebetteten Systemen. *Datenbank Spektrum*, Band 7, Nr. 20, 2007.

Rosenmüller, M.; Siegmund, N.; Schirmeier, H.; Sincero, J.; Apel, S.; Leich, T.; Spinczyk, O.; Saake, G.: Fame-DBMS: Tailor-Made Data Management Solutions for Embedded Systems. In *Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, 2008.

Schäler, M.; Leich, T.; Rosenmüller, M.; Saake, G.: Building Information System Variants with Tailored Database Schemas using Features. In *24th International Conference on Advanced Information Systems Engineering (CAiSE)*, LNCS, Band 7328, S. 597 – 612. Springer, 2012.

Schäler, M.; Leich, T.; Siegmund, N.; Kästner, C.; Saake, G.: Generierung maßgeschneiderter Relationenschemata in Softwareproduktlinien mittels Superimposition. In *14. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web*, LNI, Band P-180, S. 414–534. GI, 2011.

Saake, G.; Rosenmüller, M.; Siegmund, N.; Kästner, C.; Leich, T.: Downsizing Data Management for Embedded Systems. *Egyptian Computer Science Journal (ECS)*, Band 31, Nr. 1, S. 1–13, 2009.

Soffner, M.; Siegmund, N.; Rosenmüller, M.; Feigenspan, J.; Leich, T.; Saake, G.: A Variability Model for Query Optimizers. In *International Baltic Conference on Databases*

and Information Systems. IOS Press, 2012.

Thüm, T.; Kästner, C.; Benduhn, F.; Meinicke, J.; Saake, G.; Leich, T.: FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 2012.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

2PL . . . . . *Two Phase Locking*

ACCORD . . . . . *Aspectual Component-based Real-Time System Development*

AHEAD . . . . . *Algebraic Hierarchical Equations for Application Design*

AOP . . . . . *Aspektorientierte Programmierung*

AST . . . . . *Abstract Syntax Tree*

C2PL . . . . . *Conservative Two Phase Locking*

CCS . . . . . *Car-Control-Systems*

DBI . . . . . *Datenbankimplementierer*

DBMS . . . . . *Datenbankmanagementsystem*

FODA . . . . . *Feature-Oriented Domain Analysis*

FOP . . . . . *Feature-Oriented Programming*

FOSD . . . . . *Feature-Oriented Software Development*

IECU . . . . . *Instrumental Electronic Control Unit*

LOC . . . . . Lines of Code

MVCC . . . . . *Multiversion Concurrency Control*

PC . . . . . *Personal Computer*

PUMA . . . . . *Pure Manipulator*

S2PL . . . . . *Strict Two Phase Locking*

VECU . . . . . *Vehicle Electronic Control Unit*

KAPITEL 1

Einleitung

Bedeutung eingebetteter Systeme

Das Gebiet der eingebetteten Systeme ist eines der stark expandierenden Teilgebiete der Informatik [Com00] in den letzten 10 Jahren. Etwa 98 Prozent aller weltweit hergestellten Rechnersysteme sind als eingebettete Systeme im Einsatz [Ten00, Tur02]. *Pervasive* und *Ubiquitous Computing* [Wei93], intelligente Sensornetzwerke [BPC+07, TM03, AWSC02, GGP+03, Dre07b, Dre07a, Dre08], *Smart Dust* [WLLP01, KKP99] bis hin zu *Cyber Physical Systems* [Lee08] versprechen diesen Trend auch in Zukunft fortzusetzen. Als eingebettete Systeme werden Rechnersysteme bezeichnet, die mit Hilfe einer zugehörigen Software ihre Aufgabe in einem (übergeordneten) Produkt verrichten. Häufig erledigen sie dabei ihre Aufgaben für den Nutzer vollkommen transparent.

Softwareentwicklung von eingebetteten Systemen

Bei der Entwicklung dieser Systeme sind die Kosten für die Hardware auf Grund der zumeist hohen Stückzahlen ein sehr wichtiger Faktor. Deshalb kommt es bei der Softwareentwicklung darauf an, den Ressourcenbedarf zu minimieren, um so preisgünstige Hardware einsetzen zu können. Dominierende Programmiersprachen im Bereich der eingebetteten Systeme sind immer noch Assembler und C [Beu03], S. 14. Vor allem in der Praxis spricht derzeit viel für den Einsatz dieser Sprachen, da nur durch die systemnahe Programmierung in extremer Ressourcenbeschränkung ein für den Entwickler nachvollziehbarer schonender Umgang mit knappen Ressourcen ermöglicht wird.

Restriktionen bei der Datenhaltung

Beispielanwendungen aus dem Bereich des Automobilbaus [NTN+04] und der Sensornetzwerke [WMG04, GGP+03, MCP+02] zeigen, dass eingebettete Systeme stark variierende Elemente klassischer Infrastruktursoftware zur Datenhaltung benötigen. Des Weiteren gilt der sparsame Umgang mit den knappen Ressourcen als ein wichtiges Optimierungsziel. Wir bezeichnen dieses stark variierende und auf die beschränkten Ressourcen von eingebetteten Systemen ausgerichtete *Datenbankmanagementsystemen*

(DBMS) als *Nanodatenmanagement* [RLAS07]. Eine Adaption vorhandener Mehrzwecksystemlösungen ist für diesen Bereich des Datenmanagements aus dem Großrechner- oder dem *Personal Computer* (PC)-Bereich auf Grund der Heterogenität der Hard- und Software sowie der extremen Ressourcenbeschränkungen nicht möglich, wie OLSON [Ols00] feststellt:

> *„The hundreds of embedded operating systems available today run on an amazing variety of processor hardware, which makes choosing a good database engine for an embedded application harder than choosing one for a desktop or server system."*
> OLSON [Ols00] S. 28

Eine im Rahmen der Dissertation angefertigte Analyse sowie existierende Analysen [NTN+04] von vorhandenen Forschungsprototypen und kommerziellen DBMS beziehungsweise Datenhaltungskomponenten für den Bereich der eingebetteten Systeme zeigt, dass der Großteil der Basis dieser DBMS auf Forschungsergebnissen der späten achtziger und frühen neunziger Jahre basiert. Neue Methoden aus dem Bereich des Software-Engineering, insbesondere Produktlinientechniken sind nicht systematisch im Umfeld von Datenbankmanagementsystemen wissenschaftlich untersucht worden.

Anpassung der Softwaretechnik

Eine Anpassung der Entwicklung von Datenhaltungssystemen an die aktuellen Entwicklungen in der Softwaretechnik oder der Programmiersprachenentwicklung hat nur in Ansätzen stattgefunden. Ein Grund hierfür dürfte sein, dass Softwareentwicklung in eingebetteten Systemen sowohl in der Softwaretechnik als auch in der Programmiersprachenentwicklung nicht im Fokus der Forschungsbemühungen steht und die speziellen Bedürfnisse von eingebetteten Systemen bezüglich der Hardwarerestriktionen kaum Beachtung finden. Das Resultat in der Praxis ist die wiederkehrende Entwicklung ähnlicher Datenhaltungskomponenten als Teil der Anwendungssoftware, die für den konkreten Anwendungsfall zugeschnitten sind und somit keinen unnötigen Speicher oder Rechenzeit verbrauchen. Dieses ist nicht nur sehr kosten- sowie zeitintensiv und verhindert eine schnelle Markteinführung, sondern führt auch häufig zu deutlichen Qualitätseinbußen der Software [PBL05]. Die Hauptgründe liegen vor allem in den eingesetzten Programmiersprachen und -konzepten Assembler und C, die nachweislich Defizite bezüglich einer systematischen Wiederverwendung haben [CE00]. Moderne Konzepte, die auf dem heutigen – in anderen Bereichen längst gängigen – Konzept der Objektorientierten Programmierung beruhen, scheitern bisher immer noch im Bereich der eingebetteten Systeme, weil die konzeptbedingten *Overhead*-Kosten noch immer den Einsatz von Sprachen wie Assembler oder C rechtfertigen. Um hier eine nachhal-

tige Änderung zu erreichen und auf die neuen Anforderungen aus dem Bereich des *Pervasive* und *Ubiquitous Computing*, *Smart Dust* und *Cyber Physical Systems* [Lee08] zu reagieren, muss zum einen die immer noch nicht ausreichende Wiederverwendung in der Objektorientierten Programmierung verbessert werden. Zum anderen ist es erforderlich, den konzeptbedingten *Overhead*, der vor allem durch eine höhere Flexibilität beziehungsweise erhöhte Wiederverwendung erzeugt wird, zu minimieren.

Produktlinien und Programmfamilien

Die Skalierbarkeit und Wiederverwendung von Softwareeinheiten ist schon seit Jahrzehnten ein zentraler Forschungsschwerpunkt der Softwaretechnik und wurde erstmals von DIJKSTRA unter dem Fakt der Softwarekrise zusammengefasst [Dij72]. Neue Ansätze aus dem Software-Engineering insbesondere aus den Bereichen der Komponententechniken, Produktlinien, Programmfamilien und der Merkmalsorientierten Softwareentwicklung versprechen Hilfe, um die beschriebenen Schwierigkeiten zu überwinden. Allerdings wurden diese Methoden bislang nur in einem sehr begrenzten Maße an Infrastruktursoftware für eingebettete Systeme erprobt, das heißt sie sind derzeitig nur eingeschränkt einsetzbar.

1.1 Zielsetzung der Arbeit

Ziel des vorgestellten Dissertationsvorhabens ist es, den Bereich der variablen Nanodatenmanagementlösungen zu untersuchen und auf der Basis von Produktlinientechniken zu ergründen, inwieweit der Entwicklungsprozess und die derzeitig eingesetzten Methoden und Techniken verbessert werden können.
Dabei liegt aus datenbanktechnischer Sicht der Hauptforschungsaspekt auf dem Überdenken vorhandener Dekompositions- und Implementierungsstrukturen beziehungsweise der Anpassung von vorhandenen modularen Strukturen von konfigurierbaren DBMS auf moderne Produktlinientechnologientechnologien. Aus software- beziehungsweise programmiertechnischer Sicht müssen Methoden, Werkzeuge und Sprachen auf die speziellen Belange des Bereiches der eingebetteten Systeme adaptiert werden.

1.2 Wissenschaftlicher Beitrag der Arbeit

Der wissenschaftliche Beitrag kann zwei Forschungsdisziplinen der Informatik zugeordnet werden. Zum einen dem Bereich Datenbanken und zum anderen dem Bereich des Software-Engineering. Die folgenden vier Punkte fassen den Beitrag der Arbeit zusammen:

1. **Analyse, Einordnung und Bedeutung des variablen Nanodatenmanagements für eingebettete Systeme:** Beispielhaft zeigen wir typische Anwendungsgebiete vom Nanodatenmanagement im Bereich der eingebetteten Systeme und die verschiedenen Anforderungen an das Nanodatenmanagement. Des Weiteren grenzen wir uns zu Forschungen in Bezug auf variables Datenmanagement im Bereich von klassischen DBMS-Lösungen ab, das sehr stark durch Forschungsarbeiten in den achtziger und neunziger Jahren beeinflusst wurde. Darüberhinaus zeigen wir Probleme an existierenden Lösungen, die dem Bereich des Nanodatenmanagements bezüglich einer mangelnden feingranularen Variabilität zugeordnet werden können. Diese können somit nicht ausreichend auf die knappen Ressourcen reagieren.

2. **Analyse von Problemen bei der Umsetzung des variablen Nanodatenmanagements mit aktuellen Produktlinientechniken:** Wir schlagen zur Umsetzung des variablen Nanodatenmanagements Produktlinientechniken auf Basis der Merkmalsorientierten Programmierung vor. Die Merkmalsorientierte Programmierung, vorgestellt von PREHOFER [Pre97] und BATORY et al. [BSR03], ermöglicht auf Basis der Objektorientierten Programmierung die Abbildung von Merkmalen und deren Variabilität. Mit Hilfe einer explorativen Fallstudie evaluieren wir systematisch die Probleme bei der Umsetzung des variablen Nanodatenmanagements mit Produktlinientechnologien.

3. **Integration der Merkmalsorientierten und Aspektorientierten Programmierung für C++:** Überwiegend wendeten bisher die Forscher bei der Umsetzung von variabler Datenmanagementsoftware bestehende Software- und Programmiertechniken an (vergleiche hierzu beispielsweise *TinyDB* [MFHH05a], *PicoDBMS* [PBVB01], *XXL-Liberay* [BBD+01], *COMET DBMS* [MFHH05b] und *Cougar* [YG02], *DASDBS* [PSS87], *EXODUS* [CDRS86], *KIDS* [GSD97], *Texas* [SKW92]). Ähnlich zu den variablen Datenbankansätzen *GENESIS* [Bat86] und *P2* [BT97] aus den neunziger Jahren erweitert unser Ansatz auch Methoden des Software-Engineering, um auf die Anforderungen des feingranularen variablen Datenmanagements für eingebettete Systeme zu reagieren. Hierzu erweitern wir die Merkmalsorientierte Programmierung für C++ und integrieren die Aspektorientierte Programmierung, um auf die beschriebenen Probleme aus unserer explorativen Fallstudie zu reagieren. Als Lösung präsentieren wir FEATUREC++ als eine Erweiterung von C++ und FEATUREIDE als eine integrierte Werkzeugunterstützung für die Merkmalsorientierte Softwareentwicklung.

4. **Evaluierung von Produktlinientechniken zum Umsetzung des variablen Nanodatenmanagements:** Im Schlussteil der Arbeit evaluieren wir an drei Fallstudien unseren Produktlinienansatz für variables Nanodatenmanagement. Am Beispiel von FAMEDBMS zeigen wir in Form einer Referenzimplementierung eine Produktlinie des variablen Nanodatenmanagements, die auch die Basis für die weitere konkrete Umsetzung ist. Mit ROBBYDBMS präsentieren wir eine extrem feingranulare Produktlinie von Nanodatenmanagement zur Evaluierung der Grenze unseres Ansatzes. Mit der dritten Fallstudie zeigen wir durch eine merkmalsorientierte Refaktorisierung am Beispiel von Berkeley DB, dass wir mit unserem Ansatz feingranulare Variabilität ermöglichen können, ohne Performanz- oder Größennachteile in Kauf zu nehmen.

1.3 Kapitelüberblick

Die in dieser Dissertation bearbeiteten Forschungsziele werden in der folgenden Kapitelstruktur vorgestellt:

Kapitel 2: Im Kapitel zwei geben wir einen Überblick zum Thema Datenmanagement in eingebetteten Systemen. Hierzu führen wir zunächst in die wichtigsten Eigenschaften von eingebetteten Systemen ein und zeigen an zwei Beispielszenarien unterschiedliche Formen und Ausprägungen des Datenmanagements in diesem Umfeld. Darauf folgend geben wir einen Literaturüberblick zu variablen und anpassbaren Datenmanagementsystemen.

Kapitel 3: Im dritten Kapitel führen wir in die Grundlagen der Variabilitätsgestaltung und Wiederverwendung von Softwareeinheiten in der Softwareentwicklung ein. Der Fokus liegt auf der Definition wichtiger und für die Arbeit notwendiger Begriffe.

Kapitel 4: An Hand einer explorativen Studie wird im Kapitel vier der Einsatz der Merkmalsorientierten Softwareentwicklung im Umfeld des hochkonfigurierbaren Datenmanagements für eingebettete Systeme evaluiert. Hierzu wird auf Basis einer idealisierten Fallstudie aus dem Bereich der Sensornetzwerke ein Speichermanager entworfen, implementiert und evaluiert. Aufbauend auf diesen Ergebnissen werden explorative Forschungsfragen für dieses Umfeld aufgezeigt, die wir im weiteren Verlauf der Arbeit diskutieren.

Kapitel 5: Aufbauend auf den in Kapitel vier beschriebenen Problemen hinsichtlich der Implementierungsebene wird im fünften Kapitel die Umsetzung von FEATUREC++ präsentiert. Unsere Erweiterung der Merkmalsorientierten Programmierung auf der Basis von C++ fokussiert neben der Vermeidung von *Overhead* insbesondere auf Probleme mit querschneidenden Belangen, nicht-hierarchiekonformen Erweiterungen aus ungeplanten Änderungen und der Merkmalskohäsion im Bereich der Interaktion von optionalen Merkmalen auf Implementierungsebene in hierarchischen Softwaredekompositionen.

Kapitel 6: Nicht alle Probleme der Merkmalsorientierten Programmierung, die wir in unserer explorativen Studie aufgezeigt haben, lassen sich auf Sprachebene lösen. Im sechsten Kapitel stellen wir deshalb FEATUREIDE auf Basis von Eclipse als eine integrierte Entwicklungsumgebung zur Unterstützung der Merkmalsorientierten Softwareentwicklung vor.

Kapitel 7: Im Kapitel sieben präsentieren wir drei verschiedene Fallstudien, die den Bau von variabler Datenmanagementsoftware für eingebettete Systeme aus unterschiedlichen Richtungen evaluieren. Als eine Art Referenzarchitektur wird zunächst FAMEDBMS bestehend aus einem Speichermanager, einer Transaktionsverwaltung und einer Anfrageoptimierung vorgestellt. Mit ROBBYDBMS untersuchen wir in der zweiten Studie den Bereich der tief eingebetteten Systeme hinsichtlich der mobilen autonomen Roboter mit einer extremen Ressourceneinschränkung. Diesbezüglich soll die Skalierbarkeit der Merkmalsorientierten Programmierung mit FEATUREC++ untersucht werden. In der dritten Studie wird Berkeley DB auf Basis von FEATUREC++ merkmalsorientiert refaktorisiert. In einem vergleichenden Benchmark wird dann die originale Version von Berkeley DB auf der Basis von C mit unserer refaktorisierten Version verglichen, die eine bessere Variabilität und keine Nachteile bezüglich der Performanz aufweist.

Kapitel 8: Im achten Kapitel schließen wir die Arbeit mit einer Zusammenfassung der Ergebnisse der Arbeit und diskutieren offene Forschungsfragen aus dem Kontext der Ergebnisse, die im Rahmen dieser Arbeit nicht bearbeitet wurden.

KAPITEL 2

Datenmanagement in eingebetteten Systemen

*Dieses Kapitel enthält Ergebnisse der Veröffentlichungen des Datenbank-Spektrums 2007 „Von Mini- über Micro- bis zu Nano-DBMS: Datenhaltung in eingebetteten Systemen“(*ROSENMÜLLER, LEICH, APEL, SAAKE *[RLAS07]).*

Glaubt man optimistischen Wachstumsraten, wächst der Markt für eingebettete Systeme um bis zu 18 Prozent pro Jahr [BCC+05, Sch05, HKM+05]. Einhergehend mit dieser Entwicklung ist in den letzten Jahren das Datenaufkommen, das von diesen Systemen verarbeitet wird, gewachsen. In Folge dieses stetig wachsenden Datenaufkommens benötigen diese, teilweise aus Kosten- und Ressourcengründen, sehr leistungsschwachen Geräte ein adäquates Datenmanagement. Anders als im Bereich der Datenbankserver mit Datenvolumina im Tera- und Petabyte-Bereich, in dem die Mehrzwecklösungen etablierter Datenbankhersteller einen größtmöglichen Einsatzbereich abdecken müssen, ist die Verarbeitung von Daten im Bereich der eingebetteten Systeme als Teil der Infrastruktursoftware vielmehr eine Frage der Variation, Spezialisierung und Leichtgewichtigkeit. Dieses Kapitel zeigt Anforderungen an Variations- und Ausprägungsformen von Datenmanagementfunktionalität im Bereich der eingebetteten Systeme. Hierzu werden nach einer Einführung in das Gebiet der eingebetteten Systeme zwei Beispielszenarien vorgestellt, die exemplarisch verdeutlichen, welcher Funktionsumfang und welches Leistungsspektrum an Datenmanagementfunktionalität in diesem Umfeld benötigt wird. Zum einen wird hierzu der Automobil- sowie Nutzfahrzeugbereich und zum anderen der Bereich der drahtlosen Sensornetzwerke analysiert.

Bereits in den achtziger und neunziger Jahren des vorherigen Jahrhunderts gab es Forschungsaktivitäten, die im Gegensatz zu Mehrzwecklösungen eine Anpassbarkeit beziehungsweise Erweiterbarkeit von vorhandenen Systemen zuließen. In einer Klassifizierung werden repräsentative Vertreter vorgestellt und deren mangelnde Eignung für die zuvor beschrieben Einsatzszenarien diskutiert. Anschließend wird ein Überblick zum aktuellen Stand der Forschung im Bereich des Datenmanagements von eingebetteten Systemen gegeben. Darauf aufbauend werden die Anforderungen an ein variables und erweiterbares Datenmanagement definiert, die in dieser Dissertation betrachtet werden.

2.1 Eingebettete Systeme

Technische Definition

Die Definitionen eines eingebetteten Systems variieren je nach Blickwinkel auf die Thematik. Zum einen gibt es die technische Sichtweise, wie beispielweise von BARR [Bar99]:

> *"An embedded system is a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a specific function."*
> BARR *[Bar99], S. 1*

Diese sieht in einem eingebetteten System eine Kombination von Hard- und Software, die für einen speziellen Aufgabenbereich entwickelt wird.

Anwendungsorientierte Definition

Zum anderen gibt es die anwendungsorientierte Sichtweise, wie beispielsweise von MARWEDEL [Mar03] vertreten:

> *"Embedded systems can be defined as information processing systems embedded into enclosing products such as cars, telecommunication or fabrication equipment."*
> MARWEDEL *[Mar03], S. 1*

Diese sieht ein eingebettetes System als informationsverarbeitendes System, eingeschlossen in ein übergeordnetes Gesamtsystem, an.

Transparenz

Häufig sind die verrichteten Funktionen in dem übergeordneten Produkt für den Anwender völlig transparent. Nach einer Schätzung in MARWEDEL [Mar07], S. 9 kam bereits im Jahr 1996 jeder US-Bürger 60 mal täglich mit Mikroprozessoren aus einem eingebetteten System in Berührung. Selten dürfte er dabei das eingebettete System bemerkt haben.

Komplexität der Anwendungsbereiche

Während Anfang der neunziger Jahre der Mikroprozessor einer Waschmaschine fest vorgegebene Waschprogramme steuerte, besitzen moderne Waschmaschinen heute eingebettete Rechnersysteme, die beispielsweise aus Beladungszuständen, Wassertemperatur, Wasserhärtegrad, Waschmittelmenge und weiteren Parametern ein möglichst ökologisches Waschprogramm während des Waschvorgangs berechnen und bedarfsgerecht steuern. Wurden früher durch eingebettete Systeme einfache Prozesse angestoßen, so werden heute komplexe Modelle berechnet, die entsprechende komplexere Prozesse steuern und überwachen. Ein bisher unzureichend gelöstes Problem ist die Entwicklung und Beherrschung dieser komplex vernetzten eingebetteten Systeme. Nach einer Studie der EU-Kommission aus dem Jahr 2005 betragen die Entwicklungskosten für eingebettete Systeme im Automobil- und Flugzeugbau mittlerweile mehr als 50 Prozent der Gesamtkosten [HKM+05], S. 32. Ein Grund für diese hohen Kosten sieht VAANDRAGER in [RV96], der in Anlehnung an das Mooresche Gesetz folgendes formulierte:

> *"For many products in the area of consumer electronics the amount of code is doubling every two years."* VAANDRAGER *in [RV96], S. 1*

Beispiel für komplexe eingebettete Systeme

Zu den komplexesten Systemen zählen heute Flugzeuge oder U-Boote, die aus tausenden eingebetteten Systemen bestehen. Laut eines NATO-Reports [BCC+05] zum Thema *"Embedded Systems"* verrichten in einem modernen U-Boot der US-Streitkräfte eingebettete Systeme mit einem Softwarevolumen von 30.000.000.000 Zeilen Programmcode ihre Arbeit. Dieser Quellcode ist in 142 unterschiedlichen Programmiersprachen entwickelt worden [BCC+05]. Dass die Hardware dieser Systeme dabei stark heterogen ist und ein hoher Grad an Vernetzung dieser Systeme vorliegt, erhöht die Komplexität des Gesamtsystems zusätzlich. Ob diese Systeme noch beherrscht werden, wird in dem Bericht skeptisch betrachtet [BCC+05].

Effizienz und Verlässlichkeit

Zwei der wichtigsten Eigenschaften, die von eingebetteten Systemen gefordert werden, sind *Verlässlichkeit* und *Effizienz* [Mar07], S. 2. Gerade der Einsatz in sicherheitskritischen Anwendungsbereichen wie beispielsweise Atomkraftwerken, Flugzeugen, Zügen, Autos usw. fordert von den Systemen: [Mar07], S. 2

- Zuverlässigkeit,
- Verfügbarkeit,
- Wartbarkeit,
- Sicherheit,

- Integrität.

Tief eingebettete Systeme

Der Einsatz in *tief eingebetteten Systemen* (engl. *deeply embedded systems*) [BGP+99] fordert neben Verlässlichkeit noch den sparsamen Umgang mit Ressourcen. Wichtige Parameter hierbei sind [Mar07], S. 2 f.:

- Codegröße,
- Laufzeit-Effizienz,
- Energie-Effizienz,
- Gewicht,
- Preis.

Datenaufkommen

Als Nebeneffekt der ständig komplexeren Prozessbearbeitungen, die durch eingebettete Systeme gesteuert werden, steigt auch das Datenaufkommen, das durch diese Systeme verarbeitet werden muss. Klassische Fragestellungen wie zum Beispiel logische und physische Datenunabhängigkeit, integrierte Datenbestände, Transaktionen, effizienter Zugriff auf Daten und sichere, konsistente Speicherung aus dem DBMS-Bereich werden immer mehr in Szenarien von eingebetteten Systemen diskutiert. Im folgenden Abschnitt werden Beispielszenarien und die dort benötigten Datenmanagementfunktionalitäten thematisiert.

2.2 Eingebettete Systeme im Automobilbau

Eingebettete Systeme in Automobilen

Viele Funktionen im Umfeld von modernen Automobilen werden durch eingebettete Systeme kontrolliert oder sogar vollständig gesteuert. Dabei ist seit vielen Jahren der Trend festzustellen, dass zunehmend mechanische Steuerungssysteme durch elektronische oder mechatronische Systeme ersetzt werden [SZ06] S. 2 ff.. Haben in einen VW Golf der IV. Generation (2001) je nach Ausstattung ca. 13 vernetzte Steuergeräte ihre Arbeit verrichtet, so waren es im Golf V vier Jahre später über 40 Steuergeräte [BS06], S. 628. Die Gründe hierfür sind vielschichtig. Steigende Kundenansprüche und strenge gesetzliche Vorgaben wie beispielsweise die Verringerung des Kraftstoffverbrauchs und der Schadstoffemissionen sowie der Wunsch nach einer erhöhten Fahrsicherheit und einem erhöhten Fahrkomfort können heutzutage nur noch durch elektronische Systeme umgesetzt werden [SZ06, Rei06, BS06]. SCHÄUFFELE und ZURAWKA [SZ06], S. 1 bezeichnen das Automobil durch die Kombination einer Vielzahl von mechanischen und elektronischen Komponenten als das komplexeste Konsumgut unserer Zeiten. Diese

auch als *Car-Control-Systems* (CCS) bezeichneten Systeme überwachen nicht nur das Fahrzeug, sondern steuern aktive, in vollem Umfang des englischen Wortes „*control*“, Prozesse im Fahrzeug.

Probleme durch zunehmende Komplexität

Eines der größten Probleme der Automobilbauer in diesem Umfeld ist die zunehmende Komplexität dieser Systeme. Durch die Übernahme weiterer Aufgaben in diesem Umfeld werden nicht nur die einzelnen CCS komplexer, sondern auch das Zusammenspiel dieser Systeme. Eines der bekanntesten Probleme nach der Jahrtausendwende dürfte in diesem Umfeld das Qualitätsproblem des deutschen Automobilherstellers Mercedes Benz sein. Dem Stuttgarter Unternehmen und seinen Zulieferern gelang es über Jahre nicht, dass komplexe Zusammenspiel dieser CCS in der E-Klasse zu beherrschen. Durch die Vielzahl von elektronischen Systemen kollabierte bei vielen Fahrzeugen die komplette Bordelektronik inklusive der Batterie[1]. Eine Konsequenz aus diesen Fehlern war sogar die Rückrüstung auf die alt bewährte Technik. Das als großer Entwicklungssprung der E-Klasse gepriesene Sensotronic Brake System[2] (SBC-Bremse) wurde in der Nachfolgegeneration des Fahrzeuges auf das herkömmliche System zurückgerüstet. Die Ursachen sind eindeutig identifiziert worden. Sie lagen mehr im mangelhaften komplexen Zusammenspiel von Softwaresystemen und weniger an den elektronischen Bauteilen selbst. Dennoch wird auch in Zukunft eines der dominierenden Themen im automobilen Umfeld die Entwicklung dieser softwaregestützten Systeme sein. Eine Ursache für die immer wiederkehrenden Probleme durch fehlerhafte Software ist die mangelnde Wiederverwendung von bereits implementierten Softwarebausteinen.

Ansteigendes Datenvolumen im Fahrzeug

Einhergehend mit der steigenden Funktionalität, die durch solche CCS im Fahrzeug gesteuert und überwacht wird, wächst auch das Datenaufkommen. In einer Studie von Volvo wurde ermittelt, dass das zu verarbeitende Datenaufkommen in einem Fahrzeug jährlich um 7 bis 10 Prozent steigen wird [CRM98]. Vergleicht man die in einem Automobil anfallenden Datenmengen mit denen typischer Datenbankanwendungen in einem Unternehmen, so sind die anfallenden Datenmengen im Auto nur sehr gering. Die wirkliche Herausforderung wird erst sichtbar, wenn man die zur Verfügung stehende Rechenleistung zu Vergleichszwecken mit heranzieht. In modernen Autos dominieren hocheffiziente und kostengünstige 8- und 16-Bit Systeme. Einige wenige Systeme sind mit 32-Bit Prozessoren ausgerüstet. Hier ist in Zukunft wenig Veränderung zu erwarten, da der sparsame Umgang mit Energie und nicht zuletzt auch die Kosten für solche eingebetteten Systeme einen entscheidenden Einfluss nehmen. In modernen Fahrzeugen

1 AUTO BILD 13/2005, 01.04.2005: Interview mit Mercedes-Benz-Chef Eckhard Cordes.

2 Das elektrohydraulische Bremssystem SBC vereint die Funktionen des Bremskraftverstärkers, des Antiblockiersystems und der Fahrdynamikregelung (auch ESP genannt) [SZ06, PB02].

können die CCS in vier Subsysteme unterteilt werden [SZ06], S. 6:

Antriebsstrang: Motor-, Abgas- und Getriebesteuerung,

Fahrwerk: Antiblockiersystem (ABS), elektronisches Stabilitäts- und Traktionsprogramm (ESP), elektrohydraulische oder pneumatische Fahrwerksysteme und Lenkung,

Karosserie: Heizungs- und Klimatisierungssteuerung, Zentralverriegelung, Sitz- und Spiegelverstellung sowie Airbag- und Rückhaltesysteme und

Multimediasystem: Navigations- und Entertainmentsysteme und Telefonsystem.

Zunehmende Vernetzung der Subsysteme

Die Steuergeräte in den einzelnen Subsystemen sind im Allgemeinen hochgradig vernetzt beziehungsweise in einem Steuergerät integriert. In den letzten Jahren nahm aber ebenfalls die Vernetzung der Subsysteme untereinander zu. So sind in modernen Oberklassefahrzeugen Motor- und Getriebesteuerung eng mit Fahrwerksystemen (ABS, ESP, elektronisch geregelte Differenzialsperren) gekoppelt, um den Bedürfnissen des Fahrers auf Knopfdruck nach sportlichen oder komfortablen Fahrprogrammen zu entsprechen.

Integrierte Datenhaltung

Immer häufiger stößt das bisherige Vorgehen, dass die einzelnen Applikationen und Subsysteme ihre Daten in den Anwendungen selbst verwalten, auf Grenzen. Durch die starke Vernetzung der Systeme untereinander ist der Datenaustausch stetig gestiegen. So entsteht die Forderung nach einer integrierten Verwaltung der Daten [CRM98].

Steuersysteme im Fahrzeug

Die Aufgaben, die durch Datenmanagementfunktionalität abgedeckt werden, sind sehr unterschiedlich. Heutige Techniken, bei denen ein CCS seine Daten losgelöst von den anderen Systemen in interne Speicherstrukturen ablegt und manipuliert, sind bei der zunehmenden Vernetzung der Systeme nicht mehr durchzusetzen. Vielmehr besteht immer häufiger die Notwendigkeit, dass unterschiedliche Systeme gleichzeitig einen konsistenten und effizienten Zugriff auf die Daten benötigen.

2.2.1 Datenaufkommen des Antriebsstrangs

Motor- und Getriebe-steuerung

Typische, eng vernetzte, eingebettete Systeme im Fahrzeug sind die Motor-, Abgas- und Getriebesteuerung. Im Gegensatz zu den Karosserie- und Multimediasystemen werden sie vom Nutzer nicht wahrgenommen. In den letzten zehn Jahren ist die Software dieser Geräte immer komplexer geworden. Verantwortlich hierfür sind vor allem gestiegene Anforderungen im Bereich Emissionsschutz und Ressourcenkonsum, aber auch die

gestiegenen Anforderungen an den Fahrkomfort oder die Verlängerung der Wartungs- und Serviceintervalle [SZ06] S. 8 ff..

Zusammenspiel von Motor-, Abgas- und Getriebe-steuerung

Im komplexen Zusammenspiel müssen Motor-, Abgas- und Getriebesteuerung in einem breiten Temperaturband ein nahezu gleiches Verhalten aufweisen. So wird beispielsweise die Getriebesteuerung eines Automatikgetriebes in der Warmlaufphase durch spätere Schaltzeitpunkte und längere Schaltphasen angepasst, um den Motor und das Getriebe schneller auf Temperatur zu bringen und dem Fahrer das gewohnte Beschleunigungsverhalten zu suggerieren. Dies passiert ebenfalls bei Fahrten im Gebirge, bei denen die Motorleistung durch den geringeren Luftdruck abnimmt, aber ähnliche Fahrleistungen wie im Flachland gewährleistet werden sollen [SZ06] S. 8 ff.. Unter anderem werden diesbezüglich zur Steuerung des Motors folgende Daten von Sensoren benötigt [SZ06, Rei06, BS06]: Luftmasse/-menge, Winkel-/Drehzahlgeber von Kurbel- und Nockenwelle(n), Drosselklappenstellung, barometrischer Umgebungsluftdruck, Signal der Lambdasonde(n), Kraftstoffdrucksignal, Kühlmitteltemperatur, Temperatur und Druck des Motoröls, Temperatur der angesaugten Luft, Klopfsensor, Batteriespannung und Fahrgeschwindigkeit. Neben diesen Daten, die den IST-Zustand des Motors repräsentieren, gibt es ebenfalls Sensorsignale, die den SOLL-Zustand des Fahrers abfragen. Hierzu geben Sensoren Daten an die Motorsteuerung weiter wie beispielsweise Gaspedalwinkel, Bremssignal, Kupplungspedalschalter und Daten des Fahrgeschwindigkeitsregelungssystems (Tempomat).

Daten im Bereich der Motor- und Getriebe-steuerung

Unter Berücksichtigung dieser Eingangssignale werden mit Hilfe der hinterlegten Kennlinien und -felder Ausgangssignale errechnet beziehungsweise den entsprechenden Stellgliedern Sollgrößen vorgegeben. Zu diesen berechneten Daten gehören: Einspritzmenge, -zeitpunkt, Zündwinkel, -zeitpunkt, -energie, Drosselklappenstellung, Ladungsbewegungsklappen, Nockenwellenverstellung, Ventilhub, Abgasrückführventil, Tankentlüftungsventil, Kompressoransteuerung, Turboladeransteuerung, Kraftstoffpumpe, Generatorerregung und die Klimakompressor-Kupplung. Des Weiteren werden Parameter zur Ansteuerung und Kalibrierung von Akteuren wie beispielsweise der Drosselklappe, Ein- und Auslassventile, Abgasrückführung und des Turboladers benötigt. Die Abbildung 2.1 zeigt den komplexen Datenfluss der verschiedenen Subsysteme einer Motorsteuerung. Besonders die Vielzahl an verschiedenen Sensordaten zeigt die Komplexität und das hohe Datenaufkommen.

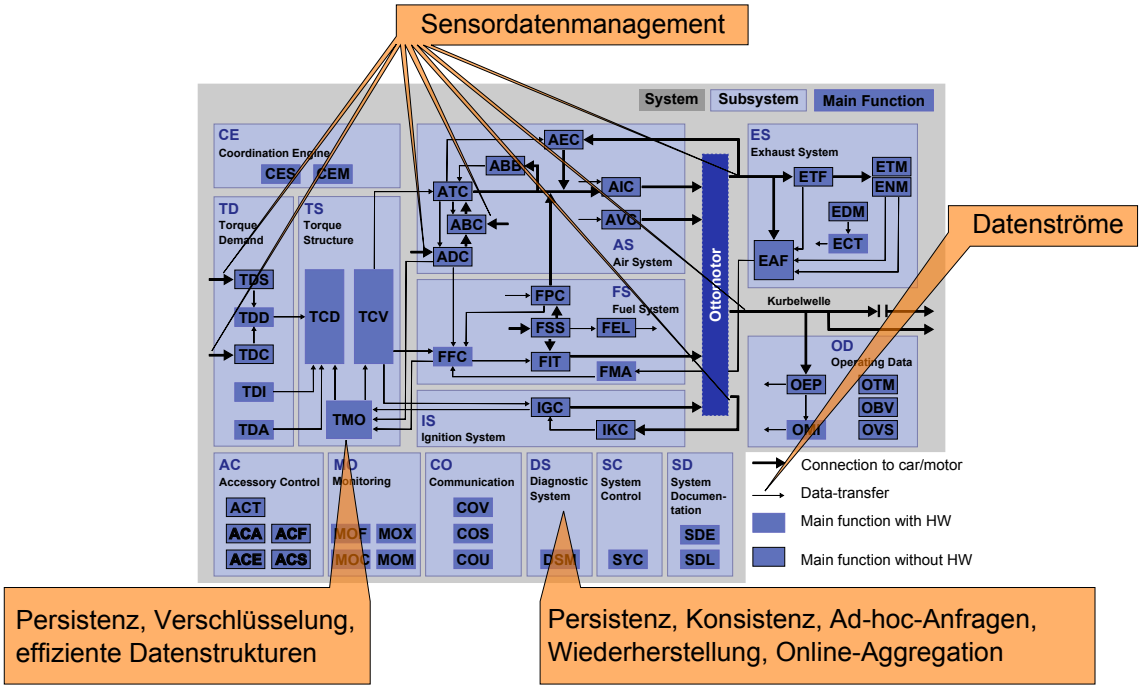

Abbildung 2.1: Datenfluss zwischen Sensoren und Aktoren in einer Bosch Motorsteuerung [Bau03], S. 39 (verändert)

Weitere Anforderungen an das Datenmanagement

Eine effiziente Transaktionsverwaltung, die den Echtzeitkontext berücksichtigt, gilt im Motormanagement als unerlässlich. So müssen beispielsweise Fahrzustände und Parametereinstellungen konsistent abgespeichert werden. Des Weiteren spielt die Sicherung des Know-hows eine immer größere Rolle in der Praxis. Während früher der größte Teil des Wissens in der mechanischen Konstruktion des Motors steckte, so steht diesbezüglich heutzutage die Software mit den dazugehörigen Daten im Mittelpunkt. Kennfelder, Kalibrierungsdaten und Parametersätze, die komplette Modelle für Steuerung bereitstellen, müssen zum einen vor unerlaubtem Zugriff, aber auch zum anderen vor unerlaubter Manipulation geschützt werden. Hier muss das Datenmanagement beispielsweise Methoden für eine Ver- und Entschlüsslung anbieten. Ein weiterer Bestandteil der Motorsteuerung sind On-Board-Diagnose-Systeme, die zum einen der Eigendiagnose und der Fehlerspeicherung dienen und zum anderen auch auf Grund gesetzlicher Bestimmungen zum Emissionsschutz benötigt werden. Neben klassischen Fehlermethoden aus der Statistik geht der Trend hin zu modellbasierten Verfahren und dynamischen neuronalen Netzen. Für das Datenmanagement bedeutet dies, dass neben klassischen Anforderungen wie Persistenz und Konsistenz auch Online-Aggregationen oder so-

gar Sensordatenfusionsalgorithmen sowie Ad-hoc Anfragebearbeitung in Zukunft eine größere Rolle spielen.

2.2.2 Daten des Multimedia- und Navigationssystems

Moderne Navigationssysteme

Ein weiterer, aus Datenmanagementsicht, interessanter Bereich ist das Datenmanagement im Multimedia- und Navigationssystem. Nominell ist das Datenaufkommen im Bereich der Navigations- und Multimediasysteme am stärksten gestiegen. Ein modernes Navigationssystem verarbeitet ca. 50 GB an Daten. Neben dem klassischen digitalen Kartenmaterial, Ortsinformationen und Daten zu Sehenswürdigkeiten geht der Trend zu 3D- und photorealistischen Darstellungen von Landkarten mittels Luft- oder Satellitenaufnahmen von Gebäuden und Sehenswürdigkeiten. Des Weiteren werden Informationen zur Verkehrsführung, wie etwa Geschwindigkeitsbegrenzungen, Einbahnstraßenregelungen, bedingte Befahrbarkeit, Abbiegehinweise und Restriktionen bereitgestellt. Zur Interaktion mit dem Fahrer wird neben der graphischen Benutzerschnittstelle auch eine Spracheingabe und -ausgabe angeboten. Dies führt dazu, dass in der Navigationsdatenbank sämtliche Orte in verschiedenen Sprachen als Audiosequenz hinterlegt sind. Für die Spracheingabe werden verschiedene aufgenommene Sprachen und Dialekte in Phoneme zerlegt, einzeln oder in Sequenzen abgespeichert und mit einem Index versehen, um die Suche des jeweiligen, durch die Spracheingabe getätigten Begriffs effizient zu ermöglichen. Hierzu werden spezielle, auf Audiodaten optimierte Indexe, die eine nachbarschaftsunterstützende Suche ermöglichen, benötigt [BS06], S. 425 ff..

Hauptspeicherdatenmanagement

Während die bisher genannten Daten auf den Sekundärspeicherbetrieb optimiert sind, kommt dem Datenmanagement von Hauptspeicherdaten ebenfalls eine große Rolle zu. Die während der Berechnung der Route ermittelten Daten müssen effizient im Hauptspeicher gehalten und beispielsweise mit aktuellen Verkehrsinformationen abgeglichen werden. Typischerweise werden hierfür *Hash*-Strukturen benutzt, die alle zuvor berechneten Objekte und teilweise auch die benachbarten Objekte verwalten und zusätzlich Hinweise für den Fahrer in Form von *Plain Text* -, *Pre-Information* -, *Information* -, *Activation Announcement* enthalten. Ebenfalls werden vom Datenmanagement im Navigationssystem Parameter über das Fahrverhalten und die Präferenzen verarbeitet. Die starke Anreicherung von Kartendaten durch zusätzliche Informationen führt zunehmend dazu, dass ca. zehn Prozent der Daten pro Jahr verändert und aktualisiert werden. Dieser Trend ist steigend. Abbildung 2.2 fasst die verschiedenen Daten des Navigationssystems und die sich daraus ableitenden Anforderungen an das Datenmanagement zusammen.

Daten der Navigation	Aufgabe	Typische Repräsentation	Beziehungen zu	Anforderungen an das Datenmanagement
Geometrische Daten	Visuelle Darstellung von Karteninformationen in 2D, 3D, photorealistische Darstellung und hybride Formen	Polygone, Farbwerte, Segmente (trianguliert),Kurvaturpunkte, Texturen, Bilddaten und Piktogramme	Topologischen Daten	Effiziente Suche, möglichst nachbarschaftserhaltende Speicherung, Multidimensionale Indexstruktur,
Topologische Daten	Standortbestimmung, Routenberechung, Map-Matching	Graphen (Knoten, gewichtete und gerichteet Kanten) mit Mapping zu Längen- und Breitengradinformationen	Ortsinformationen, geometrischen Daten	Effzienter Aufbau von Graphen, möglichst Speicherung
Phonetische Darstellungen	Repräsentation von verschiedenen Sprachen und Dialekten für die Spracheingabe	Audiodaten	Ortsinformationen	Effiziente Ähnlichkeitssuche von Phonemen in Audiodaten bzw. in Seqenzen von phonetischen Darstellungen
Ortsinformationen	Beschreibung der Ortsinformation, statische Verkehrsinformation (Höchstgeschwindigkeit usw.) und Points-of-Interest mit einem Mapping auf topologische Daten	Relationale Daten	Topologischen Daten, dynamischen Daten, phonetischen Darstellungen	Indexstrukturen
Parameterdaten	Speicherung der Präferenzen des Nutzers (Durchschnittsgeschwindigkeit)	Key-Value Daten	keine	Konsistente, persistente Speicherung
Dynamische Verkehrsinformationen	Speicherung und Aktualisierung von TMC/TMCpro Informationen	Längen und Breitengradinformation mit zugehörigen Ereignissen (Key-Value Daten)	Ortsinformationen	Konsistente Speicherung, ggf. Plausibiltätsprüfung, Triggereigenschaften, Transaktionseigenschaften wie Atomarität und Konsistenz
Routenrepräsentation/ Navigationsmodell	Effiziente Repräsentation der Route zur Laufzeit, angereichert um Announcement, Pre-Information Announcement, Information Announcement, Activation Announcement	Graphen mit Pointer zu statischen Daten und Key-Value Daten	Dynamischen Daten, topologischen Daten, geometrischen Daten, Ortsinformationen	Effiziente Hauptspeicherrepräsentation (meist durch Hashmaps oder T-Trees)

Abbildung 2.2: Anforderungen des Navigationssystems an Datenmanagementfunktionalität

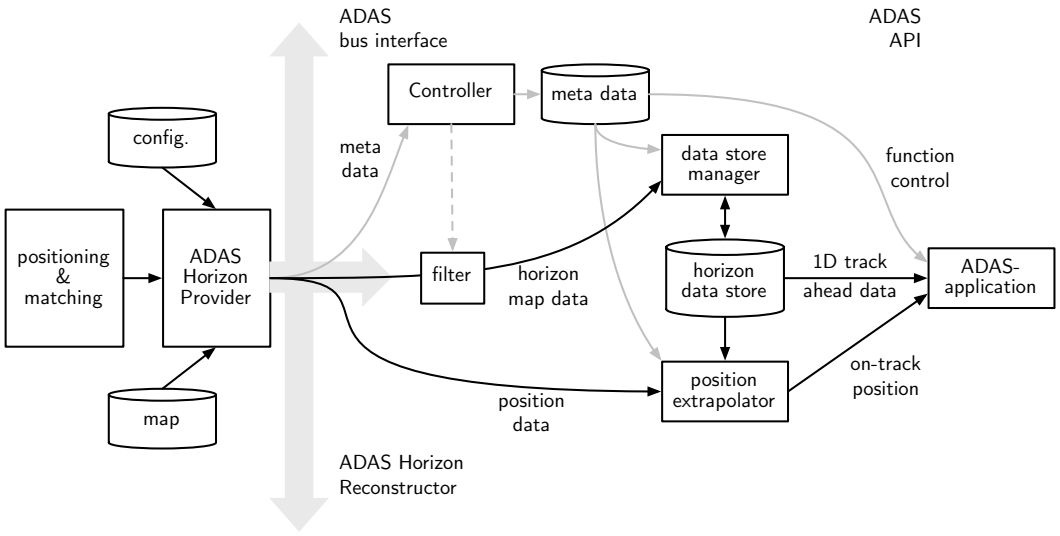

Abbildung 2.3: Datenfluss zur Umsetzung des elektronischen Fahrerhorizontes [ALM06], S. 370

Erweiterter elektronischer Fahrerhorizont

Aktuelle Forschungsprojekte im Bereich der Fahrzeugsicherheit zeigen, dass dem Datenmanagement im Navigationsbereich in Zukunft eine noch wichtigere Rolle zukommt. Zunehmend verschmelzen neue Sicherheitsfunktionen mit den Navigationsdaten. Die Abbildung 2.3 zeigt den Datenfluss eines Advanced Driver Assistance Systems (ADAS), das als elektronischer Fahrerhorizont [SMI+08] bezeichnet wird. Neben einer Vielzahl an Sensoren, die den Nahbereich des Fahrzeuges überwachen, werden auch Informationen anderer Verkehrsteilnehmer oder Verkehrsstationen empfangen, um potenzielle Gefahrensituationen zu erkennen, die noch nicht in Sichtweite des Fahrers sind. So könnten etwa Fahrzeuge, die auf Grund eines technischen Defektes hinter einer Kurve stehengeblieben sind, andere Fahrzeuge warnen. Die Herausforderung für das Datenmanagement eines solchen ADAS ist die Filterung und Aufbereitung der einlaufenden Daten für den Fahrer.

2.2.3 Datenmanagement am Beispiel eines Radladers

Studie in Zusammenarbeit mit Volvo

Im Unterschied zu der bereits präsentierten Analyse im PKW-Bereich, die im Wesentlichen auf einer Literaturrecherche beruht, wird im folgenden Abschnitt Datenmanagementfunktionalität im Nutzfahrzeugbereich an einem konkreten Fallbeispiel diskutiert. Hierzu dient eine Studie (vergleiche hierzu NYSTRÖM et al. [NTN+02]) der Universität Lingköping, die mit dem Fahrzeughersteller Volvo in Schweden durchgeführt wurde. Dabei wurden die Daten, die in einem modernen Bagger und einem Radlader anfallen, genauer untersucht. Während im PKW-Markt Großserienproduktionen mit mehreren

100.000 Fahrzeugen dominieren, werden in diesem Nutzfahrzeugbereich Kleinserien mit teilweise Stückzahlen unter 100 Fahrzeugen gebaut. Einige Fahrzeuge werden auch komplett kundenindividuell gefertigt.

Typischer Aufbau der Steuerungen

Typischerweise besteht ein solches Fahrzeug aus bis zu fünf Hauptkomponenten - der zentralen Steuerung für die Display-Einheit, der Kabinensteuerung, der Steuerung des Getriebes, der Steuereinheit für Fahr- und Arbeitseinrichtungen und der Motorsteuerung. Genauer werden im Folgenden die Steuereinheit für die Fahr- und Arbeitseinrichtung[3] *Vehicle Electronic Control Unit* (VECU) und die Displayeinheit *Instrumental Electronic Control Unit* (IECU) betrachtet. Beide Systeme dienen als zentrale Steuereinheiten und sind mit Subsystemen ausgerüstet.

Eingesetzte Hardware

Bei der Hardware steht der VECU und der IECU ein 16- beziehungsweise 32-Bit Kontroller mit 64 KB RAM, 512 KB Flash und 32 KB EEPROM zur Verfügung. Für komplexere Spezialanfertigungen wird der Speicher auf ein bis zwei MB erweitert. Je nach Funktionsumfang des Fahrzeuges wird der Prozessor mit 16 oder 20 MHz betrieben. Als Betriebssystem wird das in Schweden entwickelte Echtzeitbetriebssystem *RUBUS OS* [HMTN+08] verwendet. Die Kommunikation der Systeme untereinander ist durch einen CAN-BUS beziehungsweise durch eine langsame Diagnoseverbindung sichergestellt. Der Funktionsumfang und die zu speichernden Daten unterscheiden sich wie folgt:

VECU: Die Hauptaufgabe der Fahrzeug- und Arbeitseinrichtungssteuerung ist die Steuerung und die Kontrolle der unterschiedlichen Zustände des Fahrzeuges im Fahr- beziehungsweise im Arbeitsmodus und das Identifizieren von Anomalien oder Störungen, wie zum Beispiel erhöhte Temperaturen oder Druckverluste. Um größeren Schaden vom Fahrzeug abzuwenden, kann die VECU das komplette System zu einem kontrollierten Nothalt herunterfahren. Hierzu werden vor allem unterschiedliche Sensoren und Aktoren ausgewertet beziehungsweise gesteuert. Weiterhin erfolgt eine fortlaufende Protokollierung zu Wartungszwecken und zur Prüfung eventueller Garantieansprüche. Abbildung 2.4 zeigt den Datenfluss in der VECU. Die Daten, die in diesem Bereich auftreten, können in folgende fünf Kategorien unterteilt werden:

Sensor/Aktor-Rohdaten: Als Rohdaten werden in diesem Zusammenhang alle Daten bezeichnet, die von einem Sensor gelesen oder auf einen Aktor

3 Als Arbeitseinrichtungen werden in diesem Zusammenhang zum Beispiel Schaufeln oder der Arbeitsarm des Baggers verstanden.

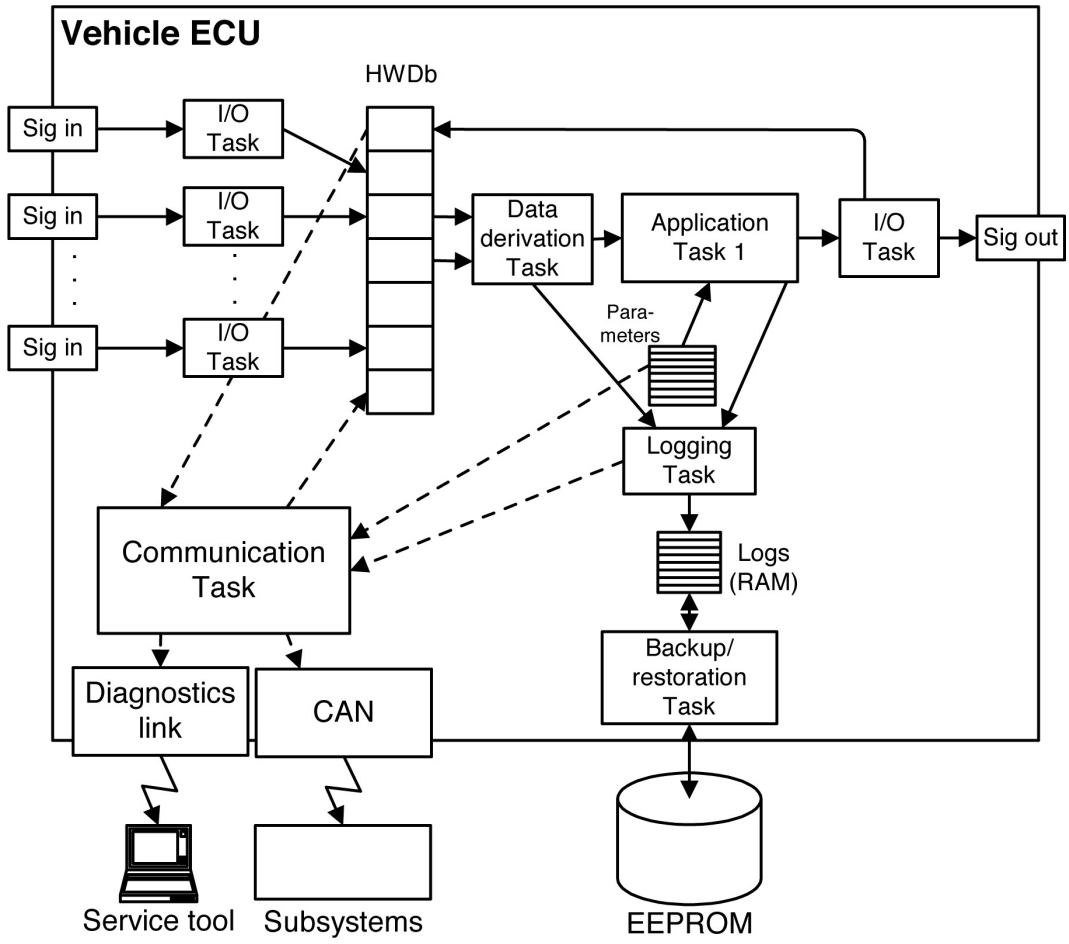

Abbildung 2.4: Auszug des Datenflusses in der VECU eines Radladers [NTN+02], S. 252

geschrieben werden. Eine sehr wichtige Eigenschaft beim Lesen beziehunsweise Schreiben dieser Daten ist die zeitliche und inhaltliche Konsistenz der Daten.

Sensor/Aktor-Parameter: Die Sensor/Aktor-Parameterdaten werden beispielsweise zum Initialisieren beim Neustart, als Referenzdaten oder zur Umrechnung der Sensor/Aktordaten benötigt. Dabei werden Teile der Daten im EEPROM gesichert. Primär werden diese Daten von unterschiedlichen Prozessen gelesen. Treten Änderungen auf, müssen diese Änderungen atomar und konsistent durchgeführt werden.

Sensor/Aktor-Prozessdaten: Als Prozessdaten werden die durch Applikationen verarbeiteten Daten bezeichnet. Der größte Teil dieser Daten wird kurzzeitig zwischengespeichert, um ihn in folgenden Prozessen als Input zu benutzen.

Protokolldaten: Einige Daten der Applikationen werden zu statistischen Zwecken oder für Wartungsarbeiten benötigt. Diese Daten werden auf einen stabilen Speicher geschrieben. Viele dieser Daten werden kumulativ von unterschiedlichen Prozessen fortgeschrieben.

Applikationsparameter: Diese Gruppe umfasst alle für die Applikation notwendigen Parameter. Ein kleiner Teil dieser Daten ist statisch und der andere Teil unterliegt ständigen Änderungen.

Viele der Daten haben nur eine beschränkte Zeit Gültigkeit und werden periodisch geändert. Aufgabe der Transaktionsverwaltung ist es, die geforderten Echtzeiteigenschaften konsistent sicherzustellen.

IECU: Die zentrale Aufgabe der Displayeinheit ist nicht nur die Darstellung von Ereignissen, wie Warnungen und Instrumentenanzeigen, sondern auch die Steuerung zahlreicher Funktionen im Umfeld des Fahrers. Einige Funktionen haben eine starke Ähnlichkeit zu den Funktionen der VECU, sodass im Folgenden nur disjunkte Funktionen vorgestellt werden. Für die Darstellung in der Displayeinheit werden viele unterschiedliche Bilder und Textinformationen benötigt. Die Text-Datenbank einer solchen Displayeinheit umfasst 30 verschiedene Sprachen. Alle Menüeinträge, Anweisungen, Fehlermeldungen und Bedienungsanleitungen sind in allen Sprachen hinterlegt. Für einen effizienten Zugriff werden hier Indexstrukturen verwendet. Die Abbildung 2.5 zeigt den Datenfluss in einer IECU.

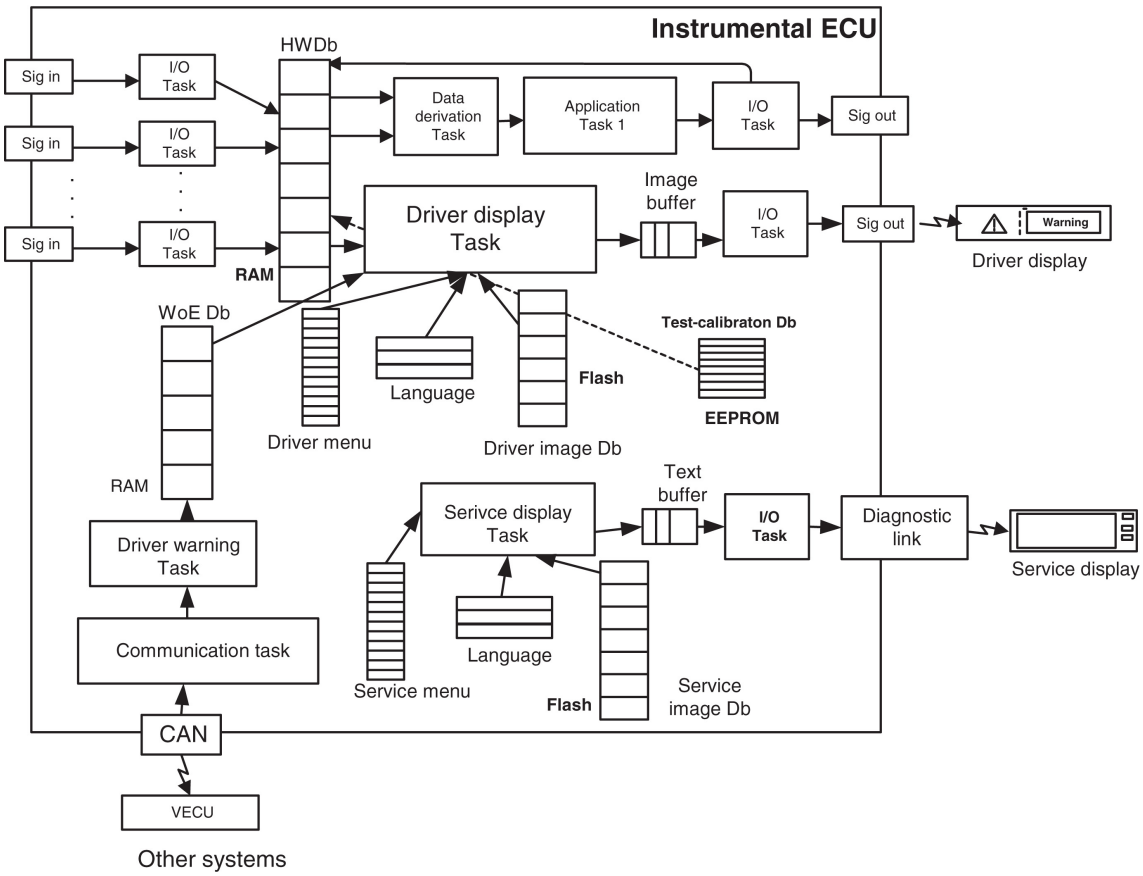

Abbildung 2.5: Auszug des Datenflusses in der IECU eines Radladers [NTN+02], S. 252

Ergebnis der Analyse

Wie die Abbildungen 2.4 und 2.5 zeigen gibt es keine zentrale Instanz, die die Daten der Subsysteme verwaltet. Die Abhängigkeiten der Daten untereinander sind in den letzten Jahren immer stärker angewachsen und führten zu extremen Problemen in der Entwicklung. Als Ergebnis der Analyse wurde festgestellt, dass die bisherige Form der Speicherung und Verwaltung der Daten nicht mehr handhabbar ist. Durch die nicht realisierte Trennung von Daten und Funktionalität wird nicht nur die Phase der Spezifikation und Entwicklung verteuert, sondern auch die Testung und Verifikation, die auf Grund der Anwendung dieser Systeme in sicherheitskritischen Bereichen notwendig sind. Deshalb sollen nach Aussage der Studie alle bisher dezentral gespeicherten Daten zentral pro Subsystem integriert werden, um so eine, wenn auch nur eingeschränkte, logische und physische Datenunabhängigkeit zu gewährleisten und eine vereinheitlichte Echtzeittransaktionsverwaltung zu ermöglichen. Zusammenfassend lässt sich festhalten, dass der Bedarf von stark spezialisierter Datenmanagementfunktionalität in diesem Umfeld steigt, da diesbezüglich keine Lösungen vorhanden sind.

2.3 Drahtlose Sensornetzwerke

Wurzeln drahtloser Sensornetzwerke

Ein weiteres klassisches Einsatzszenario von eingebetteten Systemen sind drahtlose Sensornetzwerke [AWSC02, YMG08]. Die Geschichte dieser Systeme geht auf die fünfziger Jahre des letzten Jahrhunderts zurück, als die Amerikaner mit dem *Sound Surveillance System* begannen, mittels im Ozean verteilter Hydrophone, Bewegungen russischer U-Boote zu dokumentieren. Die damals verwendeten Sensoren waren noch durch ein Kabel mit den Basisstationen am Festland verbunden. Durch die Weiterentwicklung der drahtlosen Kommunikation und der explosionsartigen Verbreitung von kleinsten eingebetteten Systemen werden heute in der Forschung hauptsächlich drahtlose, teilweise autonom agierende Sensornetzwerke untersucht[4]. Die verteilten Sensorknoten kommunizieren in der Regel drahtlos mit einem zentralen Gateway, der eine Verbindung zu drahtgebundenen Geräten bietet (vgl. hierzu Abbildung 2.6). Dort werden die Messdaten erfasst, verarbeitet, analysiert und dargestellt. Um Entfernung und Zuverlässigkeit in einem Wireless-Sensornetzwerk zu erhöhen, kann man Router einsetzen. So entsteht eine zusätzliche Kommunikationsverbindung zwischen Endknoten und dem Gateway.

4 Da zunehmend auch Aktoren in Sensornetzwerken beteiligt sind, wird häufig auch von *Smart Cooperating Objects* gesprochen.

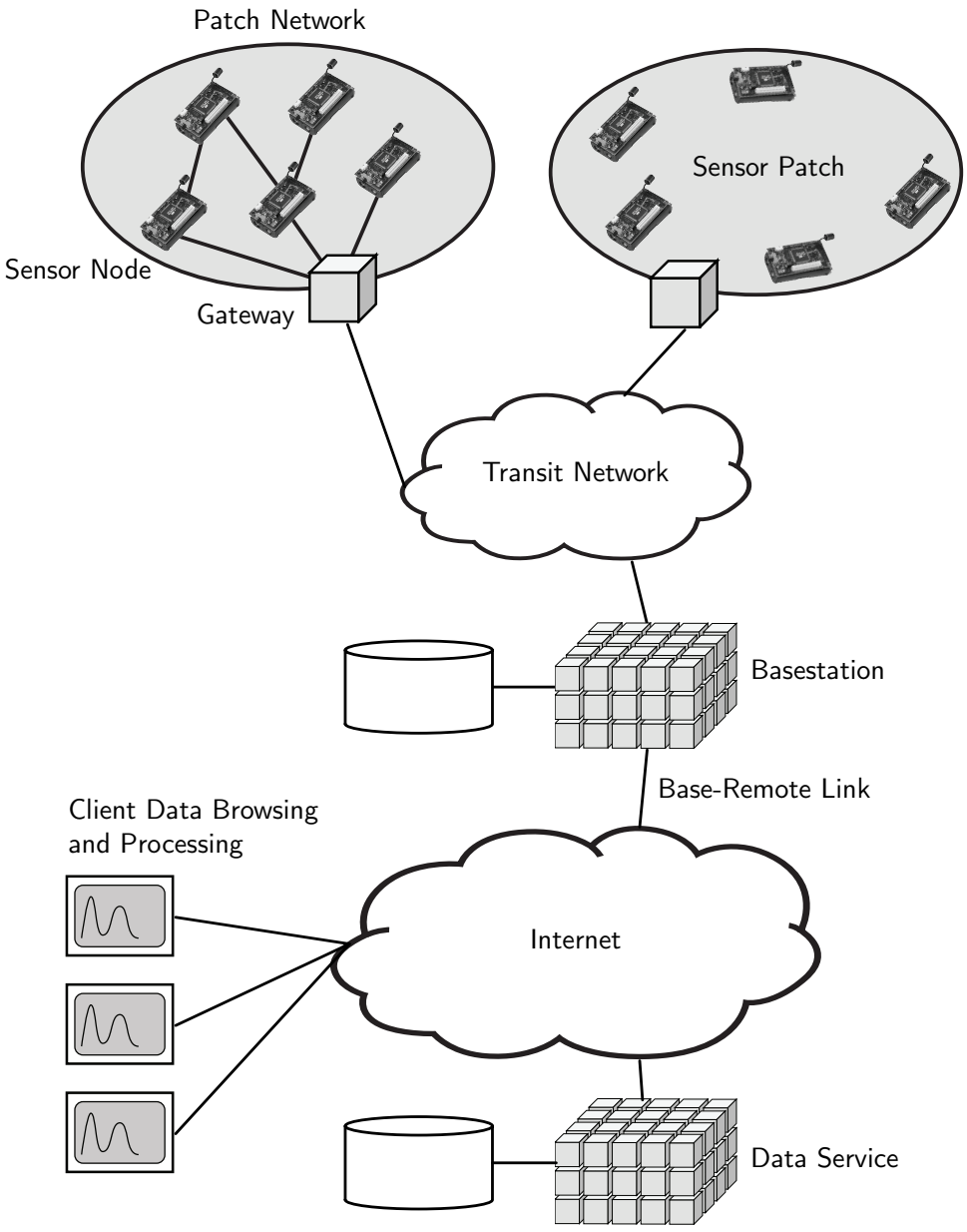

Abbildung 2.6: Typische Architektur eines Sensornetzwerkes [MCP+02], S. 90 (verändert)

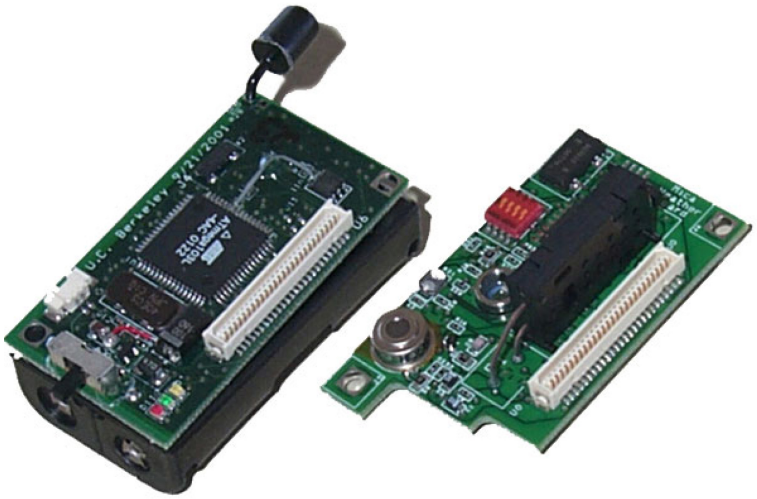

Abbildung 2.7: Beispiel eines Sensorknotens (Mica, UC Berkeley) [MCP+02], S. 91

Im Bereich der Netzwerkkommunikation [EGHK99] wird insbesondere auf den Gebieten der ressourceneffizienten Kommunikation, der Reichweitenoptimierung und der Unempfindlichkeit gegenüber Verbindungsabbrüchen sowie der spontanen Vernetzung geforscht. Die Forschung im Hardwarebereich konzentriert sich im Wesentlichen auf die Verkleinerung der Hardware, den effizienten Ressourcenkonsum und die Entwicklung neuer leistungsstärkerer Sensoren. Typische Hardwaresysteme im Bereich der drahtlosen Sensornetzwerksysteme sind: *BTnode*[5], *FireFly*[6] *Imote2*[7], *Mica2, Mica2Dot* und *MicaZ*[8], *Particles*[9], *SquidBee*[10], *Sun SPOT*[11] und *TinyNode*[12].

Anwendungsszenarien

Typische Sensoren messen in Anwendungsszenarien beispielsweise Temperatur, Druck, Vibrationen, Feuchtigkeit, Schall, Beschleunigungen, Konzentrationen verschiedener Gase und weitere Parameter. Abbildung 2.7 zeigt einen solchen Sensorknoten, der mit verschiedenen Sensoren ausgestattet werden kann.

Organic Computing

Sich selbst organisierende drahtlose Sensornetzwerke werden häufig auch unter dem Begriff des *Organic Computing* [MSvdMW04, Dre08, Dre07b, Dre07a] subsummiert. Die konsequente Weiterentwicklung bezüglich der Miniaturisierung und Autonomie der

5 http://www.btnode.ethz.ch/Documentation/BTnodeRev3HardwareReference
6 http://www.ece.cmu.edu/firefly/index.html
7 http://www.xbow.com/Products/productdetails.aspx?sid=253
8 http://www.xbow.com/Products/wproductsoverview.aspx
9 http://particle.teco.edu/
10 http://www.libelium.com/squidbee/index.php?index.php
11 http://sunspotworld.com/products/
12 http://www.tinynode.com/

Sensoren wird in der aktuellen Forschung unter *Smart Dust* (dt. intelligenter Staub) zusammengeführt [KKP99, WLLP01] und deutet an, dass extrem ressourcenbeschränkte Systeme auch in Zukunft ihre Bedeutung nicht verlieren.

Anwendungsgebiete

Die Anwendungsgebiete von drahtlosen Sensornetzwerken sind vielseitig. Ein Großteil der Einsatzgebiete hat auf Grund der historischen Entwicklung und der Anstrengungen in der Rüstungsforschung einen militärischen Hintergrund, wie etwa die Kampffeldüberwachung oder Aufklärung. Dennoch rückten in den letzten Jahrzehnten mehr und mehr zivile Einsatzszenarien in den Fokus der Forschung. Im Folgenden werden drei repräsentative Einsatzgebiete vorgestellt:

Bauwerksüberwachung: Der Bauwerksüberwachung mittels Sensoren kommt insbesondere im Zusammenhang mit einem zunehmenden Alter von Bauwerken und steigenden Anforderungen an Tragfähigkeit sowie Dauerhaftigkeit eine größere Bedeutung zu. Derzeitig wird an exponierten Bauwerken wie zum Beispiel Brücken und Hochhäusern eine Vielzahl von charakteristischen Kennwerten mit Sensorik erfasst und ausgewertet. Zu den typischen Kennwerten gehören Luft- und Bauteiltemperatur, Bauteilfeuchte oder -dehnungen sowie Bauteilverschiebungen und -schwingungen. Diese zerstörungsfreien Prüfmethoden ermitteln Kennwerte aus dem Inneren eines Bauwerkes und lassen so genauere Aussagen über die Veränderungen des Tragwerks und die Restlebensdauer zu. [LWS+08]

Naturschutz und biologische Studien: Der Einsatz eines drahtlosen Sensornetzwerkes zur Verhaltensanalyse von wild-lebenden Tieren und die Überwachung ihrer Lebensräume stellt ein weiteres Einsatzgebiet dar. Hierzu werden je nach Szenario, Tiere und/oder ihre Umgebung mit Sensorknoten bestückt. Dadurch kann die Position und Geschwindigkeit der Tiere ermittelt werden. Im Zentrum des Interesses steht das Verhalten eines Individuums und/oder einer Gruppe von Tieren im Kontext zu den erfassten Umweltbedingungen wie zum Beispiel Sonne, Regen, Temperatur, Standort und Zeit. In regelmäßigen Abständen werden dann mobile Basisstationen beispielsweise per Fahr- oder Flugzeug, durch diese überwachten Gebiete bewegt, welche die Daten von allen Knoten einsammeln. Aus den gewonnenen Daten können Biologen das Tierverhalten erforschen und gegebenenfalls auch Störungen erkennen. Neben dem Erkenntnisgewinn für Biologen können so Hinweise für den Naturschutz abgeleitet werden. Beispielprojekte in diesem Umfeld sind *Great Duck Island* [WMG04, GGP+03], in dem das Brutverhalten von Seevögeln überwacht wird und *ZebraNet*, welches das Zug- und Herdenverhalten von Wildtieren in einem afrikanischen Nationalpark überwacht [JOW+02, Mar06].

Ozeanklimaüberwachung: Ein großes Sensornetzwerk ist das *Global Ocean Sensor Network*. Es besteht aus mehr als 3.250 Bojen[13], die in allen Ozeanen und den angrenzenden Meeren mit einem Abstand von ca. 300 Kilometer zueinander verteilt sind. Die Bojen sind dabei nicht stationär gebunden, sondern driften mit den jeweiligen Meeresströmungen. Die Sensoren der Bojen messen die Temperatur, den Salzgehalt und Wellengang. Dabei tauchen sie in mehr als 2.000 Meter Tiefe für eine zehntägige Periode und sammeln so Daten in unterschiedlichen Wasserschichten. Nach zehn Tagen werden die Daten an der Meeresoberfläche über Satellitenverbindungen übermittelt. Die Klimadaten der Bojen werden sowohl für die Simulation der Meeresströmung als auch zur Klimaberechnung benötigt.

Die drei exemplarischen Anwendungsgebiete zeigen, dass typische Funktionalitäten wie beispielsweise Datenvorverarbeitung, verschiedene Aggregationen, *Caching*-Mechanismen, verteilte Anfrageverarbeitung und Konsistenzchecks aus dem Bereich des Datenmanagements benötigt werden.

Datenmanagement in drahtlosen Sensornetzwerken

Ein Großteil der Datenbankforschung der letzten Jahre beschäftigte sich mit der Anfrageverarbeitung und Anfrageoptimierung in Sensornetzwerken [MFHH03], insbesondere unter Berücksichtigung von Ressourcenknappheit [MH02], Online Aggregation in Ad-hoc Anfragen auf Datenströmen [MFHH02] und die Anfrageverarbeitung von Datenströmen [HRR98, GÖ03] selbst. Bekannte Systeme in diesem Bereich sind zum Beispiel *TinyDB* [MFHH05b], *COMET DBMS* [NTN+04], *Directed Diffusion* [IGE00] und *Cougar* [YG02]. Das Einsatzgebiet dieser Systeme ist jedoch sehr begrenzt, da keine ausreichende Erweiterbarkeit beziehungsweise Konfigurationsmöglichkeiten existieren, um die Anforderungen der stark heterogenen Systeme mit variierender Hardware und gewünschter Funktionalität zu unterstützen.

2.4 Konfigurierbares Datenmanagement - *State of the Art*

Forschungsarbeiten im Bereich der konfigurierbaren, erweiterbaren und flexiblen Datenmanagementsoftware wurden aus unterschiedlichen Problemstellungen heraus schon in den achtziger Jahren durchgeführt. Einer der Hauptgründe war, dass auf Grund neuer Anwendungsgebiete wie beispielsweise CAD-Anwendungen, Geo-Informationssysteme, Multimediaanwendungen und neuer Datenbankkonzepte, wie OODBMS, die bisherigen monolitischen Strukturen nicht angepasst werden konnten. Insbesondere die Forschung brauchte während dieser Zeit Systeme, die sich schnell erweitern ließen.

13 Stand 2009

Gedankenmodell 5-Schichtenarchitektur

Ansätze, wie man ein DBMS in Teilfunktionalitäten splitten kann, werden schon seit Jahren in der Lehre als Referenzmodell mit der 5-Ebenen Architektur nach SENKO et al. [SAAF73] bzw. HÄRDER [HR85] gelehrt. Die Basis dieses Modells bilden Arbeiten an IBM's *SystemR* von BLASGEN et al. [BAC+81], das auch gern als die „Mutter" der heutigen Relationalen Datenbanksysteme bezeichnet wird.

2.4.1 Klassifikation erweiterbarer DBMS

General-Purpose-Systeme

Konfigurierbare DBMS erlauben eine Anpassung an die speziellen Einsatzbedingungen der Anwendung. Dies ist immer dann notwendig, wenn die Standardfunktionalität von *General-Purpose*-DBMS für den Anwendungsfall nicht ausreichend ist oder wie im Fall von eingebetteten Systemen mit Ressourcenbeschränkungen der Einsatz von *General-Purpose*-Systemen zu viel *Overhead*-Funktionalität mit sich bringt. Das Hauptproblem bei den derzeitigen verfügbaren *General-Purpose*-DBMS ist die monolitische Architektur der Systeme. Durch die direkte Verschachtelung der Funktionalität zu einem Monolit sind Erweiterungen und Modifikationen dieser Funktionalität nur schwer möglich. Die Gründe hierfür sind vielfältig. Die Basisstrukturen der heutigen kommerziell verfügbaren Systeme sind schon vor teilweise mehr als 20 Jahren entstanden. Die damals vorhandenen programmiertechnischen Möglichkeiten entsprachen nicht den heutigen Erkenntnissen der Softwaretechnik.
Aber auch heutzutage wird oft bei Erweiterungen und sogar bei Neuimplementierungen auf viele vorhandene Modularisierungstechniken (wie zum Beispiel OOP) nicht zurückgegriffen. Der Grund hierfür liegt nicht darin, dass die Programmierer im Umfeld der Datenbankimplementierung Trends in der Softwaretechnik ignorieren. Vielmehr geht es um den besonderen Anspruch an die Performanz. Nur durch diese monolitische Implementierung konnten bisher die Ansprüche an Laufzeitperformanz und Durchsatz der DBMS Rechnung getragen werden. Trotz dieser nur auf Laufzeitperformanz und Durchsatz optimierten Sichtweise werden bereits jetzt Anstrengungen zur Beseitigung der monolitischen Strukturen unternommen. Die Gründe liegen in den durch die monolitischen Strukturen verursachten Problemen bezüglich der Wartbarkeit, Erweiterbarkeit und Fehleranfälligkeit, die erhebliche Kosten nach sich ziehen.

Erweiterbare DBMS

Einen Ansatz für eine Klassifikation von erweiterbaren und konfigurierbaren DBMS liefern DITTRICH UND GEPPERT in [DG00]. Eine Disjunktion der Gruppen ist leider nicht realisiert und zu großen Teilen der historischen Entwicklung der Vertreter dieser Gruppen geschuldet. Viele der hier vorgestellten Systeme sind in der Zeit der Entwicklung von objektorientierten Datenbanksystemen entstanden. Während dieser Zeit hatte

die Forschung ein besonders großes Interesse an einfach erweiterbaren Systemen, um schneller prototypisch neue Theorien im Bereich der Anfrageoptimierung, der abstrakten Datentypen oder der Index-Implementierungen zu evaluieren. Die folgende Klassifikation erweitert die Klassen von DIETRICH und GEPPERT [DG00], S. 12 ff.:

Kernsysteme: Kernsysteme stellen eine definierte Grundfunktionalität von Datenbanksystemen bereit, die von allen Nutzergruppen des DBMS verwendet werden. Bezüglich der Basisfunktionalität ist das DBMS nicht variabel. Das reine Kern-System stellt an sich noch kein funktionstüchtiges DBMS dar. Durch die Implementierung von Funktionalität, die auf den Anwendungsfall zugeschnitten ist, wird das Kernsystem zu einem lauffähigen System erweitert. Die zusätzliche Funktionalität wird durch den *Datenbankimplementierer* (DBI) realisiert. Hierfür wird durch das Kernsystem eine Schnittstelle bereitgestellt. Die Schwierigkeit beim Bau eines Kernsystems liegt vor allem im Abwägungsprozess, welche Funktionalität im Kern als Basis bereitgestellt werden sollte. Wird zu viel Funktionalität in den Kern verlagert, muss der DBI zwar weniger Funktionalität selbst implementieren, aber bezüglich der Kernfunktionalitäten ist der Entwickler nicht mehr variabel. Im Umkehrschluss muss der DBI bei sehr wenig bereitgestellter Funktionalität viel selbst implementieren, das ihm sehr viel Freiheit einräumt, aber recht aufwändig werden kann. Ein klassischer Vertreter dieser Art der erweiterbaren DBMS ist das *Wisconsin Storage System (WISS)* [CDKK85]. Es stellt einfache Funktionen eines Speichermanagers bereit und dient als Grundlage für das kommerzielle ODBMS *O2*.

Ein weiterer Vertreter dieser Klasse ist das *Darmstadt Database System (DASDBS)* [PSS87, SPSW90]. Als Kernfunktionalität ist ebenfalls ein Speichermanager implementiert. Die Besonderheit des Speichermanagers ist, dass er mit geschachtelten Relationen umgehen kann. In einem späteren Schritt wurde der Kern um Funktionalität für die Transaktionsverwaltung erweitertet. Diese Funktionalität wurde in einem anschließenden Projekt um Multi-Level-Transaktionen [Wei91] erweitert. Weiterhin werden dieser Gruppe die Objektspeichermanager *EOS* [Bil92] und dessen Nachfolgersystem *Bess* [BP95, BP96] (inklusive einer Transaktionsverwaltung), *Kiosk* [ND96] (Speichermanager des *KIDS*-Ansatzes [GSD97]), *ObServer* [HZ87, SZR91], *Texas* [SKW92] und *KOMS* [YSL91] zugeordnet.

Echte erweiterbare Systeme: Echte erweiterbare Systeme (engl. *pure extensible database systems*) sind im Gegensatz zu Kernsystemen lauffähige DBMS. Die Anzahl der Variationspunkte ist meist stark beschränkt. Des Weiteren sind diese Varia-

tionspunkte fest in die Architektur eingearbeitet. Die häufigsten Erweiterungsmöglichkeiten beziehen sich auf das Hinzufügen neuer abstrakter Datentypen und spezieller Operationen auf diesen Datentypen oder Indexstrukturen. Vertreter dieser Klassen sind die Datenbankmanagementsysteme *Ingres/ Postgree* [LS88], die als erste Systeme das Hinzufügen von abstrakten Datentypen und das Erweitern von Operationen sowie Indexstrukturen erlauben. Heutzutage ist diese Art der Erweiterungsmöglichkeit in vielen kommerziell verfügbaren Systemen Standard, wie beispielsweise in *IBM DB2* [DG00].

Anpassungsfähige Systeme: Anpassungsfähige Systeme sind ebenfalls komplette funktionstüchtige DBMS, die bei Bedarf an den Anwendungskontext angepasst werden. Im Gegensatz zu den echt erweiterbaren Systemen sind hier nur sehr eingeschränkte Anpassungen an den Anwendungskontext möglich. Die häufigsten Anwendungsgebiete lassen sich im Bereich der Anfrageverarbeitung oder des Speichermanagements finden. In der Anfrageverarbeitung ist es beispielsweise dem Datenbankadministrator oder Anwendungsentwickler erlaubt, zusätzliche Transformationsregeln für die Anfrageoptimierung hinzuzufügen und somit semantische Aspekte der Anwendung bei der Optimierung von Anfragen zu berücksichtigen. Der bekannteste Vertreter dieser Klasse ist das System *Starburst* [MP87, HFLP89, HS90]. *Starburst* erlaubt beispielsweise das Einfügen von neuen Operatoren auf Relationen oder das Hinzufügen von neuen Regeln zur Manipulation des internen Anfragegraphen für die Optimierung. Des Weiteren ist es möglich, das Speichersystem von Starburst anzupassen.

Toolkit-Systeme: *Toolkit*-Systeme stellen dem Datenbankimplementierer in Form einer Bibliothek alternative Implementierungen für Funktionen einer Datenbank bereit. Die Hersteller von DBMS können ihren Kunden somit alternative Lösungen in einem Baukasten zur Verfügung stellen, mit deren Hilfe eine Erweiterung oder Anpassung einfacher zu realisieren ist. Häufig wird gerade der Toolkit-Ansatz mit anderen Realisierungsmöglichkeiten gekoppelt. Der Speichermanager von *EXODUS* [CDRS86, GD87, CD87, RCDS87] ist mit Hilfe eines solchen *Toolkits* sowohl konfigurierbar als auch erweiterbar. Der DBI kann aus einer Bibliothek verschiedene Zugriffsmethoden auswählen und in der gewünschten Kombination betreiben. Werden zusätzliche Funktionen oder Erweiterungen dieser Zugriffsstrukturen benötigt, kann der DBI mit Hilfe der Datenbankprogrammiersprache E[14] [RCS93] diese Funktionalität implementieren. Besonderer Wert bei der Realisierung von

14 Eine Erweiterung der Programmiersprache C++.

EXODUS wurde auf die Reduzierung von Verknüpfungen zwischen den einzelnen Modulen gelegt. Dadurch sollte die Komplexität der Systeme reduziert werden, um eine Erweiterung einfacher zu realisieren. Ein weiteres System dieser Klasse ist *Berkeley DB*[15].

Transformationssysteme: Die Gruppe der Transformationssysteme im Bereich der erweiterbaren DBMS kann teilweise auch zu den Generator-Ansätzen abgegrenzt werden. Ein Transformationssystem wandelt ein *Input*-Programm in ein *Output*-Programm. Die Transformation wird dabei entweder in deklarativen Sprachen oder mit Hilfe von "herkömmlichen"[16] beziehungsweise erweiterten[17] Programmiersprachen beschrieben. Ein Transformationssystem kann somit verwendet werden, um ein Generator-System zu realisieren. Das *GENESIS*-System [Bat86, BBG+88] gilt im Bereich der Transformationssysteme als Referenzsystem. Auf Grund der hierarchisch organisierten Transformationsschritte wird häufig als Klassifikationsmerkmal für Transformationssysteme eine hierarchische Anordnung der Transformationsschritte in Form einer Schichtenarchitektur gefordert. Im Ergebnis von *GENESIS* wurde das *GenVoca*-Modell entwickelt. Weitere Ansätze in dem Bereich sind *Predator* [BSST93] und *P2* [BT97].

Generator-Systeme: Generatoren ermöglichen aus einer gegebenen Spezifikation von Datenbankfunktionalitäten die Generierung von implementierten Komponenten eines DBMS. Hierzu definiert der DBI ein, in der Struktur vorgegebenes Modell und parametrisiert dieses entsprechend des Anwendungskontextes. Ein Generator generiert im Anschluss die Teilkomponenten des DBMS. Eine vollständige Generierung vom kompletten DBMS ist auf Grund der Komplexität der DBMS derzeitig nicht realisierbar. Häufig werden deshalb Generatoransätze mit dem Toolkit-System gekoppelt. Hauptanwendungsgebiet solcher Generatoren im Bereich DBMS sind Anfrageoptimierer. Einer der bekanntesten Vertreter dieser Klasse ist der Anfrageoptimierer des EXODUS-Systems. Der Prototyp des Generators diente dazu, verschiedene Anfrageoptimierer automatisch zu generieren, um neue Methoden und Operatoren in einem Anfragoptimierer zu testen. Nachfolger des *EXODUS-Ansatzes* waren *Volcano* von GRAEFE UND MCKENNA [GM93] [MFHH05a] und *Open OODB* [FBB93].

15 *Berkeley DB* wird im Abschnitt 2.5.1 genauer beschrieben.
16 wie beispielsweise Objektorientierte Programmierung und Strukturierte Programmierung
17 wie beispielsweise Aspektorientierte Programmierung, Merkmalsorientierte Programmierung, Subjektorientierte Programmierung usw.

Frameworks: Eine konsequente Anpassung der Ansätze der Kernsysteme und der *Toolkit*-Systeme an die Objektorientierte Programmierung ermöglichen *Frameworks*. Sie bieten einen festen Kern an Funktionalität und Variationspunkte zur Erweiterung in Form von unterschiedlich realisierten Klassen an. Zum einen kann der DBI entscheiden, welche der verschiedenen Implementierungen, die in unterschiedlichen Klassen vorliegen, er für seinen Anwendungsfall benötigt. Zum anderen kann er abstrakt gehaltene Implementierungen für seinen Anwendungsfall anpassen. Klassische Einsatzgebiete sind Index-Strukturen und Anfrageoptimierer. HELLERSTEIN et al. [HNP95] präsentieren *Generalized Search Tree (GiST)*, die Basisfunktionalitäten für Suchbäume bereitstellen. Der Entwickler kann durch Erweiterungen neue Indexstrukturen hinzufügen. *OPT++* [KD99] stellt ein Framework für einen Anfrageoptimierer bereit. Die Klassen des Frameworks stellen Funktionalität für den Aufbau, die Manipulation und die Transformation von Operatorbäumen in Ausführungsplänen zur Verfügung. Ähnliche Ansätze schlagen auch GRAEFE [Gra95] mit *Cascades*, *EROC* [MBHT96] und VAN BERCKEN ET AL. [BBD+01] mit der Anfrageoptimierbibliothek *XXL* vor. Ein objektorientiertes *Framework*, das ein komplettes DBMS bereitstellt, ist *Berkeley DB* in der Java-Variante.

Komponentenbasierte DBMS: Mit dem Aufkommen der komponentenbasierten Softwareentwicklung wurden beispielsweise von GEPPERT ET AL. [GSD97] mit *KIDS*, CHAUDURI und WEIKUM [CW00] mit ihrem Vorschlag einer RISC-Style-Architektur und NYSTRÖM et al. [NTN+04] mit *COMET DBMS* Komponententechniken zum Bau von erweiterbaren DBMS vorgeschlagen. Dabei werden alle wesentlichen Funktionalitäten eines DBMS in Komponenten implementiert. Alle Komponenten kommunizieren über definierte Schnittstellen. Werden in verschiedenen Anwendungsszenarien unterschiedliche Implementierungen benötigt, können diese Komponenten einfach ausgetauscht werden. Die Granularität der vorgeschlagenen Komponenten ist sehr unterschiedlich. Während *KIDS* aus recht groß gefassten Komponenten besteht, schlagen CHAUDURI und WEIKUM [CW00] sehr kleine Komponenten vor, um so möglichst feingranulare Variabilität anzubieten. Eine zu feingranulare Zerlegung führt aber auf der anderen Seite zu einem extremen Kommunikations-*Overhead* und damit zu Performanz-Einbußen, sodass eine Kosten-Nutzen-Analyse vorgenommen werden muss.

Bis auf wenige Forschungsansätze (*GENESIS*, *Predator* und *P2*) sind die hier präsentierten Forschungssysteme funktional getrieben, das heißt sie greifen zu wenig Aspekte des

Software-Engineering auf. Im folgenden Abschnitt werden die daraus entstehenden Probleme diskutiert.

2.4.2 Diskussion

Umsetzung in kommerziellen Systemen

Bezogen auf die Variabilität, Konfigurier- und Erweiterbarkeit konnten sich die meisten Ansätze bisher in der Praxis nicht oder nur in einem sehr begrenzten Maße durchsetzen. Im Gegensatz dazu haben viele konkrete Datenbankfunktionen, die in erweiterbaren Systemen erprobt wurden, den Einzug in die Praxis erhalten. Abstrakte Datentypen, Objekt-(relationale)-Erweiterungen und die Einführung von Indexen sind heute in vielen kommerziellen Systemen möglich. Hingegen sind Eingriffe in die Anfrageoptimierung oder die Transaktionsverwaltung nur sehr eingeschränkt vorhanden.

Eingeschränkte Nutzbarkeit in eingebetteten Systemen

Viele der Erweiterungsmöglichkeiten, die derzeitig im Einsatz sind und ein Großteil der in den achtziger und neunziger Jahren entwickelten Systeme, betreffen die konzeptionelle oder die externe Ebene, wie beispielsweise die Erweiterung um abstrakte Datentypen oder Anpassungen der Anfrageoptimierung auf konzeptioneller Ebene. Für das Anwendungsgebiet der eingebetteten Systeme sind jedoch die Variationsmöglichkeiten auf interner Ebene entscheidender. Der unveränderbare Kern dieser Systeme sollte so klein wie möglich sein.

Unterstützung heterogener Hardware

Die vorgestellten Systeme sind im Vergleich zum Einsatzgebiet der eingebetteten Systeme auf homogener Hardware entwickelt worden. Die hohe Adaptivität an die Hardware ist eine Grundvoraussetzung für den Einsatz im Bereich der eingebetteten Systeme.

Eingesetzte Softwaremethoden sind veraltet

Viele der in den achtziger und neunziger Jahren verwendeten Modularisierungsansätze und Programmiertechniken sind heute veraltet. Fortschritte im Bereich der Modularisierung und der Programmierung ermöglichen heutzutage eine feinere Modularisierung.

2.5 Datenmanagement für eingebettete Systeme

Datenmanagement für eingebettete Systeme

Losgelöst von den Entwicklungen der erweiterbaren Datenbankmanagementsysteme hat sich ein breites Umfeld stark anwendungsorientierten Datenmanagements für eingebettete Systeme entwickelt. Der Begriff der eingebetteten DBMS wird von zwei Sichtweisen geprägt. Zum einen werden darunter DBMS verstanden, die im Bereich der eingebetteten Systeme verwendet werden. Zum anderen wird diesbezüglich auch Datenmanagementfunktionalität verstanden, die in eine Anwendung integriert ist. Dabei existiert die Instanz der Datenbank nur so lange, wie die Applikation selbst und stellt somit keinen eigenständigen Service dar.

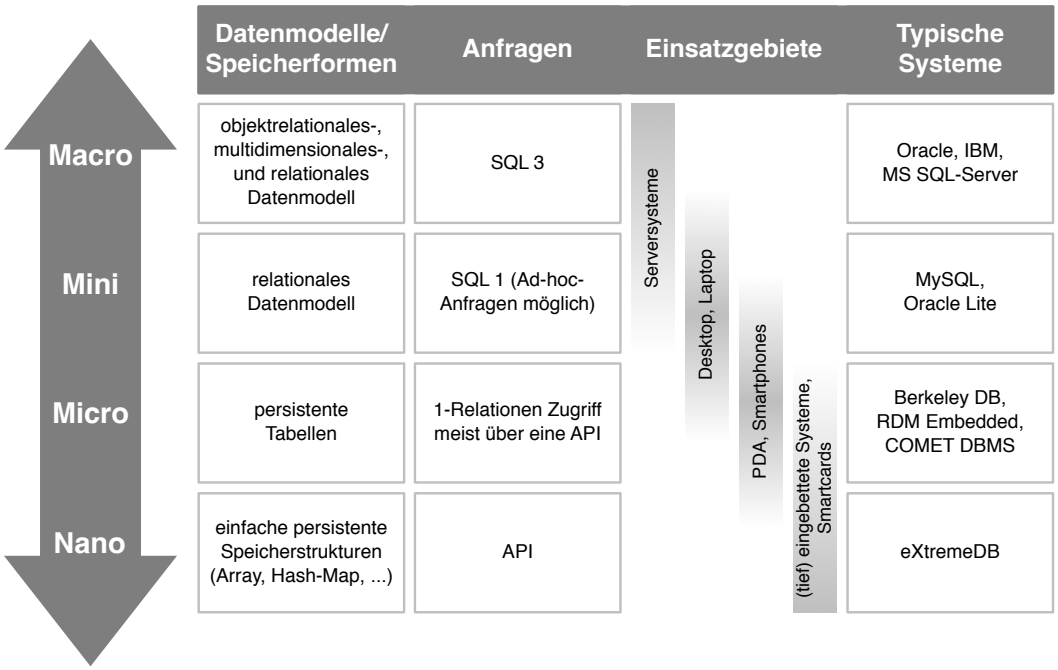

Abbildung 2.8: Kategorisierung des Datenbankmanagements [RLAS07], S. 34 (verändert)

Einordnung des Datenmanagements

Abbildung 2.8 zeigt eine Einordnung des Datenmanagements für eingebettete Systeme. Klassische Datenmanagementsysteme für Serversysteme können dabei als Macrosysteme bezeichnet werden. Minisysteme sind meist abgespeckte Varianten der großen Macrosysteme, die dabei im Desktop-PC-Bereich zum Einsatz kommen. Minisysteme bilden den Übergang zum Bereich der eingebetteten Systeme mit eingeschränkten Ressourcen. Sie sind im Funktionsumfang stark eingeschränkt beziehungsweise sehr stark auf den Anwendungskontext zugeschnitten. Der Bereich der Nanosysteme hingegen bietet stark eingeschränkte Funktionalität, wie beispielsweise Persistenz einzelner Datenstrukturen. Nanosysteme sind vorwiegend für den Einsatz in tief eingebetteten Systemen konzipiert und sind somit Hauptgegenstand dieser Arbeit.

Entwicklung des Datenmanagements

Die Entwicklung von DBMS für eingebettete Systeme unterscheidet sich sehr stark von der Entwicklung klassischer Client-Server-DBMS. Klassische DBMS werden in Hinsicht des Datendurchsatzes, all umfassender Flexibilität, Skalierbarkeit und eines möglichst großen Funktionsumfangs entwickelt. Die ausführbare Programmcodegröße, der Ressourcenverbrauch oder die Prozessorleistung werden, wenn überhaupt, nur untergeordnet betrachtet. Die wichtigsten Ziele, die bei der Entwicklung des eingebetteten Datenmanagements zu beachten sind, können wie folgt zusammengefasst werden:

Minimierung des Speicherverbrauches: Auf Grund des enormen Kostendrucks, dem eingebettete Systeme meist unterliegen, ist die Minimierung des Speicherverbrauchs eines der wichtigsten Ziele. Hierbei muss zum einen der Speicherverbrauch für die Datenmanagementfunktionalität so niedrig wie möglich sein. Zum anderen soll durch den geschickten Einsatz von stark spezialisierten Algorithmen besser auf den Bereich der eingebetteten Systeme reagiert werden.

Minimierung des CPU Verbrauches: Durch die Entwicklungen im Bereich der Prozessoren ist die CPU-Leistung beim Bau klassischer DBMS kein limitierender Faktor mehr. Im Bereich der eingebetteten Systeme, in dem sich in den überwiegenden Fällen der Datenmanagement-Prozess und die Applikation die Ressourcen teilen, ist es immer noch besonders wichtig, dass der Datenmanagement-Prozess sehr sparsam mit dieser Ressource umgeht. Neben den Kosten ist ein weiterer wesentlicher Faktor vor allem der Stromverbrauch, der durch höhere Prozessorleistung steigt. Die damit verbundene Wärmeentwicklung ist nur ein Problem.

Unterstützung unterschiedlicher Hardware und Betriebssysteme: Der Hardwarebereich der eingebetteten Systeme ist sehr heterogen. Vergleicht man den Hardwarebereich der Server und PC-Systeme mit der Hardware von eingebetteten Systemen, so kann im Server und im PC-Bereich von einer vergleichsweisen Homogenität der Hardware ausgegangen werden. Einhergehend mit der hohen Heterogenität der Hardware und Infrastruktursoftware im Bereich der eingebetteten Systeme entsteht auch ein großer Variantenraum, der von den Entwicklern der Datenmanagementsoftware für eingebettete Systeme berücksichtigt werden muss.

Zur Zeit existieren eine Reihe von unterschiedlichen DBMS oder Bibliotheken von Datenmanagementfunktionalität, die im weiteren Umfeld der eingebetteten Systeme eingesetzt werden. Tabelle 2.1 zeigt verschiedene DBMS im Bereich der eingebetteten Systeme. Untereinander sind die Systeme schwer vergleichbar. Dies liegt an dem sehr unterschiedlichen Funktionsumfang der DBMS. In größeren eingebetteten Systemen ist Berkeley DB eines der am weit verbreitetsten Systeme. Im Abschnitt 2.5.1 stellen wir Berkeley DB genauer vor.

Variable Forschungsprototypen

Viele DBMS im Bereich der eingebetteten Systeme sind auch im Forschungsumfeld entstanden. Vertreter dieser Forschungsprototypen sind beispielsweise: *TinyDB* [MFHH05a], *PicoDBMS* [PBVB01], *XXL-Library* [BBD+01], *COMET DBMS* [MFHH05b] und *Cougar* [YG02]. Mit verschiedenen Variabilitätstechniken sind diese Systeme erweiterbar beziehungsweise konfigurierbar. Im Folgenden sollen zwei Vertreter genauer vorge-

DBMS	Herausstellungsmerkmal	Speicher-verbrauch (minimal)	Einsatzgebiet
Berkeley DB C-Version	konfigurierbares, performanzoptimiertes DBMS	484 KB	Netzwerk-Router, Switch, Smartphone, WAP Gateway, MP3-Player
Berkeley DB Java-Version	Java-fähiges Betriebssystem	850 KB	Smartphone, Mobile Java-Applikationen
DB2-Everyplace	Optimiertes DBMS für die Synchronisation von Daten der Serversysteme auf mobile Geräte; Unterstützung der Betriebssysteme PalmOS, WinCE u. a.	150 KB	Smartphone, PDA und Handhelds
DB4o	NoSQL Datenbank	600 KB	Smartphone
eXtremeDB	optimiert für eingebettete Systeme, SQL89-API, ACID-Transaktionen, unterstützt mehr als 12 verschiedene OS; auch 64-Bit Unterstützung möglich	50 KB	Set-Top Boxes, MP3 Player, Mobile Phone Handset, Netzwerk-Router, WiMAX-Basisstationen
HSQLDB	Plattformunabhängiges DBMS; setzt Betriebssystem mit Java-Unterstützung voraus; unterstützt SQL92 und Teile von SQL99 und SQL2003;	100 KB	integriert in OpenOffice, aber auch in spezieller Version für PDA und Handhelds
LGeDBMS	Optimiert für Flash-Speicher; untersützt SQL und Transaktionen; Betriebssystem: REX, pSOS, QNX, Windows, Linux MacOS	600 KB	PDA, Smartphone
Metakit	für 16- bis 64-Bit Architekturen konzipiert; unterstützt Windows, Linux und MacOS;	125 KB	eingebettet in Adressenanwendungen
RDM-Embedded	Native- und SQL-API; Verschiedene Index-Strukturen, MVCC, In-Memory Database Betriebssystem: AIX, Linux ARM, QNX, Windows CE;	270 KB	PDA, Smartphone
SQLite	Eingeschränkter SQL92-Zugriff; Transaktionen und Recover; Index; Betriebssystem: Linux, MacOS, Android, iOS, Win32, WinCE, WinRT (ANSI C);	300 KB	PDA, Smartphone, MP3-Player

Tabelle 2.1: Beispiele Datenmanagementsysteme im Bereich der eingebetteten Systeme

stellt werden. Dies ist zum einen Berkeley DB, das es von einem im Forschungsumfeld entstandenen Prototyp in die Praxis geschafft hat und im Bereich des eingebetteten Datenmanagements weit verbreitet ist. Zum anderen wollen wir COMET DBMS vorstellen, da es sich hierbei um einen Forschungsprototyp handelt, der moderne Variabilitätstechniken bezüglich der Produktlinientechniken im Bereich des Datenmanagements für eingebettete Systeme erforscht. Der Fokus der folgenden Betrachtung der beiden Systeme liegt dabei nicht auf der funktionalen Analyse der angebotenen Algorithmen und Verfahren, sondern im Bereich der nicht-funktionalen Eigenschaft Variabilität.

2.5.1 Berkeley DB

Berkeley DB allgemein

Es gibt zwei verschiedene Varianten von Berkeley DB [Sel07], zum einen eine Java-Version[18] und zum anderen eine C Version. Für beide Versionen ist der Quelltext in Form einer *Open Source*-Lizenz frei zugänglich. Abbildung 2.9 zeigt den Produktaufbau von Berkeley DB. Die Implementierung von Berkeley DB mit der Programmiersprache

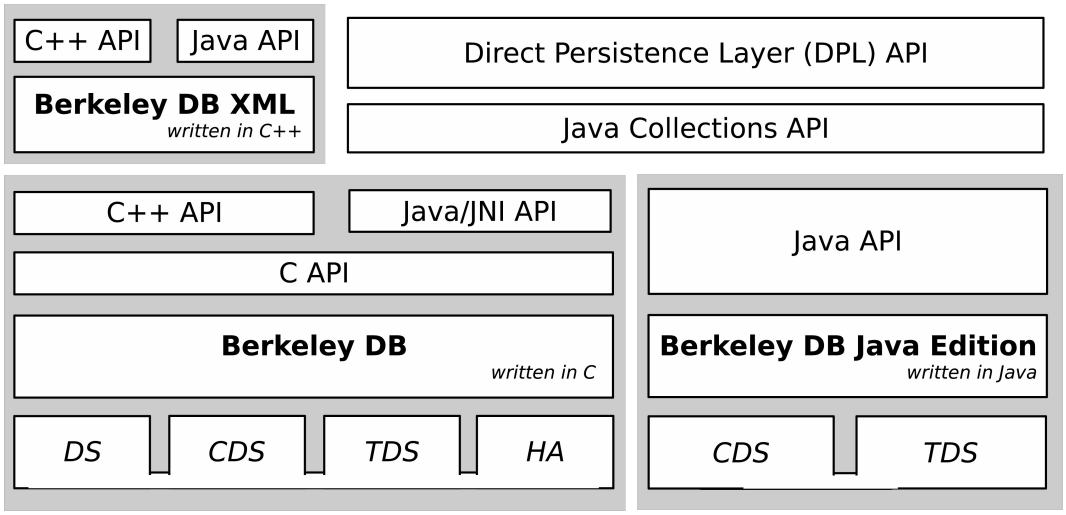

Abbildung 2.9: Produktausbau von Berkeley DB [Sel07], S. 23

C ist die leistungsstärkere Variante. Durch eine systemnahe Implementierung ist die C-Version besonders auf Performanz und Programmgröße optimiert. Berkeley DB wird in einem großen Anwendungsgebiet eingesetzt. Beispiele sind die Produktdatenbank von Amazon, *Set-Top*-Boxen von Sony, *Digitalrecorder* von Samsung, *SIP-Router* von Cisco oder *Smartphones* von Motorola. Je nach Anwendungsgebiet können die Varianten von Berkeley DB unter anderem transaktional mit *Recovery*-Funktionalität Daten verwalten oder in einer performanzoptimierten Variante nur mit einer einfachen Synchronisation vor dem gleichzeitigen Zugriff auf das gleiche Objekt schützen.

Kennzahlen und Aufbau von Berkeley DB

Der für die Evaluierung verwendete C-Quelltext von Berkeley DB in der Version 4.4.202 umfasst ca. 96.000 Zeilen. Dieser ist, wie in C üblich, in ca. 280 Dateien modularisiert. Die Basis für die Entwicklung von Berkeley DB ist die Strukturierte Programmierung. Diese wurde teilweise um die Objektbasierte Programmierung[19] erweitert.

18 Berkeley DB Java-Version ist nicht Gegenstand der Untersuchung, da sie nicht auf Konfiguration ausgerichtet ist. Des Weiteren ist die Version ungeeignet, weil sie nicht auf Ressourcenschonung abzielt.

19 Die Objektbasierte Programmierung versucht mit Mitteln der Strukturierten Programmierung Eigenschaften der Objektorientierten Programmierung nachzuahmen.

Variabilität

Einige Merkmale von Berkeley DB sind optional und lassen sich somit für die Erstellung unterschiedlicher Varianten komplett deaktivieren. Eine minimale Variante von Berkeley DB hat eine Größe von 416 KB. Die vollumfängliche Konfiguration von Berkeley DB mit allen möglichen optionalen Merkmalen wie Replikation und verschiedenen Indexstrukturen hat einen binären Programmcode von 664 KB. Damit beläuft sich die Größe der minimalen Variante noch auf ca. 71 Prozent der vollständigen Variante. Um diese Merkmale zu trennen, wurde in vielen Fällen eine direkte Zuordnung von Dateien zu Merkmalen von Berkeley DB realisiert. Zusätzlich werden hierzu für die unterstützten Betriebssysteme (Linux, Windows, BSD UNIX, etc.) Teile des Quelltextes generiert, um eine Anpassung an die Systemkonfigurationen zu ermöglichen. Der Quellcode von Berkeley DB ist sehr stark mit Konfigurationsanweisungen für unterschiedliche Varianten verwoben. Die Konfigurierbarkeit von Berkeley DB wird mit dem Präprozessor CPP umgesetzt. In Berkeley DB werden hierzu sowohl typische bedingte Übersetzungsanweisungen als auch Makros verwendet. Auffallend sind die zum Teil sehr langen und verschachtelten Funktionen (vgl. hierzu Abbildung 2.10) sowie die extensive Nutzung von Makros. Die Folge ist eine Verschlechterung der Verständlichkeit des Programmcodes [SC92]. Des Weiteren beheben Präprozessoranweisungen nicht das Problem von getrenntem Quelltext, da die Quelltextelemente nur virtuell durch die Präprozessoranweisungen aufgeteilt sind. Dies führt unter anderem zu repliziertem Quelltext und im Folgenden wiederum zu Problemen bei der Erweiterung und Wartung des Systems [TSH04]. Eine ausführliche Diskussion zu den Problemen des Präprozessors CPP führen wir in Abschnitt 3.3.

```
1  __rep_queue_filedone (...)
2  {
3   #ifndef HAVE_QUEUE
4     COMPQUIET(rep, NULL);
5     COMPQUIET(rfp, NULL);
6     return (...);
7   #else
8      db_pgno_t first , last;
9      u_int32_t flags;
10     int empty, ret, t_ret;
11  #ifdef DIAGNOSTIC
12  DB_MSGBUF mb;
13  #endif
14    ... //92 Lines of Code
15  #endif
16 }
```

Abbildung 2.10: Quelltextausschnitt von Berkeley DB mit verschachtelten Präprozessoranweisungen

2.5.2 COMET DBMS

COMET DBMS

Das COMET DBMS Projekt ist ein Gemeinschaftsprojekt der Universität Linköping und der Mälardalen Universität in Schweden. Als Praxispartner arbeitet unter anderem der Automobilhersteller Volvo in diesem Projekt mit. Ein wesentliches Ziel des noch nicht abgeschlossenen Projektes ist es, neue Software-Engineering-Methoden für den Bereich des Echtzeitdatenmanagements in tief eingebetteten Systemen zu entwickeln und zu etablieren. Eine Analyse existierender Ansätze zeigte auf, dass es keine ausreichend maßgeschneiderte Datenmanagementfunktionalität für den Bereich der eingebetteten Echtzeityssysteme gibt, die die Belange der Ressourcenknappheit und der Variabilität der eingebetteten Echtzeitsysteme berücksichtigt [NNT+04, TNH+03, NTNH03]. Auf Wunsch der Praxispartner wurde als Basissprache für die Umsetzung C verwendet.

Architektur

Aus den Ergebnissen einer Analyse bestehender maßschneiderbarer Datenmanagementfunktionalität wurde schnell ersichtlich, dass eine Trennung der Funktionalität in kleine möglichst unabhängige Komponenten mit den bisher verwendeten Methoden nicht möglich ist. Als typische Vertreter gelten hier die konkurrierenden Zugriffe und die Transaktionsverwaltung, da sie viele andere Komponenten direkt betreffen. Aus diesem Grund wurde nach neuen Möglichkeiten der Zerlegung von Komponenten und Aspekten (vergleiche hierzu Abschnitt 3.5) geforscht. Die Grundlage für die Zerlegung bildet das speziell hierfür entwickelte *ACCORD*-Konzept (*Aspectual Component-based Real-Time System Development* (ACCORD)) [TNHN03]. Auf Basis von *ACCORD* können beliebige Echtzeit-Softwaresysteme in Komponenten und Aspekte zerlegt werden. Der derzeitige Prototyp von COMET DBMS besteht aus den folgenden sechs Komponenten (vgl. hierzu auch Abbildung 2.11):

User Interface Component (UIC): Die Benutzerschnittstellenkomponente dient als abstrakte Schnittstelle zwischen Applikation und Datenbankfunktionalität. Neben einer sehr eingeschränkten Menge an SQL-ähnlichen Operationen wie *insert*, *update*, *delete* und *select* können auch Datenbanktransaktionen durch ein *BeginTrans*, *Commit*, *About* und *Rollback* realisiert werden. Die wesentliche Aufgabe ist dabei die Überführung der SQL-ähnlichen Notationen in Datenbankinterne Operationen. Eine Optimierung der Anfragen findet nicht statt, da alle Operationen nur eine physische Repräsentation besitzen.

Scheduling Manager Component (SMC): Die Komponente des *Scheduling Managers* übernimmt die Verwaltung der Transaktionen. Dabei werden die eingehenden Transaktionen je nach gewähltem Transaktionsprotokoll in zwei Listen verwaltet.

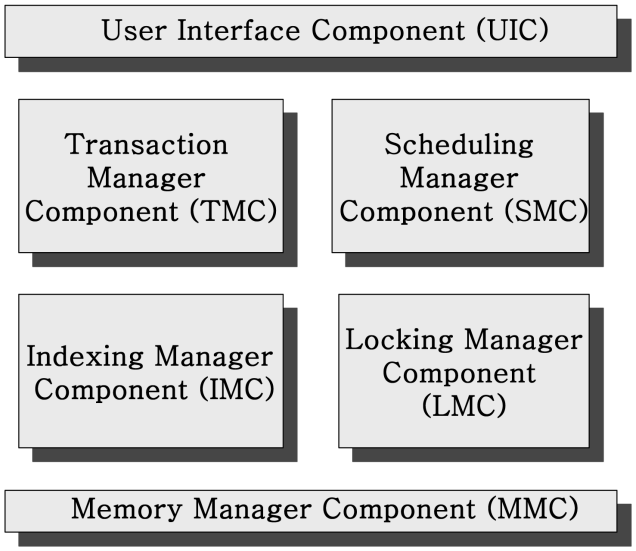

Abbildung 2.11: Aufbau von COMET DBMS [TNH+03], S. 571 (verändert)

Eine Liste enthält alle aktiven Transaktionen. In einer zweiten Liste werden im Gegensatz dazu alle Transaktionen verwaltet, die fertig zur Ausführung bereit stehen. Diese Liste wird vor allem dann benötigt, wenn, wie bei Echtzeittransaktionen, vor der Ausführung eine Priorisierung der Anfragen notwendig ist. Werden durch die Anwendung keine Transaktionen benötigt, so kann diese Komponente vollständig herausgelöst werden. Ein wesentlicher Nachteil dieser SMC ist die starke Spezialisierung auf Echtzeittransaktionen.

Locking Manager Component (LMC): Die Sperrverwaltung organisiert das Sperren von Datensätzen zur Serialisierung von Transaktionen und zur Vermeidung von Verklemmungen der Transaktionen. Gesperrt werden Lese- und Schreibzugriffe auf *Tuple-Identifier*.

Indexing Manager Component (IMC) : Der Index-Manager verwaltet die verschiedenen Indexstrukturen, die durch das DBMS angeboten werden. Derzeitig werden vom Indexmanager nur T-Bäume[20] [LNT00] und B-Bäume unterstützt. Ein Hauptgrund, warum bisher nur wenige Indexstrukturen integriert wurden, sind die Schwierigkeiten bei der Integration von Synchronisationsfunktionalität.

20 Diese Indexstruktur eignet sich besonders gut für Daten, die nur im Hauptspeicher gehalten werden (sog. Main-Memory-Datenbanken).

Transaction Manager Component (TMC): Der Begriff des Transaktionsmanagers ist im Zusammenhang mit den Aufgaben, die diese Komponente ausführt, missverständlich gewählt. Als eine Art zentrale Instanz übernimmt diese Komponente das Zusammenspiel zwischen Sperrverwaltung, dem *Scheduling Manager* und des Speichermanagers. Zusätzlich ist ein Puffermanager Teil des Transaktionsmanagers.

Memory Manager Component (MMC): Der Speichermanager übernimmt die physische Speicherung der Daten und die Verwaltung des Sekundärspeichers. Dabei abstrahiert der Speichermanager die unterschiedlichen Lese- und Schreiboperationen sowie das Allokieren und Freigeben von Speicher auf unterschiedlichen Hard- und Softwareplattformen.

Variabilität

Die sechs Komponenten bilden für sich gesehen noch kein funktionstüchtiges DBMS. Erst durch das Einweben von Aspekten wird aus den einzelnen Bausteinen eine vollständig funktionstüchtige Datenbankinstanz. Dabei übernehmen drei verschiedene Arten von Aspekten die Integration bezüglich der Anwendung und die Konfiguration des DBMS:

1. *Applikationsaspekte:* Die Applikationsaspekte erweitern die Applikation, die COMET DBMS benutzen wollen, um die notwendige Funktionalität zur Nutzung dieser Datenmanagementfunktionalität. Die Applikationsaspekte dienen als Verbindung von Anwendung und Datenmanagement.

2. *Laufzeitaspekte:* Die Laufzeitaspekte weben Überwachungsquelltext zur Reflexion des Systems in Bezug auf die Laufzeiteigenschaften ein. Besonders das Verhalten über die Ausführungszeiten und Speicherallokationen, das für die Berechnung von Echtzeiteigenschaften benötigt wird, kann mit diesen Aspekten, in Abhängigkeit vom unterliegenden Betriebssystem, in die Hauptkomponenten eingefügt werden.

3. *Kompositionsaspekte:* Die Kompositionsaspekte übernehmen das Zusammenbinden der verschiedenen Komponenten. Sie weben neben der Komponenteninfrastruktur auch *Glue-Code* zum Verbinden der Komponenten untereinander in den Komponentencode. Somit soll eine maximale Unabhängigkeit der Komponenten untereinander bis zum Webeprozess gewährleistet sein. Im Quelltext dieser Kompositionsaspekte werden die Kompositionsmöglichkeiten explizit abgelegt.

Fazit

Bezüglich der sechs Komponenten ist COMET DBMS recht starr organisiert. Lediglich der Austausch der Speicherverwaltung, der Serialisierungskomponente und die Wahl der Indexstruktur sind recht gut möglich. COMET DBMS unterstützt auf dem

Architekturlevel den Austausch aller sechs Komponenten. Allerdings stehen keine alternativen Komponenten zur Verfügung, die statt dessen ausgewählt werden können. Des Weiteren gibt es auch kein Konzept, wie diese Komponenten über den angebotenen Funktionsumfang hinaus erweitert werden können. Will man beispielsweise den *Scheduling Manager Component* oder die *User Interface Component* neu implementieren, müssen alle Aspekte, die diese beiden Komponenten erweitern, neu implementiert werden. Eine Wiederverwendung der Aspekte oder ein Framework zur Erweiterung derselben existiert nicht.

2.5.3 Diskussion

Variabilität

Berkeley DB und COMET DBMS zeigen auf verschiedene Weise Konfigurierbarkeit von Datenmanagementfunktionalität. Während das Austauschen von Index-Strukturen und die Abstraktion der Speicherverwaltung gut zu organisieren sind, gibt es Funktionalität wie etwa Transaktionsverwaltung und Synchronisation, die schwer zu kapseln sind. Trotz des sehr feingranularen Konfigurationsmechanismus' durch den Präprozessor ist die Transaktionsverwaltung von Berkeley DB nur schwer zu konfigurieren und variabel zu erweitern. Dies bestätigt auch KÄSTNER [Käs07a] für die Java-Version. Der Grund sind querschneidende Belange und Merkmalsinteraktionen [Käs07a]. In COMET DBMS werden diese querschneidenden Belange durch Aspekte zwar separiert, aber fast alle variablen Ansätze im Bereich des Datenmanagements vermögen keine klare Trennung von Konfigurationsmechanismus und Quelltext. COMET DBMS versucht diese Trennung mit Aspekten zu erreichen. Dies gelingt nur insofern, dass die Konfigurationsanweisungen nun im Aspektcode zusammengefasst werden.

2.6 Forschungsziele der Arbeit

Abgrenzung zu bisherigen Forschungsansätzen

Die Notwendigkeit von hochspezialisierter Datenmanagementfunktionalität, die leicht an unterschiedliche Anforderungen anpassbar ist, wird schon seit Jahren für eingebettete Systeme propagiert. CHAUDHURI und WEIKUM [CW00] plädieren beispielsweise für ein Datenmanagement im RISC-Style und beschreiben Eigenschaften, die ein solches System bieten müsste. Eine Diskussion der Probleme aus Software-Engineering-Sicht wird nicht geführt. Nennenswerte Umsetzungen, der von CHAUDHURI und WEIKUM [CW00] beschriebenen RISC-Style-Architektur, die den Bereich der tief eingebetteten Systeme genügen und ausreichend evaluiert sind, gibt es nach unserer Auffassung nicht. Anders als existierende Ansätze im Bereich der eingebetteten Systeme, wie etwa Berkeley DB [Sel07], *TinyDB* [MFHH05a], *PicoDBMS* [PBVB01], *COMET DBMS* [MFHH05b] und

Cougar [YG02], die eher einen von funktionalen Eigenschaften der Systeme dominierten Forschungsansatz präsentierten, wollen wir Software-Engineering-Aspekte in unseren Ansatz mit einbeziehen und diese im Umfeld des Nanodatenmanagements weiterentwickeln. Hierzu wollen wir in den drei folgenden Bereichen unsere Ziele verfolgen:

Domänenanalyse im Umfeld des Nanodatenmanagements: Das erste Ziel der Arbeit ist die exemplarische Untersuchung von datenhaltungsspezifischen Variationspunkten im Bereich der eingebetteten Systeme, um dadurch ein besseres Verständnis für die Abhängigkeiten von Datenhaltungsmerkmalen untereinander zu bestimmen. Hierzu soll die Granularität für die Zerlegung der Merkmale in der Domäne der eingebetteten Systeme bestimmt werden. Des Weiteren sollen Techniken zur systematischen Aufarbeitung von Abhängigkeiten in der Domäne auf Implementierungsebene untersucht werden. Diese funktionale Sichtweise benötigen wir als Grundlage für unsere folgenden Untersuchungen.

Software-Engineering für eingebettete Systeme: Wir glauben, dass modernes, produktlinienorientiertes Software Engineering, insbesondere die *Merkmalsorientierte Softwareentwicklung*, ein adäquates Konzept darstellt, das die unterschiedlichen Anforderungen im Bereich der eingebetteten Systeme vereint, um angepasstes variables Datenmanagement für diese Systeme bereitzustellen. Dies soll dem Nutzer bei der Erstellung seiner Anwendung unterstützen, ein optimal konfiguriertes Datenmanagement zu erhalten. Das zweite Ziel ist es, über eine Weiterentwicklung von Konzepten des konfigurierbaren Datenmanagements hin zu modernen Produktlinientechniken besser auf die Bedürfnisse von Anwendungsentwicklern im Bereich der eingebetteten Systeme reagieren zu können. Hierzu sollen entsprechende Konzepte und Techniken untersucht und gegebenenfalls angepasst beziehungsweise erweitert werden. Im Fokus unserer Untersuchungen wird der Produktlinienansatz der Merkmalsorientierten Softwareentwicklung stehen. Dieser soll bezüglich der querschneidenden Belange, Granularität, Robustheit gegen ungeplante Änderungen, Praktikabilität, Automatisierbarkeit der Konfiguration sowie Werkzeugunterstützung zur Beherrschung großer Produktlinien untersucht und auf das Anwendungsgebiet des Nanodatenmanagements erweitert werden.

Ressourcenoptimierung: Modernes Software-Engineering und moderne Programmiersprachenentwicklung fokussieren auf Variabilität und Konfiguration, vor allem auch zur Laufzeit. Ressourcenoptimierung ist selten Gegenstand dieser Forschungen. Viele Varianten der Merkmalsorientierten Programmierung stellen ebenfalls nicht die Ressourcenoptimierung in den Mittelpunkt. Mit Bezug auf die

stark ressourcenbeschränkten Umgebungen im Bereich der eingebetteten Systeme soll als Ziel die Anpassung der Merkmalsorientierten Programmierung an diese Bedürfnisse erreicht werden.

Vorgehen und Ergebnisbewertung

Auf Basis einer explorativen Studie untersuchen wir an einem Fallbeispiel aus der Domäne der eingebetteten Systeme den Ansatz der Merkmalsorientierten Programmierung zur Gestaltung von feingranularen Produktlinien mit einer hohen Anzahl an Variationspunkten. Die sich ergebenden Probleme durch die von uns eingesetzte Implementierungsmethode werten wir systematisch aus. Aus den hieraus entwickelten Anforderungen werden unsere Werkzeuge und Methoden als Lösungsvorschlag abgeleitet. Diese entwickelten Werkzeuge und Methoden evaluieren wir hinsichtlich unserer Ziele an drei nicht trivialen Fallstudien aus dem Bereich des eingebetteten Datenmanagements.

2.7 Zusammenfassung

In diesem Kapitel zeigten wir einen Überblick zum Thema Datenmanagement für eingebettete Systeme. Dazu führten wir zunächst in die wichtigsten Eigenschaften und Anwendungsgebiete von eingebetteten Systemen ein. Anschließend wurden anhand von Beispielszenarien, unter anderem aus dem Automobilbau, Anforderungen an das Datenmanagement für eingebettete Systeme evaluiert. Des Weiteren gaben wir einen Überblick zum Themenfeld des variablen und erweiterbaren Datenmanagements sowie von Datenmanagementlösungen im Bereich der eingebetteten Systeme. Am Beispiel von zwei Vertretern wurden Probleme bei der Umsetzung von Konfigurationsmechanismen aufgezeigt. Geschlossen wird das Kapitel mit der Definition der Zielstellung der Arbeit.

KAPITEL 3

Grundlagen: Software-Engineering

Kurze Entwicklungszeiten, eine hohe Produktqualität und eine kostengünstige Wartung von Softwaresystemen sind Schlüsselindikatoren für eine gute Softwareentwicklung. Ein Hauptziel der Softwareentwicklung ist es, durch eine gezielte Wiederverwendbarkeit von Teilen der Software oder anderer Software diese Schlüsselfaktoren positiv zu beeinflussen. Demgegenüber steht der Fakt, dass die Komplexität des Entwicklungsprozesses durch die gestiegenen Ansprüche an die heutigen Softwaresysteme und an Individualität überproportional wächst. Damit wird die gezielte Wiederverwendung von Teilen der Software in immer größerem Umfang verhindert. Softwareproduktlinien versprechen, diese Komplexität besser zu beherrschen. Da die Wiederverwendung in der Softwareentwicklung seit der Propagierung der Softwarekrise immer wieder mit unterschiedlichem Fokus betrachtet wurde, sind viele Begriffe unterschiedlich belegt. Dieses Kapitel führt in die für diese Arbeit notwendigen Definitionen des Software-Engineering ein. Dabei stellt es aber keinen Survey oder historischen Überblick dar, sondern greift nur die für diese Arbeit notwendigen Themengebiete auf.

3.1 Grundbegriffe

In diesem Abschnitt werden zunächst grundlegende Begriffe definiert. Begriffe, die häufig im Umfeld der Softwareentwicklung genannt werden, sind (*De-)Komposition, Trennung von Belangen* und *Modularisierung*. Des Weiteren wird in die Begriffe der *Produktlinien, Programmfamilien* und den in den letzten Jahren häufig diskutierten Begriff der *querschneidenden Belange* eingeführt.

3.1.1 (De-)Komposition, Trennung von Belangen, Modularisierung

Eine Trennung verschiedener Funktionalitäten eines Programms in einzelne verständliche *Komponenten* beziehungsweise *Module* ist einer der Schlüsselfaktoren für eine erfolgreiche Implementierung einer komplexen Software.

Software (De-) Komposition

Das Zusammensetzen einer Software aus kleineren Bestandteilen wird als *Softwarekomposition* bezeichnet. Der hierzu notwendige Prozess der Zerlegung einer Software in einzelne Teile wird *Dekomposition* genannt. Durch diese Zerlegung der Software in kleinere Bestandteile wird nach dem Prinzip „Teile und Herrsche“ dem Entwickler einer Software die Möglichkeit gewährt, die Komplexität eines Gesamtprogramms zu beherrschen. Streng genommen umfasst der Begriff der Dekomposition noch nicht, wie eine Software zerlegt wird. Eine Möglichkeit wäre die Zerlegung der Software in Codeblöcke von jeweils 10 Zeilen. Dass eine solche Zerlegung nicht sinnvoll ist, gilt sicherlich als unbestritten.

Trennung von Belangen

PARNAS [Par76, Par79] und DIJKSTRA [Dij76] definierten mit der *„Trennung von Belangen“* (engl. *seperation of concerns*) erstmals ein Prinzip, nach dem eine Dekomposition idealerweise stattfinden sollte. Jede Einheit der zerlegten Software soll demnach eine wohldefinierte Aufgabe ausführen oder einen Belang[1] (engl. *concern*) des Programms darstellen. Dabei soll die semantisch zusammengehörige Funktionalität kohärent gekapselt werden. Bis heute ist der Begriff *concern* in der Literatur nicht einheitlich geklärt (vergleiche hierzu z. B. TARR et al. [TO00], (STANLEY et al. [SMSR02], KANDÉ [Kan03]). Eine aus der Sichtweise dieser Arbeit vertretbare Definition liefert das *Institute of Electrical and Electronics Engineers, Inc. (IEEE)* [IEE00]:

> *„Concerns are those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders." [IEE00], S. 4*

Ein Beispiel für die Trennung von Belangen ist nach OSSHER und TARR [OT00] der objektorientierte Entwurf, der Software in Klassen aufteilt. Um die Belange einer Software isoliert betrachten zu können, sollten diese so wenig Wissen wie möglich über andere Belange besitzen.

1 Oftmals wurde auch in den Originalveröffentlichungen der engl. Begriff *„point of interest“* benutzt.

Auswirkungen einer sauberen Trennung

Eine saubere Trennung der unterschiedlichen Belange einer Software hat nach OSTERMANN [Ost03] positive Auswirkungen auf die Qualität einer Software. Diese Auswirkungen können in vier verschiedenen Dimensionen betrachtet werden:

Verständlichkeit: Einer der Hauptgründe, warum eine Software in kleinere Teile zerlegt werden soll, liegt in den natürlichen Verarbeitungsmöglichkeiten des menschlichen Gehirns begründet. Durch eine isolierte Betrachtung kleinerer Teile einer Software und einer Ausblendung aller Abhängigkeiten kann bei späterem Zusammensetzen der Software ein besseres Gesamtverständnis für die Software erreicht werden. Dabei lautet die Grundthese: Je besser eine Zerlegung einer Software in unabhängige kleine Teile gelingt, desto einfacher ist es für den Menschen, ein Verständnis für das gesamte Softwaresystem zu erreichen.

Wiederverwendung: 1968 formulierte MCILROY [McI68] als Erster das Konzept einer formellen Wiederverwendung von Softwarefragmenten. Dabei nannte er die Hardwareindustrie als Vorbild, die durch die Wiederverwendung von Komponenten enorme Kosteneinsparungen realisieren konnte. Grundsätzlich kann die Wiederverwendung in Kategorien aufgeteilt werden. Zum einen gibt es die geplante Wiederverwendung, bei der schon beim Entwurf der Komponente die unterschiedlichen Einsatzmöglichkeiten beachtet werden. Zum anderen gibt es die ungeplanten Wiederverwendungen, wobei ein bestimmtes Softwarefragment in einem nicht geplanten Kontext wiederverwendet wird. Um eine möglichst häufige Wiederverwendung dieser Softwarekomponenten zu gewährleisten, ist eine Minimierung der Kontextanbindung (auch: Trennung von Belangen) sicherzustellen.

Wartbarkeit: Unter der Wartung eines Softwaresystems wird die ständige Erweiterung, Verbesserung oder Anpassung einer Software an neue, sich verändernde Anforderungen bezeichnet. Seitdem die Kosten für die Wartung die Entwicklungskosten überholt haben [Boe81], ist die gute Wartbarkeit eines Softwaresystems ein wesentliches Qualitätsmerkmal. Die Aufgaben im Wartungsprozess lassen sich im Wesentlichen auf das Auffinden bestimmter Belange und die Durchführung der notwendigen Änderungen an diesen Belangen zurückführen. Sind die Belange nicht klar getrennt beziehungsweise über das ganze Softwaresystem verteilt, erschwert dies die Wartung desselben enorm.

Skalierbarkeit: Seitdem Softwaresysteme nicht mehr nur von einem Entwickler umgesetzt werden, ist eine skalierbare Softwareentwicklung ein wichtiges Ziel des Software-Engineering. Bei einer vollständigen Skalierbarkeit in der Softwareent-

wicklung lässt sich durch die Erhöhung der Anzahl der Entwickler die Zeit für die Herstellung der Software proportional senken. Dass dies in der Praxis nicht funktioniert, formulierte 1975 BROOKS [Bro75] sinngemäß in der Formel, dass das Hinzunehmen zusätzlicher Entwickler zu einer weiteren Verspätung eines Softwareprojektes führt. Begründet wird dies durch den gestiegenen Kommunikationsaufwand der Entwickler untereinander. Bei einer perfekten Trennung aller Belange gibt es eine maximale Unabhängigkeit der Komponenten.

3.1.2 Querschneidende Belange

Querschneidende Belange und Dekomposition

Trotz der jahrzehntelangen Forschungsarbeit und der elementaren Bedeutung ist das Problem der Trennung unterschiedlicher Belange einer Software immer noch nicht zufriedenstellend gelöst. Die Objektorientierte Softwareentwicklung, die eigentlich durch die Kapselung von Daten und Funktionen in einem Objekt eine Trennung unterschiedlicher realweltlicher Belange ermöglichen sollte, bietet ebenfalls keine zufriedenstellende Lösung dieses Problems. Das Resultat der begrenzten Möglichkeiten einer sauberen Trennung von verschiedenen Belangen besteht in den querschneidenden Belangen (engl. *crosscutting concerns*). Für das Auftreten dieser querschneidenden Belange gibt es zwei Gründe:

1. Die Aufspaltung von Programmen in Komponenten, Klassen, Funktionen usw. erfolgt bei allen gängigen[2] Dekompositionsmechanismen und Programmiersprachen immer nur anhand einer dominanten Dimension [TOH+99, CE00]. Dieses Problem wird nach TARR et al. [TOH+99] auch als *Tyrannei der dominanten Dekomposition* bezeichnet. Gibt es weitere Dimensionen, nach denen ebenfalls eine Dekomposition vorgenommen werden könnte, muss sich diese Struktur an der dominanten Dekomposition ausrichten. Dies führt in vielen Fällen dazu, dass eine gleichzeitige Kapselung dieser Funktionalität nur schwer[3] möglich oder sogar unmöglich ist.

2. Der Softwareentwicklungsprozess unterliegt ständigen Änderungen. Hat sich der Entwickler für eine bestimmte Dekomposition entschieden, so kann sich diese unter Umständen bei ungeplanten Erweiterungen als problematisch herausstellen. Können die neuen Anforderungen nicht anhand der bereits existierenden Dekomposition umgesetzt werden, entstehen ebenfalls querschneidende Belange.

2 „Multi-Dimensional Seperation of Concerns“, wie es beispielsweise *HyperJ* [SR03, OT00] anbietet, können eine Aufteilung von Programmen prinzipiell auch in mehreren Dimensionen vornehmen.

3 Diese kann zum Beispiel durch das Hinzufügen einer weiteren Abstraktionsebene ermöglicht werden.

Querschneidende Belange auf Codeebene

Durch die Verbreitung der *Aspektorientierte Programmierung* (AOP) [KLM+97] werden häufig querschneidende Belange auf Quellcode-Ebene in zwei Kategorien unterteilt:

Code Tangling: *Code Tangling* bezeichnet den Umstand, dass der Programmcode eines Belanges mit dem Programmtext eines anderen Belanges verflochten ist, obwohl der eigentliche Bezug der Funktionalität beider Belange nur begrenzt in Verbindung steht.

Code Scattering: Als *Code Scattering* wird die Verstreuung vom Code eines Belanges über verschiedene Teile des Quellcodes bezeichnet. Durch *Code Scattering* geht der Zusammenhalt (Kohäsion) eines Belanges verloren.

Sowohl *Code Tangling* als auch *Code Scattering* erschweren stark die Lesbarkeit, Skalierbarkeit, Wiederverwendung, und damit die Wart- und Erweiterbarkeit des Programmcodes. Im Kapitel 5 werden weitere Ebenen der querschneidenden Belange vorgestellt.

3.1.3 Programmfamilien, Variabilität und Produktlinien

Programmfamilien

Der Begriff der *Programmfamilie* [Par76] wurde von PARNAS [Par76, Par79] bereits 1976 geprägt. Mit dem Begriff Programmfamilie definiert PARNAS eine Menge von Anwendungen, deren Gemeinsamkeiten so weitreichend sind, dass es vorteilhaft ist, diese Gemeinsamkeiten zu analysieren und zu nutzen. Auf Grund der Gemeinsamkeiten werden die einzelnen Anwendungen als Mitglieder der Familie bezeichnet. Die Variation der einzelnen Familienmitglieder resultiert aus den notwendigen Anpassungen der Anwendungen auf den jeweiligen Anwendungskontext[4]. Die wesentlichen Qualitätseigenschaften einer Programmfamilie sind die Variabilität und der Grad der Wiederverwendung.

Schrittweise Verfeinerung nach WIRTH

Eines der wichtigsten Konzepte des Softwareentwurfs ist die schrittweise Verfeinerung (engl. *stepwise refinement*). Im Zusammenhang mit der Entwicklung von Programmfamilien wird dieser Begriff aber oft in zwei Dimensionen verwendet. WIRTH [Wir71] definiert die schrittweise Verfeinerung als einen Top-Down-Entwurfsprozess, bei dem eine Anwendung durch eine Folge von Verfeinerungsschritten entwickelt wird. Beginnend mit einer sehr abstrakten Beschreibung der Funktionalität werden jedem Verfeinerungsschritt weitere Informationen über eine Funktionalität hinzugefügt bis der Detaillierungsgrad hoch genug ist, um die Funktionalität umzusetzen.

4 Als Anwendungskontext werden in diesem Zusammenhang zum Beispiel unterschiedliche Hard- und Softwareumgebungen, Nutzergruppen oder Implementierungen verstanden.

Schrittweise Verfeinerung nach DIJKSTRA *und* PARNAS

Eine implementierungsgetriebene Sichtweise des Begriffs der schrittweisen Verfeinerung vertreten DIJKSTRA und PARNAS [Dij76, Par76, Par78]. Die Bottom-Up Sichtweise schlägt vor, dass ein komplexes Programm aus einer minimalen Basis heraus implementiert werden soll, welche schrittweise durch Erweiterungen eine Weiterentwicklung erfährt. PARNAS [Par76, Par78] schlägt diese Art der Entwicklung für die Entwicklung von Programmfamilien vor.

Variabilität

Unter *Variabilität* wird im Zusammenhang dieser Arbeit die Möglichkeit verstanden, ein Softwaresystem auf Quelltextebene zu verändern oder anzupassen. Ein variables Softwaresystem ermöglicht eine Anpassung an unterschiedliche Anwendungsprofile. Dabei gilt, je höher die Variabilität eines Softwaresystems ist, desto einfacher können Anpassungen oder Änderungen an einem neuen bzw. veränderten Anwendungskontext vorgenommen werden. Die Variabilität eines Systems kann auf unterschiedlichen Ebenen beeinflusst werden [SvGB05]. So kann die Wahl eines geeigneten Softwareentwicklungsprozesses ebenso entscheidend sein wie die Auswahl bestimmter Entwurfs- und Architekturentscheidungen. Ebenso beeinflussen auch geeignete Implementierungstechniken die Variabilität eines Systems.

Zeitpunkte der Variabilität

Ein entscheidendes Qualitätsmerkmal im Zusammenhang mit der Variabilität ist der Zeitpunkt, wann Veränderungen am System vorgenommen werden können. Nach KANG et al. [KCH+90] kann dies grundsätzlich zu drei Zeitpunkten erfolgen:

Konfiguration zur Übersetzungszeit: Bei der Konfiguration zur Übersetzungszeit erfolgt die Festsetzung, welches Systemmerkmal in die Software aufgenommen werden soll, zum Zeitpunkt der Übersetzung des Programms. Die zum Übersetzungszeitpunkt getroffenen Entscheidungen über den Programmaufbau ändern sich während der gesamten Laufzeit des Programms nicht.

Konfiguration zur Ladezeit: Bei der Ladezeitkonfiguration erfolgt die Festlegung, welche Merkmale die Software erbringen soll, zum Zeitpunkt des Programmstarts. Die ausgewählten Merkmale können von Ausführung zu Ausführung der gleichen Software variieren, sind jedoch während der gesamten Ausführungsdauer festgeschrieben. Eine Möglichkeit, um Ladezeitkonfiguration zu ermöglichen, ist die Verwendung von Bibliotheken, welche zur Ladezeit eingebunden werden.

Konfiguration zur Laufzeit: Die Veränderung beziehungsweise der Austausch von Systemmerkmalen zur Laufzeit wird als Laufzeitkonfiguration bezeichnet. Die Logik für die Varianten, die während der Laufzeit geändert werden sollen, muss

dazu in den übersetzten Einheiten vorhanden sein, da eine externe Veränderung ausgeschlossen ist.

Eine ausführliche Diskussion zum aktuellen Stand der Technik bezüglich der Zeitpunkte von Variabilität und der Kosten liefert ROSENMÜLLER [Ros11] in seiner Dissertation.

Verzögerte Entwurfsentscheidungen

Das Verändern bereits festgelegter Entwurfsentscheidungen ist sehr kostenintensiv. Eine Eigenschaft in der Softwareentwicklung ist, dass Entscheidungen, die in frühen Phasen der Softwareentwicklung getroffen werden, in späteren Phasen nur schwer zu ändern sind. Ausgehend von der Theorie, dass jede getroffene Entwurfsentscheidung im Laufe des Softwareentwicklungsprozesses die Variabilität des Systems weiter eingrenzt, leitet sich der Grundsatz der verzögerten Entwurfsentscheidungen ab [SvGB01]. Dies bedeutet für den Bau von variablen Softwaresystemen und besonders für die Umsetzung von Programmfamilien, dass bestimmte Entscheidungen, die die Variabilität des Systems betreffen, so spät wie möglich getroffen werden sollten.

Abgrenzung der Begriffe Produktlinie und Programmfamilie

In den letzten Jahren entwickelte sich ein zweiter Begriff, der in vielen Fällen als Synonym für den Begriff der Programmfamilie verwendet wird. Während Programmfamilien eher auf technische Gemeinsamkeiten fokussieren, definieren sich Mitglieder einer *Produktlinie* über Gemeinsamkeiten auf einem Markt. CLEMENTS et al. [CLN02] sieht sinngemäß:

> *„ ... eine Produktlinie als eine Menge von Softwareprodukten an, welche im Idealfall aus Bausteinen einer oder mehrerer Programmfamilien besteht."* CLEMENTS et al. [CLN02]

Im Umkehrschluss ist es ebenfalls möglich, dass eine Programmfamilie in einer oder mehreren Produktlinien wiederverwendet werden kann [CE00], S. 31. Im weiteren Verlauf der Arbeit werden die beiden Begriffe synonym verwendet.

3.2 Domänen-Engineering

Definition Domänen-Engineering

Das *Domänen-Engineering* ist ein etablierter Softwareentwicklungsprozess, der die systematische Wiederverwendung von Analyse, Entwurfs- und Implementierungsinformationen im Kontext von Programmfamilien beziehungsweise Produktlinien zusammenfasst. Dass die Wiederverwendung von Software nicht nur das Kopieren von Quelltexten beinhaltet, sondern vor allem die Wiederverwendung von Analyse- und Entwurfsergebnissen umfassen muss, erkannte schon NEIGHBORS [Nei80] 1980. Nur die expliziten Darstellungen der Struktur und der Definitionen von Softwarekomponenten ermögli-

chen eine gezielte Wiederverwendung. Hieraus ergibt sich eine Definition des Begriffs *Domänen-Engineering* nach CZARNECKI und EISENECKER [CE00] wie folgt:

> *„Domain Engineering is the activity of collection, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets (i.e., reusable work products), as well as providing an adequate means for reuse these assets (i.e., retrieval, qualification, dissemination, adaptation, assembly, and so on) when building new systems."* CZARNECKI und EISENECKER [CE00], S. 20

Definition Domäne

Der Begriff der *Domäne* ist in der Informatik sehr stark überladen. In dieser Arbeit wird der im Zusammenhang mit Produktlinien und Programmfamilien etablierten Definition von CZARNECKI und EISENECKER [CE00] gefolgt. Sie definieren den Begriff der Domäne wie folgt:

> *„Domain: An area of knowledge*
>
> - *scoped to maximize the satisfaction of the requirements of its stakeholders,*
> - *including a set of concepts and terminology understood by practitioners in that area, and*
> - *including knowledge of how to build software systems (or parts of software systems) in that area."*, CZARNECKI und EISENECKER [CE00], S. 34

Definition Stakeholder

Unter dem Begriff *Stakeholder* werden alle Personen, Gruppen oder Organisationen zusammengefasst, die von der Systementwicklung, vom Einsatz oder vom Betrieb eines Systems in irgendeiner Weise betroffen sind. Dazu gehören auch Personen, die nicht bei der Systementwicklung mitwirken, aber das neue System nutzen oder in Betrieb halten.

Definition Applikation-Engineering

Das *Applikation-Engineering* grenzt sich insofern vom *Domänen-Engineering* ab, als dass bei der Entwicklung von Anwendungen die eigentliche Wiederverwendung von bereits vorhandenen Softwarekomponenten stattfindet. CZARNECKI und EISENECKER [CE00] bezeichnen deshalb auch das *Domänen-Engineering* als die Entwicklung zur Wiederverwendung und die Anwendungsentwicklung als Entwicklung durch Wiederverwendung. Abbildung 3.1 zeigt den Prozess des Domänen und *Applikation-Engineering* nach CZARNECKI und EISENECKER [CE00]. Trotz der klaren Trennung von *Domänen-Engineering* und *Applikation-Engineering* laufen beide Prozesse nicht getrennt voneinander ab. Neues Wissen oder Anforderungen aus der Anwendungsentwicklung fließen kontinuierlich

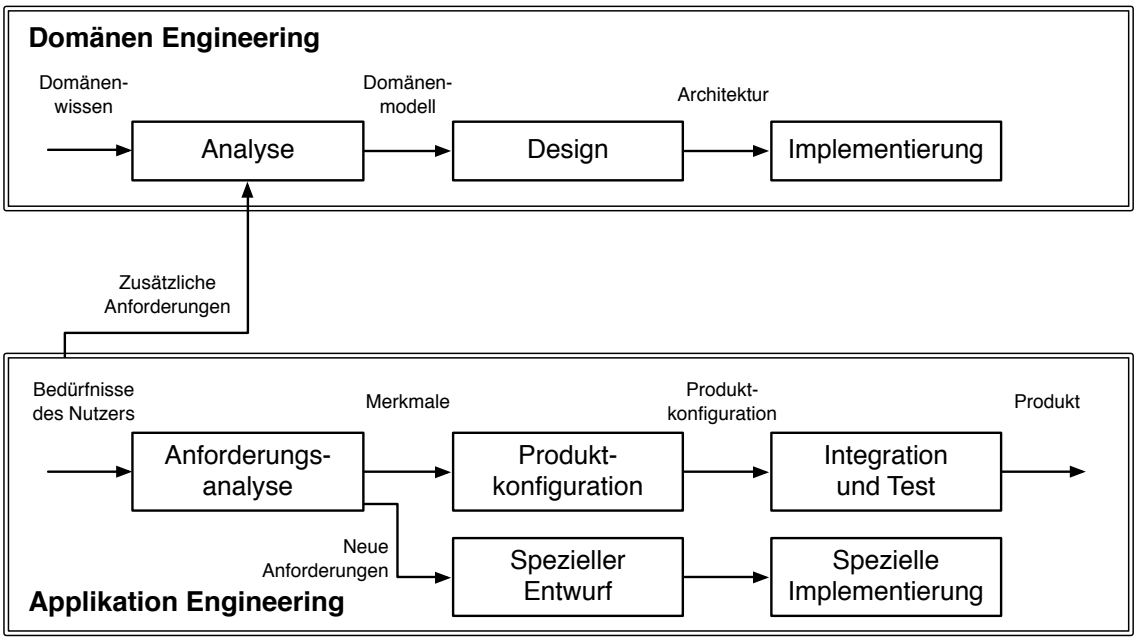

Abbildung 3.1: *Domänen-* und *Applikation-Engineering* nach CZARNECKI und EISENECKER [CE00], S. 21 (verändert)

in das *Domänen-Engineering* zurück. Der Prozess des *Domänen-Engineering* läuft in drei Phasen ab:

Domänenanalyse: Der Begriff der Domänenanalyse wurde erstmals von NEIGHBORS [Nei80] in seinen Arbeiten zu *Draco* geprägt. Die Hauptaufgabe der Domänenanalyse ist die Identifizierung von Anforderungen, die im Kontext der Domäne wiederverwendet werden können. Der Schwerpunkt liegt dabei auf der Analyse der Gemeinsamkeiten und der Unterschiede, die verschiedene Softwareprodukte in einer Domäne besitzen können. Je nach gewählter Methode umfasst die Domänenanalyse in der einen oder anderen Form die Schritte der Domänenabgrenzung und die Domänenmodellierung.

Domänenentwurf: Das Ziel des Domänenentwurfes ist die Entwicklung einer Softwarearchitektur, welche die Eigenschaften der einzelnen Programmfamilienmitglieder berücksichtigt. Dabei steht die Art der Umsetzung im Vordergrund. Des Weiteren ist zu klären, welche Struktur die Programmfamilie besitzen muss, um die konkreten variablen Anwendungen aus der Gesamtarchitektur beziehungsweise aus den wiederverwendeten Einheiten zu erzeugen.

Domänenimplementierung: In der Phase der Domänenimplementierung werden die wiederverwendbaren Softwareartefakte implementiert. Sie bilden dann den Kern der Produktlinie beziehungsweise der Programmfamilie.

In den folgenden Abschnitten werden die einzelnen Phasen vorgestellt. Des Weiteren werden dabei konkrete Umsetzungsmethoden der Phasen, die im Rahmen der Arbeit von Interesse sind, erläutert.

3.2.1 Domänenanalyse

Merkmalsorientierte Domänenanalyse

KANG et al. [KCH+90] entwickelte 1990 die merkmalsorientierte Domänenanalyse (engl. *Feature-Oriented Domain Analysis* (FODA)). Zentrale Abstraktionselemente sind Merkmale (engl. *feature*) und deren Beziehungen. Merkmale sind im Allgemeinen markante Eigenschaften einer Anwendung. KANG et al. [KCH+90] grenzen den Begriff *Merkmal* als Anforderung, die „von außen" sichtbar ist, ein:

> *„Feature: A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems."* KANG et al. [KCH+90], S. 3

Da interne Eigenschaften in dieser Definition der Merkmalsmodellierung nicht berücksichtigt werden, folgt diese Arbeit der Merkmalsdefinition von CZARNECKI und EISENECKER [CE00], S. 38. Diese ist offener formuliert, da auch Eigenschaften auf der Implementierungsebene mit eingeschlossen werden:

> *„Feature: A distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder or the concept."* CZARNECKI und EISENECKER [CE00], S. 38

Merkmalsmodellierung

Mit Hilfe der *Merkmalsmodellierung* gilt es, die Merkmale einer Domäne zu identifizieren, in denen sich die Systeme innerhalb dieser Domäne unterscheiden oder Gemeinsamkeiten aufweisen. Das Ergebnis sind implementierungsunabhängige *Merkmalsmodelle*, die explizit die Variationspunkte einer Produktlinie darstellen. KANG et al. [KCH+90] schlagen zur Darstellung azyklisch gerichtete Graphen vor[5]. Für die weitere Arbeit wird allerdings nicht auf die originale Darstellung von KANG et al. [KCH+90] zurückgegrif-

5 Ähnlichkeiten zu einigen Produktbeschreibungsmöglichkeiten aus der Betriebswirtschaftslehre, die es schon seit Jahrzehnten gibt, sind deutlich sichtbar, werden in den Originalveröffentlichungen aber meist ignoriert.

fen. Statt dessen wird die erweiterte Notation von CZARNECKI und EISENECKER [CE00] verwendet (vgl. Abbildung 3.2).

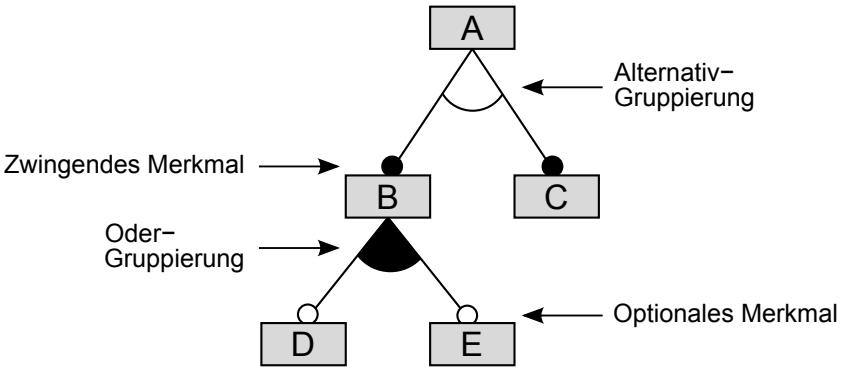

Abbildung 3.2: Konzeptdarstellung eines Merkmalsdiagramms

Merkmals-diagramme

CZARNECKI und EISENECKER [CE00], S. 39 ff. schlagen zur Darstellung der Variationspunkte einer Programmfamilie neben der Verwendung von azyklisch gerichteten Graphen zusätzlich die Verwendung von Kantendekorationen vor. Die hierarchische implementierungsunabhängige Dekomposition in *optionale, alternative, zwingende* und *Oder-Merkmale* definiert die entsprechenden Variationspunkte der Programmfamilie wie folgt:

Zwingende Merkmale: Wird ein Merkmal als Teil der Anwendung identifiziert, so müssen zwingende Sub-Merkmale auch (vgl. Abbildung 3.2) in die Anwendung mit aufgenommen werden. Gekennzeichnet werden die zwingenden Merkmale durch einen gefüllten Kreis.

Optionale Merkmale: Optionale Merkmale können in eine Anwendung mit aufgenommen werden, wenn der Elternknoten Teil der Anwendung ist. Ein optionales Merkmal wird durch einen leeren Kreis gekennzeichnet (vgl. Abbildung 3.2).

Alternative Merkmale: Die Menge der Subknoten eines Merkmals kann auch in Form von alternativen Merkmalen gruppiert werden. Wird das Eltern-Merkmal dann ausgewählt, darf höchstens ein Submerkmal Teil der Anwendung werden. Die Menge der alternativen Submerkmale wird durch einen nicht gefüllten Kreisbogen gruppiert (vgl. Abbildung 3.2).

Oder-Merkmale: Die Menge der Submerkmale eines Knotens kann auch mit Hilfe eines gefüllten Halbkreises gruppiert werden (vgl. Abbildung 3.2). In diesem Fall können ein oder mehrere dieser Submerkmale Teil der Anwendung sein.

Sowohl CZARNECKI und EISENECKER [CE00], S. 39 ff. als auch LEE et al. [LKL02] beschreiben entsprechende Leitfäden für die Merkmalsmodellierung, denen diese Arbeit folgt.

3.2.2 Domänenentwurf und -implementierung

Annotations- und Kompositions-mechanismen

Der Bereich der variablen Architekturen, Entwurfsmethoden, Komponentenmodelle und deren Implementierungen ist im Software-Engineering ein sehr weites Feld, das über den Rahmen dieser Arbeit hinausgeht. In den folgenden Abschnitten wird daher nur ein sehr kleiner Ausschnitt präsentiert, der zum Verständnis der Arbeit beitragen soll. Ein Anspruch auf eine vollständige Darlegung von Entwurfs- und Implementierungskonzepten in diesem Bereich wird nicht erhoben. Die Implementierung von Produktlinien wird derzeit mit verschiedenen Techniken auf unterschiedlichen Abstraktionsebenen wie zum Beispiel auf Werkzeug- oder Sprachebene vom Entwickler durchgeführt. Generell lassen sich nach KÄSTNER [Käs10] diese Mechanismen in die Gruppe der annotationsbasierten Ansätze und in die Gruppe der kompositionsbasierten Ansätze unterteilen:

Annotationsbasierte Ansätze: Bei annotationsbasierten Ansätzen wird der Quelltext mit zusätzlichen Statements versehen, die in der Regel bestimmte Quelltextabschnitte in die zu generierende Variante mit aufnehmen oder nicht. Des Weiteren sind Parametrisierungen des Quelltextes möglich. Der bekannteste Ansatz dieser Gruppe ist der C-Präprozessor. LIEBIG et al. [LAL+10, LKA11] zeigen in zwei Studien eine große Auswahl an derzeitig verfügbaren Produktlinien, die mit Hilfe des C-Präprozessors konfiguriert wurden. Trotz der weiten Verbreitung steht dieser Ansatz stark in der Kritik, da keine oder nur eine unzureichende Trennung der Belange erreicht wird oder diese sehr fehleranfällig sind. Typische Vertreter sind C-Präprozessor *CPP* , *XVCL* [JBZZ03] und *CIDE* [Käs10].

Kompositionsbasierte Ansätze: Unter kompositionsbasierten Ansätzen werden Mechanismen verstanden, die Merkmale in unterschiedlichen Modulen kapseln, wie zum Beispiel Dateien, Klassen oder Plug-ins. Das Zusammensetzen eines konkreten Produktes erfolgt dann im Kompositionsprozess, bei dem die einzelnen Module zu einem Produkt generiert werden. Typische Vertreter sind *Komponenten* [Szy98], *Frameworks* [JF88], *Merkmalsorientierte Programmierung* [Pre97, BSR03], *Aspektorientierte Programmierung* [KLM+97], Gerneratoransätze [CE00] und weitere.

In den folgenden Abschnitten werden für die Arbeit wichtige Vertreter beider Ansätze kurz vorgestellt.

3.3 Präprozessoren

C-Präprozessor (CPP)

Eines der in der Industrie und vor allem im Bereich der eingebetteten Systeme am weitesten verbreiteten Variabiliätswerkzeuge ist der C-Präprozessor *CPP* [Bar99]. Liebig et al. [LAL+10, LKA11] zeigen in zwei Studien eine große Auswahl an derzeitig verfügbaren Produktlinien, die mit Hilfe des CPP zusammengesetzt wurden und werten diese systematisch aus.

Direktive des CPP

Der CPP führt eine textuelle Transformation vor der eigentlichen Übersetzung des Quelltextes durch. Mit Hilfe von Makros und symbolischen Konstanten `#define`, Dateiinklusionen `#include` und Direktiven der bedingten Übersetzung wie `#ifdef`, `#ifndef`, `#if`, `#elif`, `#else` und `#endif` wird die gewünschte Konfiguration für den Übersetzungsprozess vorbereitet. Abbildung 3.3 zeigt den typischen Einsatz des CPP am Beispiel der betriebssystemabhängigen Konfiguration der internen Uhr des eingebetteten Datenmanagementsystems von BerkeleyDB. Abbildung 3.4 zeigt dagegen den massiven Einsatz des CPP, bei dem ein Vorübersetzungsschritt die eigentlichen CPP-Anweisungen auf Zeichenkettenebene in den zu übersetzenden Quelltext einsetzt.
Dieses Beispiel zeigt sehr gut die Stärken und Schwächen des CPP. Auf der einen Seite ist der CPP ein sehr mächtiges Instrument bei der Konfiguration einer Produktlinie. Auf der anderen Seite sinkt aber die Lesbarkeit des Quelltextes durch die Vermischung von Anwendungsquelltext und Konfiguration.

Nachteile des CPP

Trotz seiner enormen Popularität wird der Einsatz des CPP immer wieder stark kritisiert. SPENCER und COLLYER [SC92] diskutierten eine Reihe von Problemen des CPP. Angelehnt hieran folgt eine aktuelle Diskussion dieser Thematik nach KÄSTNER und APEL [KA09], die im Wesentlichen drei Schwächen des Ansatzes benennen:

Trennung von Belangen: Eine zusammenhängende Repräsentation des merkmalsrelevanten Quelltextes in einem Modul findet bei der Annotation mit dem Präprozessor nicht statt. Stattdessen werden merkmalsrelevante Programmteile über die gesamte Codebasis verstreut. Dies verhindert unter anderem ein leichtes Verständnis von Merkmalen, das durch eine zusammenhängende Repräsentation besser gewährleistet wird.

Fehleranfälligkeit: Der CPP arbeitet auf Zeichenketten bzw. Token. Es erfolgt keine Interpretation des zugrundeliegenden Quelltextes. Somit sind nicht einmal einfache Syntaxfehler im Basisquelltext ausgeschlossen. Diese Fehler würden erst nach einer Konfiguration einer konkreten Variante, die diesen Quelltext enthält, auftreten.

```
1  void
2  __os_clock(dbenv, secsp, usecsp)
3    DB_ENV *dbenv;
4    u_int32_t *secsp, *usecsp;  /* Seconds and microseconds. */
5  { const char *sc;
6    int ret;
7  #if defined(HAVE_GETTIMEOFDAY)
8    struct timeval tp;
9    RETRY_CHK((gettimeofday(&tp, NULL)), ret);
10   if (ret != 0) {
11     sc = "gettimeofday";
12     goto err;
13   }
14   if (secsp != NULL)
15     *secsp = (u_int32_t)tp.tv_sec;
16   if (usecsp != NULL)
17     *usecsp = (u_int32_t)tp.tv_usec;
18 #endif
19 #if !defined(HAVE_GETTIMEOFDAY) && defined(HAVE_CLOCK_GETTIME)
20   struct timespec tp;
21   RETRY_CHK((clock_gettime(CLOCK_REALTIME, &tp)), ret);
22   if (ret != 0) {
23     sc = "clock_gettime";
24     goto err;
25   }
26   if (secsp != NULL)
27     *secsp = tp.tv_sec;
28   if (usecsp != NULL)
29     *usecsp = tp.tv_nsec / 1000;
30 #endif
31 #if !defined(HAVE_GETTIMEOFDAY) && !defined(HAVE_CLOCK_GETTIME)
32   time_t now;
33   RETRY_CHK((time(&now) == (time_t)-1 ? 1 : 0), ret);
34   if (ret != 0) {
35     sc = "time";
36     goto err;
37   }
38   if (secsp != NULL)
39     *secsp = now;
40   if (usecsp != NULL)
41     *usecsp = 0;
42 #endif
43   return;
44 ... // error handling
45 }
```

Abbildung 3.3: Beispiel für die Verwendung des CPP in BerkeleyDB

```
1
2  /EXTERN:/ {
3    sub("^.*EXTERN:[␣␣␣][␣␣]*", "")
4    if ($0 ~ "^#if|^#ifdef|^#ifndef|^#else|^#endif") {
5      print $0 >> e_pfile
6      print $0 >> e_dfile $0
7      next
8    }
9    eline = sprintf("%s␣%s", eline, $0)
10   if (eline ~ "\\)\\);") {
11     sub("^[␣␣]*", "", eline)
12     print eline >> e_pfile
13     if (eline !~ db_version_unique_name && eline !~ "^int␣txn_") {
14       gsub("[␣␣][␣␣]*__P.*", "", eline)
15       sub("^.*[␣␣][*]*", "", eline)
16       printf("#define␣%s␣%s@DB_VERSION_UNIQUE_NAME@\n",
17           eline, eline) >> e_dfile
18     }
19     eline = ""
20   }
21 }
```

Abbildung 3.4: Quelltextmanipulation auf Zeichenkettenebene in BerkeleyDB

Eine Konfiguration aller Varianten und die Syntaxprüfung aller Varianten ist nicht möglich ohne weitere Werkzeugunterstützung [KA09].

Quelltextverständlichkeit durch Vermischung: Bei der Verwendung einer Präprozessorsprache wird die Konfigurationssprache mit der Programmiersprache des Basisprogramms vermischt. Dies erschwert die Lesbarkeit des Quelltextes und damit die Verständlichkeit der Anwendungslogik. Resultat sind höhere Kosten für die Wartung und Weiterentwicklung der Anwendung.

Diskussion

Die drei Schwächen des CPP führen häufiger dazu, dass Entwickler bewusst auch auf statische Konfigurationen im Quelltext verzichten. Dies ist insbesondere dann der Fall, wenn neben der Lesbarkeit des annotierten Quelltextes auch Abhängigkeiten zwischen Merkmalen existieren. Diese Abhängigkeiten führen zu verschachtelten Präprozessoranweisungen. Bei Abhängigkeiten mehrerer Merkmale führt dies oft zu komplexen Verschachtelungen von Präprozessoranweisungen, die deutlich die Verständlichkeit reduzieren. Eine pragmatische Lösung in der Praxis ist, dass lediglich die Konfiguration auf vollständige Dateien vorgenommen wird. Feingranulare Vermischungen bleiben im Quelltext und werden zur Laufzeit dynamisch abgeprüft. Das hat messbare Auswirkungen auf die Laufzeit und führt zu einer Vergrößerung des binären Programmcodes.

In Abschnitt 7.3 werden genauere Untersuchungen des Leistungsverhaltens und die resultierenden Vorteile bei reiner statischer Konfiguration aufgezeigt.

Einweiterungen des CPP

Aktuelle Arbeiten von KÄSTNER [Käs10, KATS11] zeigen wie einige dieser Nachteile beseitigt oder durch toolbasierte Ansätze gemildet werden können. Des Weiteren wird in Arbeiten zu *TypeChef* [KKHL10, KGR+] der CPP um ein produktlinienorientiertes Typsystem erweitert, um den CPP besser für den Produktlinieneinsatz zu gestalten.

3.4 Objektorientierte Programmierung

Objektorientierte Programmierung

Die zentralen Abstraktionselemente im Design und in der Implementierung der Objektorientierten Programmierung sind *Objekte* und deren Beschreibung mit Hilfe von Klassen. Klassen enthalten die Definitionen von Datenvariablen und Methoden, die den Zustand des Objektes repräsentieren und diesen manipulieren können. Des Weiteren ist es durch Methoden möglich, die Steuerung von Interaktionen mit anderen Objekten zu implementieren.

Vererbung und Polymorphismus

Wichtige Kernkonzepte der Objektorientierten Programmierung bezüglich der Wiederverwendung sind *Vererbung* und *Polymorphismus*. Die Vererbung gestattet die Wiederverwendung von bereits existierenden Objektbeschreibungen (Basisklasse). Polymorphie ermöglicht, dass Objekte, die aus einer abgeleiteten Klasse instanziiert wurden, auch grundsätzlich in der Rolle als Objekte der Basisklasse verwendet werden können. Somit ist das Objekt über verschiedene Schnittstellen nutzbar.

Erweiterungen der Objektorientierten Programmierung

Entwurfsmuster

Entwurfsmuster (engl. *design pattern*) bieten *Musterlösungen* für wiederkehrende Probleme in der Objektorientierten Softwareentwicklung [GHJV96]. Ziel dieser Entwurfsmuster ist die Entwicklung von variablen, wiederverwendbaren und erweiterbaren Softwaresystemen. Mit Vorgaben für Vererbungs-, Aggregations- und Assoziationsbeziehungen zwischen Klassen werden Lösungsstrategien vorgeschlagen, die Systeme flexibler gestalten. Aber auch Entwurfsmuster können die Grenzen der Objektorientierten Programmierung bezüglich querschneidender Belange nicht überwinden. Ein weiteres Problem von Entwurfsmustern ist die zusätzliche Komplexität, die durch die Einführung variabler Mechanismen dem System hinzugefügt wird.

Generische Programmierung

Unter der *Generischen Programmierung* verstehen CZARNECKI und EISENECKER [CE00] S. 167 ff. ganz allgemein die Parametrisierung von Komponenten. Diese kann durch Template-Parameter erfolgen. Abbildung 3.5 zeigt die Verwendung eines Template-Parameters. Die Funktion `T GetMax(T a, T b)` (Zeile 2) trennt den Algorithmus von dem konkreten Typ der Implementierung mit Hilfe einer zusätzlichen Typschnittstelle.

```
1  template <class T>
2    T GetMax (T a, T b) {
3        T result;
4        result = (a>b)? a : b;
5      return (result);
6  }
7  int main () {
8
9     int result_1 = GetMax<int>(i,j);
10    long result_2  = GetMax<long>(l,m);
11
12   return 0;
13 }
```

Abbildung 3.5: Generische Programmierung

Eine Diskussion über die Verwendung der Generischen Programmierung findet im Abschnitt 5.2.2 statt.

Komponenten

Der Begriff der Komponente ist in der Informatik sehr überladen. CZARNECKI und EISENECKER [CE00], S. 9 sehen Komponenten recht einfach als Bausteine eines Softwaresystems. Eine spezifischere Definition liefert SZYPERSKI [Szy98]:

> *„A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."* SZYPERSKI [Szy98], S. 41

Glue-Code

Die Basis für die Wiederverwendung von Komponenten bildet eine Komponentenarchitektur, die im Idealfall auf den Variationspunkten einer Produktlinie aufbaut. Komponenten kommunizieren mit dem Rest des Softwaresystems nur über verbindliche Schnittstellen. Ein konkretes Produkt wird in der Regel aus den ausgewählten Komponenten von einem Entwickler mit „*Glue-Code*" zusammengesetzt. Der Komponentenorientierte Ansatz fokussiert nicht auf die automatische Zusammensetzung des Produktes. Für sehr grobgranulare Komponenten funktioniert der Ansatz sehr gut. Je feiner die Granularität der Komponenten ist, desto mehr Aufwand muss beim Zusammenführen der Kompo-

nenten betrieben werden. Dies führt zum Einsatz von mehr „*Glue-Code*“. Des Weiteren treten bei sehr kleinen Komponenten verstärkt Probleme durch querschneidende Belange auf.

Frameworks

Der *Framework-Ansatz* von JOHNSON und FOOTE [JF88] stellt, anders als reine Komponentenansätze, eine feststehende Basis an Funktionalität bereit. In diese Basis kann neue Funktionalität hinzugefügt werden. Hierfür bietet ein Framework Erweiterungspunkte (engl. *extension points*), an denen die möglichst gut gekapselten Erweiterungen (idealerweise Komponenten) „andocken“. Hierzu bieten sich in der Objektorientierten Programmierung nach JOHNSON und FOOTE [JF88] abstrakte Klassen an.

3.5 Aspektorientierte Programmierung

Aspektorientierte Programmierung

Eine weitere Möglichkeit, die Objektorientierte Programmierung zu verbessern, bietet die *Aspektorientierte Programmierung (AOP)*. Die Erweiterung von KICZALES et al. [KLM+97] ermöglicht zusätzlich zu einer Basisdekomposition die Trennung von querschneidenden Belangen, die nicht durch die Basisdekomposition getrennt werden können. In der ursprünglichen Idee bezogen sich KICZALES et al. [KLM+97] auf die Objektorientierte Programmierung. Die im folgenden Abschnitt vorgenommenen Betrachtungen[6] zu AOP beziehen sich ebenfalls auf die Objektorientierte Programmierung.
Die Aspektorientierte Programmierung erweitert die Objektorientierte Programmierung um Aspekte, die neben Klassen ein weiteres wichtiges Abstraktionselement darstellen. Nach einer Definition von KICZALES et al. [KLM+97] wird ein Aspekt wie folgt definiert:

> *„Aspects tend not to be units of the systems functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways.“* KICZALES et al. [KLM+97] S. 223

Eine aspektorientierte Dekomposition eines Programms erfolgt in zwei Dimensionen. Im ersten Schritt wird mit Hilfe des objektorientierten Entwurfs eine Dekomposition des Programms in Klassen vorgenommen. Im zweiten Schritt werden Belange, die sich nicht in einzelne Klassen zerlegen lassen, in Aspekte gekapselt. Ein *Aspektweber* führt diese getrennten Belange an zuvor definierten Stellen wieder zusammen (vgl. Abbildung 3.6). Hierzu werden vier grundlegende Konzepte benötigt:

6 Des Weiteren beziehen sich die Betrachtungen nur auf AspectJ-ähnliche Sprachkonstrukte.

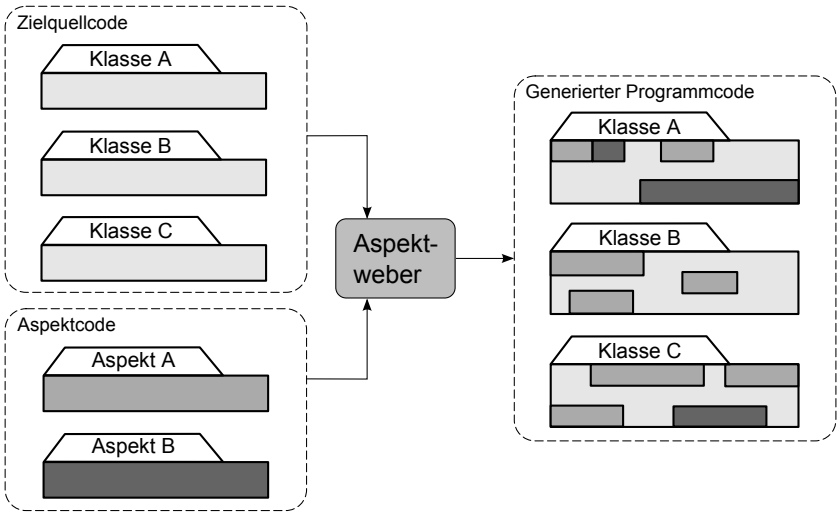

Abbildung 3.6: Aspektweber

Join Point: Ein *Join Point* beschreibt eine mögliche Stelle[7] im Programmcode, an der Code eines Belanges durch einen Aspektweber eingewebt werden soll.

Pointcut: Ein *Pointcut* beschreibt eine Menge von *Join Points*. AspectJ-ähnliche Sprachen verwenden hierzu einen deklarativen Ausdruck (engl. *pointcut expression*). Durch generische Platzhalter und logische Verknüpfungen können *Pointcut*-Ausdrücke eine Menge von einzelnen *Join Points* kapseln.

Advice: Im *Advice* wird der eigentliche Quelltext gekapselt, der ausgeführt werden soll, wenn im Programmcode ein *Join Point* erreicht wird. Dabei gibt es die Möglichkeiten entweder den *Advice Code* vor der Ausführung des *Join Points*, danach oder um den *Join Point* herum auszuführen. Ein Ersetzen des Ausführens am *Join Points* ist auch möglich. Somit bildet der *Advice* den eigentlichen Container für den Quelltext des querschneidenden Belanges.

Introduction beziehungsweise Intertypdeklarationen: Mit Hilfe von *Introduction beziehungsweise Intertypdeklarationen* können statische Anweisungen zu Klassen, Interfaces oder Aspekten hinzugefügt werden. Durch diese Anweisungen wird nicht direkt in den Kontrollfluss des Programms eingegriffen, sondern es können neue Methoden oder Attribute hinzugefügt werden. Des Weiteren bieten *Introductions* die Möglichkeit, die Vererbungshierarchie einer Klasse zu manipulieren.

7 Aus diesem Grund werden *Join Points* auch als *Join Point Shadows* bezeichnet, aus denen erst *Join Points* werden, wenn auch laufzeitabhängige Bedingungen erfüllt sind.

Abbildung 3.7 zeigt den Quelltext eines einfachen `Logging`-Aspekts in AspectC++ [SGSP02] Notation. Durch den Aspekt wird jede Manipulation durch die Methode `setValue()` der Klasse `MemoryManager` (Zeile 10) gezählt (Zeile 13), nachdem diese ausgeführt wurde (Zeile 11).

```
1  aspect logging
2  {
3    int counter;
4    // constructor
5    logging()
6    {
7      counter = 0;
8    }
9    //on execution of setValue()
10   pointcut log() = execution("void MemoryManager::setValue(...)");
11   advice log() : after()
12   {
13     ++counter;
14   }
15 };
```

Abbildung 3.7: Beispiel eines Aspektes in AspectC++

Neben AspectC++ für C++ (SPINCZYK et al. [SGSP02]) gibt es unter anderem auch AspectJ für Java (LOPES et al. [LK98]) und AspectS für Smalltalk (HIRSCHFELD [Hir02]). Neben der Möglichkeit der sauberen Trennung von Belangen einer Software, die an sich eine bessere Wiederverwendung einzelner Softwarebestandteile ermöglicht, eignen sich Aspekte auch, um optionale beziehungsweise variable Bestandteile einer Software hinzuzufügen.

3.6 Merkmalsorientierte Softwareentwicklung

Definition

Die *Merkmalsorientierte Softwareentwicklung* (engl. *Feature-Oriented Software Development* (FOSD)) ist eine Weiterentwicklung der Objektorientierten Softwareentwicklung und wird im Allgemeinem auch als Merkmalsorientierte Programmierung (engl. *Feature-Oriented Programming* (FOP)) bezeichnet. Der Begriff der Merkmalsorientierten Programmierung wurde in diesem Zusammenhang erstmals von PREHOFER [Pre97] geprägt. Ziel der Merkmalsorientierten Softwareentwicklung nach APEL und KÄSTNER [AK09] ist es:

> *„... den Entwurf, die Anpassung und die Synthese von großen Softwaresystemen"*
> APEL und KÄSTNER [AK09], S.49 (übersetzt)

zu unterstützen. Dazu soll möglichst ein 1:1 Mapping zwischen den Merkmalen im Entwurf und der Implementierung realisiert werden. Abbildung 3.8 zeigt ein solches Mapping, bei dem die Merkmale des merkmalsorientierten Entwurfs auf Implementierungseinheiten gemappt werden. In den folgenden Abschnitten werden die Grundlagen der Merkmalsorientierten Softwareentwicklung präsentiert.

3.6.1 GenVoca

GenVoca

Eine wichtige Grundlage der Merkmalsorientierten Softwareentwicklung bilden die Arbeiten an *GenVoca* von BATORY et al. [BO92]. Mit Hilfe von *GenVoca* kann das Zusammensetzen von Programmen innerhalb einer Programmfamilie beschrieben und validiert werden. Die Basisidee von *GenVoca* ist eine hierarchische Dekomposition des Softwaresystems in einer Schichtenarchitektur auf Grundlage der schrittweisen Verfeinerung nach DIJKRA und PARNAS [Dij76, Par76, Par78]. Jede Schicht stellt einer anderen Schicht ihre Dienste über eine definierte Schnittstelle bereit beziehungsweise nutzt diese Dienste. *GenVoca* bietet hierzu eine Grammatik, mit deren Hilfe die Kompatibilität einzelner Schichten geprüft werden kann. Eine Menge von Schichten mit identischer Schnittstelle wird als *Realm* bezeichnet. Diese Realms sind im Rahmen der Grammatik beliebig substituierbar. Im folgenden Beispiel sind drei *Realms* X, Y, und Z dargestellt, die jeweils unterschiedlich implementierte Schichten enthalten:

$$X = \{a, b, c\} \tag{3.1}$$

$$Y = \{s[X], e[X]\} \tag{3.2}$$

$$Z = \{n[Z], m[Y], p, q[Y, X]\} \tag{3.3}$$

Schnittstellen und Realms

In den eckigen Klammern werden die Schnittstellen angegeben, die von dieser Schicht benötigt werden. Dabei ist es auch möglich, dass eine Schicht mehrere Schnittstellen benötigt (hier $q[Y, X]$). Schichten, die die gleiche Schnittstelle eingangs- und ausgangsseitig bereitstellen (hier $n[Z]$), werden als symmetrische Schichten bezeichnet. Diese Komponenten können in beliebiger Reihenfolge und beliebig häufig an den passenden Stellen eingeführt werden. Im Folgenden sind drei Beispielprogramme angegeben, die bezüglich der drei vorangegangenen Realms X, Y und Z syntaktisch korrekt sind:

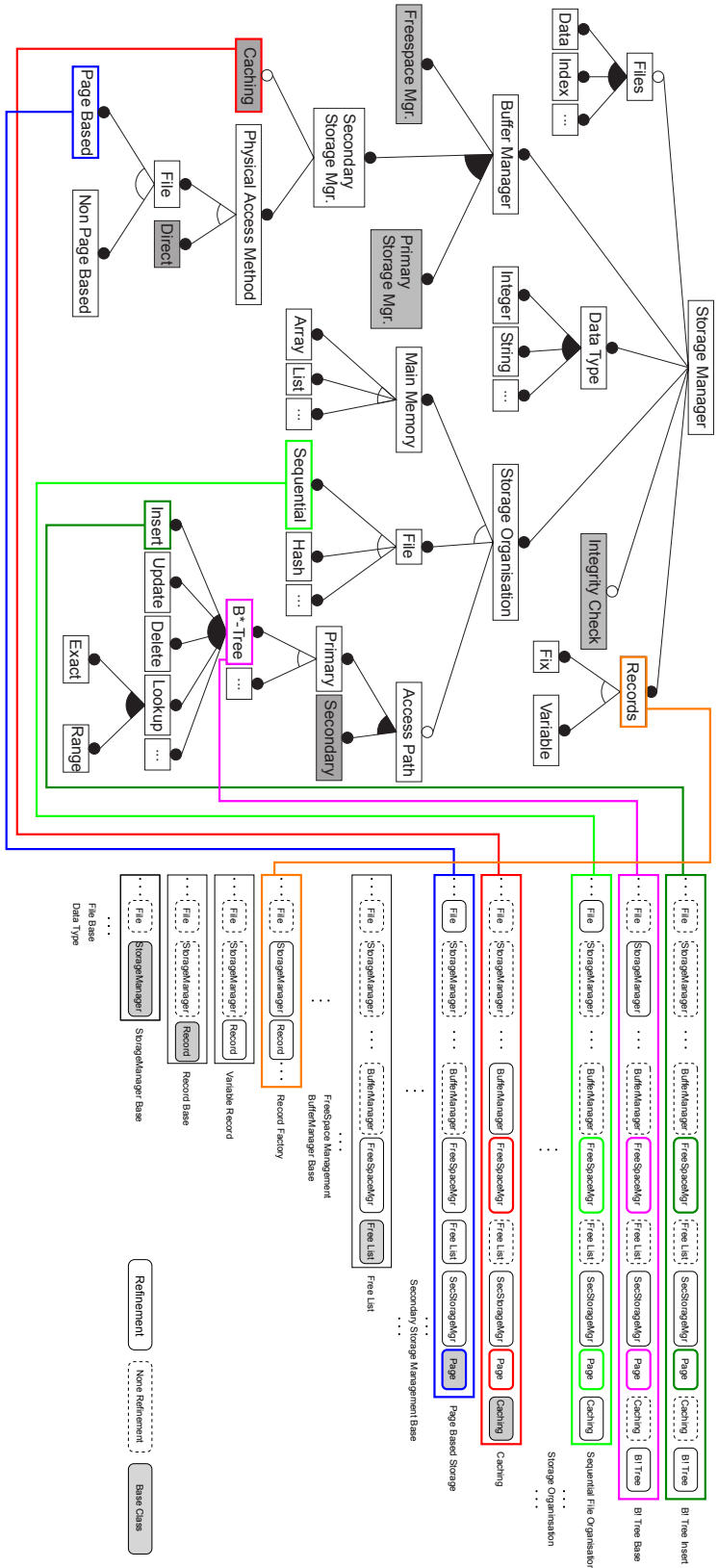

Abbildung 3.8: Mapping von Merkmalsmodell zu Entwurf und Implementierung

$$Programm_1 = p \tag{3.4}$$
$$Programm_2 = n[p] \tag{3.5}$$
$$Programm_3 = n[m[e[a]]]\} \tag{3.6}$$

Aus den *Realms* und deren Schichten lässt sich die folgende *GenVoca*-Grammatik ableiten:

$$X := a|b|c; \tag{3.7}$$
$$Y := sX|eX; \tag{3.8}$$
$$Z := nZ|mY|p|qYX; \tag{3.9}$$

Das Hinzufügen von neuen Schichten oder *Realms* zur Grammatik erweitert die Sprache und dadurch auch die Anzahl der Sätze, die durch die Sprache gebildet werden können. Auf diese Weise lassen sich Programmfamilien und ihre Variationen durch die *GenVoca*-Repräsentation sehr gut darstellen. In ihrer Auslegung stellt die *GenVoca*-Grammatik auf formalen Wege sicher, dass ausschließlich kompatible Schichten kombiniert werden können. Hierzu wird die syntaktische Korrektheit von Sätzen der Grammatik geprüft. Konzepte einer semantischen Prüfung stellt *GenVoca* jedoch nicht zur Verfügung, sodass syntaktisch korrekte Kombinationen von Schichten, die semantisch jedoch nicht sinnvoll sind, nicht erkannt werden.

3.6.2 Kollaborationenentwürfe

Kollaborationen

Eine weitere wichtige Grundlage der Merkmalsorientierten Softwareentwicklung bilden objektorientierte Kollaborationenentwürfe. Als Erweiterung zum objektorientierten Entwurf, in dem Klassen und Objekte sowie Daten und deren Funktionen die zentralen Abstraktionselemente sind, werden im Kollaborationenbasierten Entwurf Kollaborationen von Objekten als weiteres zentrales Abstraktionselement hinzugenommen. Die Idee des Kollaborationenbasierten Entwurfs [RWL95, Van97] beruht auf der Erkenntnis, dass eine Klasse selten die Funktionalität des ganzen Moduls/Merkmals oder einer Komponente implementiert. Stattdessen benötigt nach SMARAGDAKIS und BATORY[SB98] eine Erweiterung mehrere Klassen:

> *„Viewed in terms of design modularity, collaboration-based design acknowledges that a unit of functionality (module) is neither a whole object nor a part of it, but can cross-cut several different objects."* SMARAGDAKIS und BATORY[SB98], S. 551

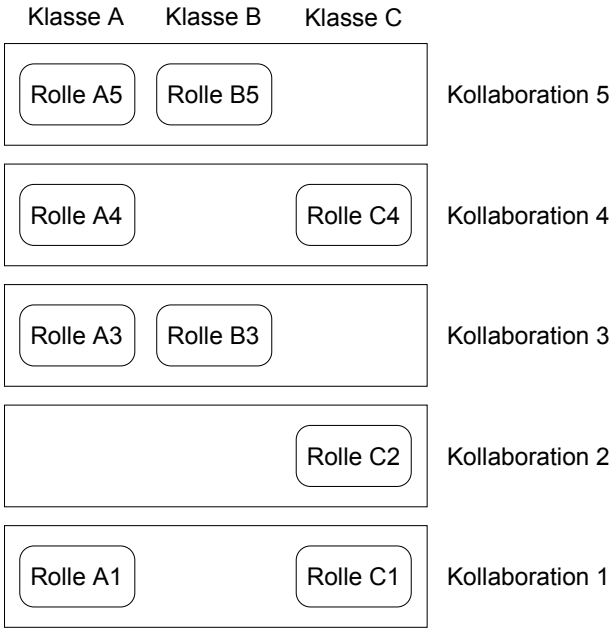

Abbildung 3.9: Darstellung eines Kollaborationsentwurfs

Eine Kollaboration besteht aus einer Menge von Klassen und einem Protokoll, das festlegt, wie diese Klassen interagieren. Abbildung 3.9 zeigt eine Hierarchie von Kollaborationen, die drei Klassen schneidet. Die folgenden Abschnitte beschreiben mögliche Umsetzungsansätze in objektorientierten Kollaborationen.

3.6.3 Mixin Layer

Verschachtelte Klasse und Parametrisierte Vererbung

Mixin Layer nutzen zwei Konzepte von objektorientierten Programmiersprachen aus, um Funktionalität analog zum Kollaborationenentwurf umzusetzen. Zum einen ermöglichen die meisten objektorientierten Programmiersprachen auch das Schachteln von Klassen. Durch die Verwendung von verschachtelten Klassen können Klassen übersichtlich zu größeren Einheiten zusammengefasst werden (vgl. hierzu Abbildung 3.10). Dabei wird in innere (Zeile 3) und äußere Klassen (Zeile 1) unterschieden. Darüber hinaus gibt

```
1 class Outerclass {
2
3   class Innerclass {... };
4 };
```

Abbildung 3.10: Verschachtelte Klassen

es die Möglichkeit, wenn zwei äußere Klassen (Zeile 1 und 11) in einer Vererbungsbezie-

hung stehen, dass auch die inneren Klassen (Zeile 13 und 16) von den anderen inneren Klassen der Vaterklasse (Zeile 3 und 6) erben können (vgl. hierzu Abbildung 3.11).

```
1  class OuterBaseClass {
2
3    class InnerClass1 {
4      ....
5    };
6    class InnerClass2 {
7      ...
8    };
9  };
10
11 class OuterDerivedClass:  public OuterBaseClass {
12
13   class InnerClass1 : public OuterBaseClass::InnerClass1{
14     ...
15   };
16   class InnerClass2 : public OuterBaseClass::InnerClass2{
17     ...
18   };
19 };
```

Abbildung 3.11: Gekapselte Vererbung von inneren Klassen

Mixin

Zum anderen gibt es in vielen objektorientierten Programmiersprachen die Möglichkeit, *abstrakte Subklassen* zu definieren [BC90]. BRACHA und COOK [BC90] bezeichnen dies als *Mixins*. Sie ermöglichen, dass ein *Mixin* mit verschiedenen Superklassen instantiiert wird. VANHILST und NOTKIN [VN96] zeigten als erste die Umsetzung eines *Mixins* mit Hilfe des Template-Mechanisums von C++ (vgl. hierzu Abbildung 3.12). Dies wird auch als anonyme beziehungsweise parametrisierte Vererbung bezeichnet.

```
1 template <class SUPER>
2 class mixin : public SUPER {
3 ..
4 };
```

Abbildung 3.12: Umsetzung eines *Mixins* mittels C++

Mixin Layer

SMARAGDAKIS und BATORY [SB98] zeigen, wie durch die Kombination von Klassenverschachtelungen und *Mixins* Kollaborationenentwürfe umgesetzt werden können. In Abschnitt 5.2.1 werden eine Umsetzung an einem Beispiel und die daraus resultierenden Probleme diskutiert.

3.6.4 AHEAD

AHEAD

Algebraic Hierarchical Equations for Application Design (AHEAD) ist ein integriertes Entwurfs- und Umsetzungswerkzeug auf der Basis eines algebraischen Modells, das die Merkmalsorientierte Softwareentwicklung in Java ermöglicht. AHEAD wurde von BATORY et al. [BSR03, Bat04] vorgestellt und stellt eine Weiterentwicklung des *GenVoca*-Ansatzes dar. Die Grundidee besteht darin, dass nicht nur der Quelltext schrittweise verfeinert werden kann, sondern dies auf alle „Artefakte"(z. B. UML-Diagramme, Test-Cases, Prozessmodelle, Makefiles usw.) einer Software ausgeweitet werden sollte [BLHM02, Bat04]. Des Weiteren können die Schichten nicht mehr nur hierarchisch angeordnet werden, sondern auch selbst aus feineren Schichten zusammengesetzt sein.

AHEAD-Notation

Die AHEAD-Notation ist eine Erweiterung des *GenVoca*-Ansatzes, bei der Basisartefakte durch Konstanten und die Verfeinerungen der Artefakte durch Funktionen repräsentiert werden. Hierzu führen BATORY et al. [BSR03] einen expliziten Verfeinerungsoperator $\bullet$ und den Begriff des *Kollektivs* (heute Merkmal oder Merkmalsmodul [ALS08] genannt) ein. Ein Merkmalsmodul kapselt alle Artefakte, die zu diesem Merkmal gehören, und kann wie folgt ausgedrückt werden [BSR03, Bat04]:

$$Basis : h = a_h, b_h, c_h; \tag{3.10}$$

$$Verfeinerung : f = a_f, b_f, d_f; \tag{3.11}$$

Durch die Komposition von h und f mit Hilfe des eingeführten Verfeinerungsoperators $h \bullet f$ erweitert die Verfeinerung f mit Hilfe von Funktionen die Basisartefakte a_h und b_h. Des Weiteren führt f das neue Basisartefakt d_f ein. Durch eine rekursive Komposition ergibt sich:

$$h \bullet f = \{a_h, b_h, c_h\} \bullet \{a_f, b_f, d_f\} \tag{3.12}$$

$$= \{a_h \bullet a_f, b_h \bullet b_f, c_h, d_f\} \tag{3.13}$$

Die konkrete Implementierung des Verfeinerungsoperators $\bullet$ wird durch den Artefakttyp bestimmt. Die Grundidee von BATORY et al. [BSR03, Bat04] ist es, mit Hilfe dieser Algebra, ähnlich zur relationalen Algebra, ein mathematisches Grundgerüst für eine automatische Optimierung vor der Ausführung der Komposition zu liefern.

AHEAD-Tool-Suite

Die AHEAD-Tool-Suite bietet eine Sammlung von Werkzeugen, mit deren Hilfe die Merkmalsorientierte Softwareentwicklung mit dem Fokus auf Java-Anwendungen betrieben werden kann. Die Kapselung der Artefakte in ein Merkmalsmodul wird durch

die Ordnerstruktur des Betriebssystems realisiert. Das zentrale Werkzeug der AHEAD-Tool-Suite ist die Java-Erweiterung *Jak*. Durch die Einführung des Schlüsselwortes `refines` kann die Verfeinerung von Klassen vorgenommen werden. Abbildung 3.13 zeigt zunächst die Basis der Klasse `StackOfChar`. Die erste Verfeinerung `Layer Size` zeigt, wie der Stack um Überwachungsfunktionalität erweitert wurde (hier durch einen einfachen Zähler angedeutet). Die zweite Verfeinerung `Layer Lock` fügt dem Stack einen einfachen Synchronisationsmechanismus hinzu (hier durch Sperren). Für den Zugriff auf die zu verfeinernde Klasse wird das Schlüsselwort `Super()` verwendet.

```
\\ Layer Base
class StackOfChar {

String s = new
String();

void empty() {
  s = "";
 }

void push( char a ) {
  s = a + s;
 }

void pop() {
  s = substring(1);
 }

char top() {
  return s.charAt(0);
 }
}
```

```
\\ Layer Size
refines class StackOfChar {

int ctr = 0;

int size() { return ctr; }
void reset() {ctr = 0; }
void inc() { ctr++; }
void dec() { ctr--; }
void empty() {
 reset();
  Super().empty();
}
void push( char a ) {
 inc();
 Super().push( a );
}
void pop() {
 dec();
 Super().pop();
}
}
```

```
\\ Layer Lock
refines class StackOfChar {
boolean lck = false;
void lock() { lck = true; }
void unlock() {
 lck = false;
}
void empty() {
 if ( !lck )
Super().empty();
}
void push( char a ) {
 if ( !lck )
Super().push(a);
}
void pop() {
 if ( !lck ) Super().pop();
}
void inc()   {...}
void dec()   {...}
void reset() {...}
}
```

Abbildung 3.13: Implementierung eines *Stacks* durch Verfeinerungen

Interne Umsetzung durch Jampack oder Mixin

AHEAD bietet zwei Transformationsmöglichkeiten des *Jak*-Quelltextes in Standard Java. Zum einen kann die Verfeinerungshierarchie in eine klassische Vererbungshierarchie umgewandelt werden. Dies wird als *Mixin* bezeichnet. Dabei wird jede Konstante und jede Verfeinerung einer Klasse in eine abstrakte Klasse umgewandelt, wobei jeder Klassenname dann aus dem Namen der Klasse und dem Merkmalsmodulnamen zusammensetzt wird. Die letzte Verfeinerung erhält dann den originalen Basisklassennamen. Zum Ändern kann in AHEAD ein sogenanntes *Jampack* erzeugt werden, bei dem die Basis und alle Verfeinerungen in einer Klasse zusammengeführt werden. Durch eine entsprechende Transformation der Memberbezeichner wird der Quelltext in einer Klasse

zusammengefasst. Semantisch betrachtet liefern beide Transformationen das gleiche Ergebnis, aber in der Praxis bietet die *Mixin*-Komposition Vorteile beim Debugging, sodass BATORY et al. [Bat04] die *Mixin*-Implementierung favorisieren.

Design Rule Checks

Bei genauer Betrachtung des Beispiels in Abbildung 3.13 fällt auf, dass die Reihenfolge der Komposition der Merkmale von großer Bedeutung sein kann. Während die Komposition *Base* • *Size* • *Lock* korrekt ist, ist die Reihenfolge *Base* • *Lock* • *Size* problematisch, da die Synchronisation des Zählers nicht gewährleistet ist. Hierfür bietet BATORY et al. [Bat04] *Design Rule Checks*. Mit Hilfe einer attributiven Grammatik können zusätzliche Beschränkungen wie Verfeinerungsreihenfolgen oder weitere Bedingungen definiert werden, die vor der Komposition überprüft werden müssen. Somit können zuvor definierte „semantische Fehler“ bei der Komposition ausgeschlossen werden. BATORY et al. [Bat04] bezeichnet dies als semantische Prüfung.

3.7 Zusammenfassung

In diesem Kapitel haben wir einen Überblick zu den Begriffen des Software-Engineering im Umfeld von Variabilität geben. Hierzu wurden zunächst Definitionen der Grundbegriffe Modularisierung, Trennung von Belangen und Komposition präsentiert. Im Weiteren Verlauf haben wir einen Einblick in den Bereich der Produktlinien und mögliche Umsetzungen gegeben. Hierbei zeigten wir im letzten Abschnitt die Merkmalsorientierte Programmierung, die die Basis für unsere Forschungsarbeit darstellt.

KAPITEL 4

Explorative Studie: FAMEDBMS-Speichermanager

*Dieses Kapitel enthält Ergebnisse der Veröffentlichung ADBIS 2005: „Using Step-Wise Refinement to build a Flexible Lightweight Storage Manager“ (*LEICH, APEL, SAAKE *[LAS05]) sowie der Veröffentlichungen ICSE 2006 „Aspectual Mixin Layers: Aspects and Features in Concert“(*APEL, LEICH, SAAKE *[ALS06]) und IEEE-TSE 2008 „Aspectual Feature Modules“ (*APEL, LEICH, SAAKE *[ALS08]). Implementierungsarbeiten wurden von* BÄRECKE *„Merkmalsorientierte Programmierung in C++“ [Ros05] im Rahmen einer zu dieser Dissertation betreuten Abschlussarbeit vorgenommen.*

In diesem Kapitel präsentieren wir eine explorative Studie, die den Einsatz der Merkmalsorientierten Softwareentwicklung im Umfeld des hochkonfigurierbaren Datenmanagements für eingebettete Systeme evaluiert. Der FAMEDBMS-Speichermanager (FameDBMS: engl. ***Fam**ily of **E**mbedded **D**ata**B**ase **M**anagement **S**ystems)* ist die erste Umsetzung des später DFG-geförderten FAMEDBMS Projektes und dient als Grundlage für die in den nachfolgenden Kapiteln vorgeschlagenen Lösungsansätze und Evaluierungen. Die Grundlage für die Evaluierung bildet eine Fallstudie mit einem idealisierten Szenario aus dem Bereich der Sensor-Netzwerke. Diese soll möglichst viele Störvariablen ausschließen und den explorativen Charakter dieses Kapitels stärken. Als Umsetzungsmethode verwenden wir die Merkmalsorientierte Softwareentwicklung unter Anwendung der AHEAD-Tool-Suite. Im Ergebnis wird eine Liste von offenen Forschungsfragen präsentiert, die in den folgenden Kapiteln bearbeitet werden.

4.1 Untersuchungsgegenstand

Gegenstand der explorativen Untersuchung

Gegenstand der nachfolgenden Untersuchung ist es, die Anwendbarkeit der Merkmalsorientierten Softwareentwicklung in der Domäne des Datenmanagements für eingebettete Systeme im Grundsatz zu evaluieren. Hierzu wird ein variabler Speichermanager auf Basis der AHEAD-Tool-Suite merkmalsorientiert implementiert. Auf der Grundlage dieser Variationsbasis können dann verschiedene Varianten maßgeschneidert werden. Dabei klammern wir eine Implementierung auf einem konkreten eingebetteten System für diese erste Analyse aus. Dadurch sollen technische Probleme wie zum Beispiel Compiler-Restriktionen, Versionskonflikte, Speicher- und Performanzrestriktionen sowie weitere Hardware-Einschränkungen vermieden werden. Des Weiteren gilt es, das Paradigma möglichst in seiner grundlegenden Funktionsweise in dieser Domäne zu evaluieren. Die Untersuchung fokussiert ausgehend von Abschnitt 2.6 dabei auf die folgenden sieben Problemfelder, die wir im Hinblick auf eine Umsetzung des variablen Nanodatenmanagements für wichtig erachten:

Granularität: Auf Grund der Ressourcenbeschränkungen im Bereich der eingebetteten Systeme müssen die Erweiterungen sehr feingranular gestaltet werden. Gegenstand der Untersuchung ist es, die schrittweise Verfeinerung mittels Merkmalsorientierter Softwareentwicklung umzusetzen und zu überprüfen, ob der Kompositionsmechanismus auf Basis von Rollen der Objekte die notwendigen feingranularen Erweiterungen ermöglicht.

Overhead: Derzeitig werden moderne Programmierparadigmen für die Entwicklung im Bereich der eingebetteten Systeme oft mit der allgemeinen Begründung abgelehnt, dass diese neuen Techniken zuviel Overhead im Bereich der Laufzeitperformanz und des Speicherverbrauches erzeugen. In diesem Zusammenhang soll die Fallstudie erste Anhaltspunkte ermitteln, die eventuellen Einfluss auf die Performanz haben oder auf den Speicherverbrauch schließen lassen.

Wiederverwendbarkeit: Als ein möglicher Grund für die mangelnde Wiederverwendung von Datenmanagementfunktionalität wird häufig die Komplexität der Abhängigkeiten verschiedener Funktionen und Merkmale untereinander genannt. Hieraus ergibt sich die Frage, inwieweit diese Abhängigkeiten isoliert beziehungsweise kontrolliert werden und inwiefern sich mit Hilfe der Merkmalsorientierten Programmierung eine bessere Wiederverwendung dieser Funktionalitäten und Merkmale erreichen lässt.

Querschneidende Belange: Die feingranulare Dekomposition eines Systems führt häufig dazu, dass sich nicht alle Belange entlang der gewählten Dekompositionsstrategie trennen lassen. Die Merkmalsorientierte Programmierung verspricht hier eine verbesserte Trennung von Belangen. Es gilt daher zu untersuchen, ob trotz der zusätzlich eingeführten Dekompositionsdimension in der Merkmalsorientierten Programmierung weiterhin querschneidende Belange auftreten, die nicht durch die Dekompositionsstruktur gekapselt werden können.

Automatisierbarkeit der Konfiguration: Auf Grund der feingranularen Zerlegung in viele Merkmale kann eine Produktlinie bezüglich der Abhängigkeiten der Merkmale untereinander sehr schnell komplex werden. Deshalb ist eine möglichst gute Unterstützung des Konfigurationsprozesses und eine automatische Generierung von verschiedenen Varianten ohne zusätzlichen Programmieraufwand notwendig. Es gilt in der Fallstudie zu untersuchen, ob der Ansatz der Merkmalsorientierten Programmierung unter Verwendung der AHEAD-Tool-Suite die notwendigen Unterstützungen für die Automatisierung bietet.

Praktikabilität: Die Praktikabilität eines Entwicklungsansatzes ist ein entscheidender Erfolgsfaktor für die Akzeptanz und die damit verbundene Etablierung dieses Ansatzes. Hierzu zählen wir Faktoren wie den Einarbeitungs- und Lernaufwand in das Paradigma und die zur Verfügung stehende, domänenspezifische Werkzeugunterstützung, deren Nutzen klar zu erkennen ist.

Umsetzung von ungeplanten Änderungen: Der Produktlinienansatz ist grundsätzlich auf einen längeren Anwendungszeitraum ausgerichtet, bei dem verschiedene Produkte aus den Produktlinien über einen längeren Zeitraum abgeleitet werden sollen. Trotz sorgfältiger Planungen bei der Entwicklung einer Produktlinie sind Änderungen in deren Struktur oder Variationspunkten nicht zu vermeiden. In diesem Zusammenhang gilt es herauszufinden, wie tolerant die Merkmalsorientierte Softwareentwicklung gegenüber ungeplanten Änderungen ist.

In den folgenden Abschnitten dieses Kapitels wollen wir an einer Fallstudie untersuchen, inwiefern die Merkmalsorientierte Programmierung mit diesen Problemfeldern umgehen kann.

4.2 Fallstudienszenario Sensornetzwerke

Um den explorativen Charakter dieser Studie zu unterstützen, wird im folgenden Abschnitt ein idealisiertes Szenario aus dem Bereich Datenhaltung in Sensornetzwerken

vorgestellt. Eine vollständige Domänenanalyse und Umsetzung dieser Funktionalität wird in diesem Szenario nicht angestrebt.

Überblick Szenario

Im Bereich der drahtlosen Sensornetzwerke konzentrierte sich die Forschung der letzten Jahre vor allem auf die Anfragebearbeitungen in Netzwerken, kontinuierliche Anfragen, die Verarbeitung von Datenströmen und (Vor-)Aggregationsalgorithmen [GGP+03, ABB+04, BW01, CcC+02, BBD+04, GÖ03]. Durch eine zentralisierte Datenhaltung und Analyse auf wenigen leistungsstarken Rechnern sollen preiswerte Sensoren mit geringem Ressourcenbedarf zum Einsatz kommen. Viele dieser Sensornetzwerke sind für den Langzeitbetrieb in unzugänglichen Bereichen entwickelt worden und somit auf Grund der knappen Ressourcen in der Kommunikation stark eingeschränkt [GGP+03].

Datenmanagement im Kontext von Ressourcenbeschränkungen

Abbildung 4.1 zeigt ein solches Szenario. Die Versorgung mit Strom oder der Austausch der Energiequellen ist meist sehr aufwändig oder überhaupt nicht möglich. Eine ständige Kommunikation und Übermittlung der Daten scheidet somit aus. Voraggregationen, Filterungen oder Kompressionen von Rohdaten auf dem Sensorknoten selbst sollen helfen, den Speicherverbrauch zu minimieren und die Kommunikationsbeschränkungen zu überwinden. Dies ist nur möglich, wenn die zu untersuchenden Merkmale a priori bekannt sind. Gerade in wissenschaftlichen Anwendungsbereichen wie der Biotopüberwachung ist dieses nicht immer möglich, da sich viele Fragestellungen erst während der Überwachung ergeben [GGP+03]. Eine Lösung dieses Problems ist die Zwischenspeicherung von Rohdaten auf dem Sensor selbst und die Implementierung von Funktionalität, die eine Ad-hoc Vorverarbeitung beziehungsweise eine Kompression bei Bedarf ermöglicht.

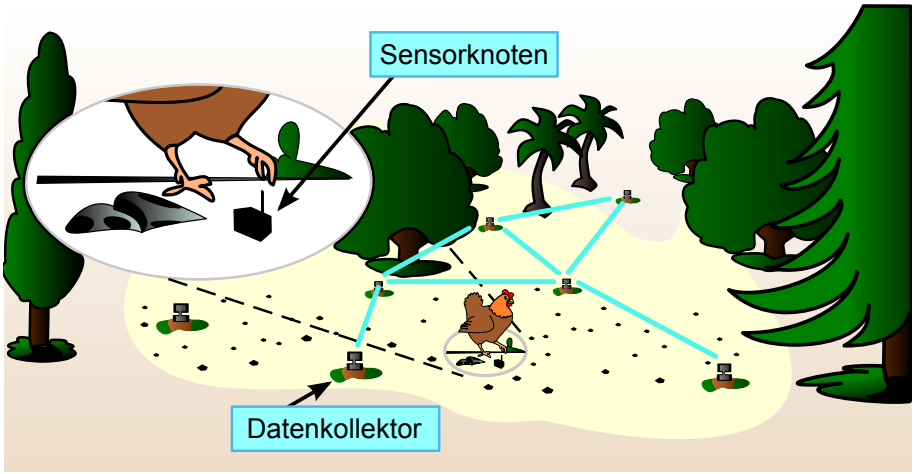

Abbildung 4.1: Sensorüberwachung eines Feuchtbiotops [Cha12]

Datenmanagementeigenschaften am Beispiel einer Biotopüberwachung

In Anlehnung an WOO et al. [WMG04, GGP+03] wird im Folgenden ein typischer Versuchsaufbau von Biologen zur Überwachung eines Biotops vorgestellt. In diesem Szenario werden die Knoten des Sensornetzwerkes in zwei Kategorien unterschieden:

Sensorknoten: Einfache Sensoren messen und speichern Umweltdaten wie zum Beispiel Temperatur, Luftfeuchtigkeit, Vibrationen, chemische Zusammensetzungen oder Lichtintensität. Bedingt durch die Spezifikation des Messverfahrens werden verschiedene Datentypen benötigt. Die jeweilige „Record"-Länge ist für einen Sensorknotentyp fest. Des Weiteren benötigen die Sensoren eine einfache Speicherstruktur mit Einfüge- und Suchfunktionalität. Eine Löschfunktion ist zum Beispiel nicht notwendig, da die im Hauptspeicher gehaltenen Daten periodisch mit einem „Reset" zurücksetzt werden.

Datenkollektoren: Datenkollektoren bilden die zweite Gerätegruppe, die die Daten verschiedener Sensoren je nach Bedarf der Wissenschaftler „zusammentragen" und Informationen für interne und externe Analysen bereitstellen. Für eine sichere und dauerhafte Speicherung wird eine Sekundärspeicherung verwendet, die durch eine Pufferverwaltung optimiert wird. Die Speicherstruktur muss für die jeweilige Analyse geeignet sein und entsprechende Operationen wie beispielsweise Gruppierungen oder Sortierungen anbieten. Des Weiteren werden Integritätschecks benötigt, um Ausfälle oder Fehlmessungen von Sensoren zu registrieren.

Im Folgenden wollen wir für diese zwei Knotentypen verschiedene Varianten von Datenmanagementfunktionalität aus einer Produktlinie ableiten.

4.3 Umsetzung des Speichermanagers

Im folgenden Abschnitt werden die Domänenanalyse sowie das Design und die Implementierung der variabel gestalteten Speicherverwaltung vorgestellt. Wir beschränken uns dabei auf die wesentlichen Aspekte, die für unsere explorative Analyse in dem nachfolgenden Kapitel relevant sind.

4.3.1 Ergebnis der Domänenanalyse

Ergebnis der Merkmals-analyse

Die Abbildung 4.2 zeigt als Ergebnis der merkmalsorientierten Domänenanalyse einen Ausschnitt der Variationspunkte unserer Speicherverwaltung. Die grau unterlegten

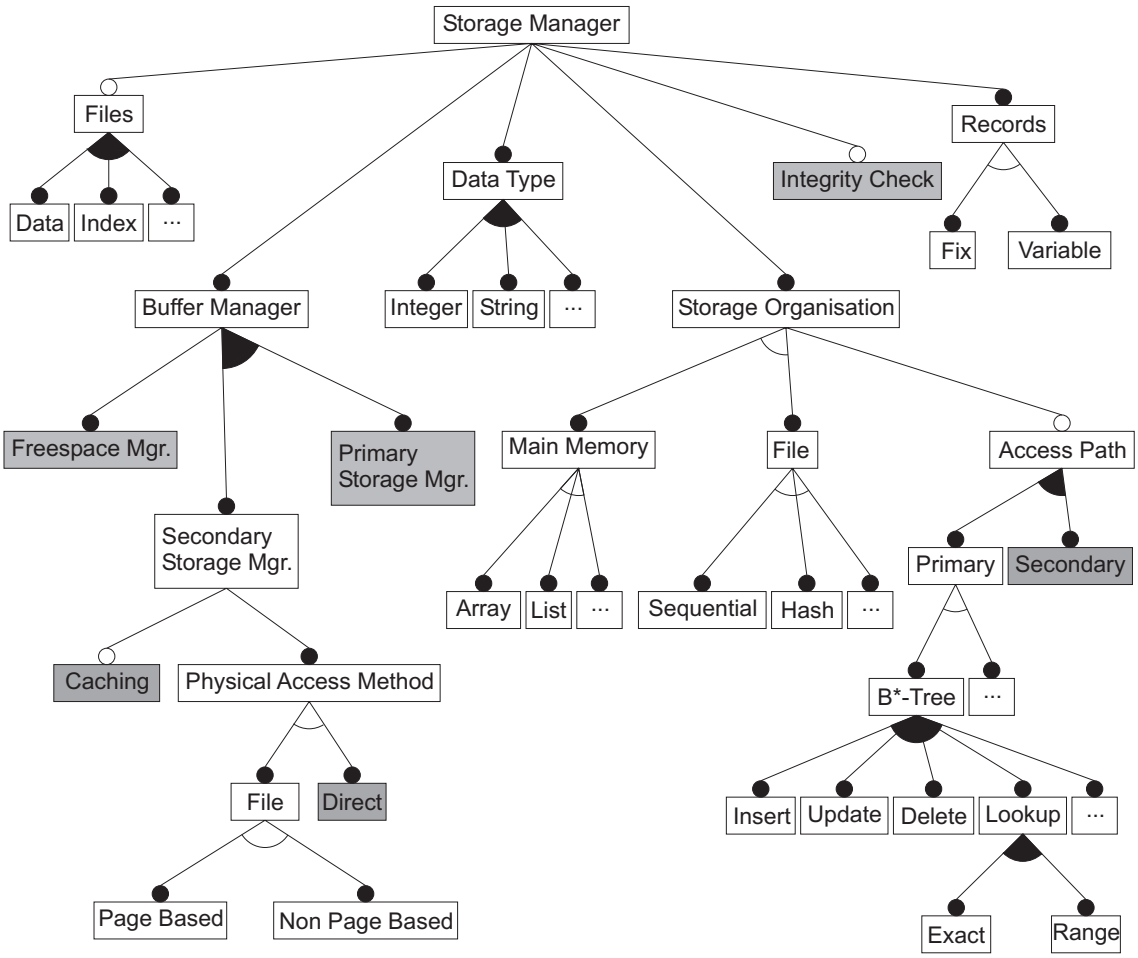

Abbildung 4.2: Ausschnitt aus dem Merkmalsmodell des Speichermanagers

Kästchen repräsentieren Merkmale, für die keine untergeordneten Merkmale aus Vereinfachungsgründen dargestellt wurden. Unsere Speicherverwaltung besteht aus vier erforderlichen Merkmalen, die wiederum variabel gestaltet werden können:

Data Type (DT): Das Merkmal *Data Type* repräsentiert alle verfügbaren Datentypen des Speichermanagers. Somit ist es möglich, eine Speicherverwaltung zu generieren, die nur Integer verarbeiten kann.

Buffer Manager (BM): Der *Buffer Manager* verwaltet den Primär- und Sekundärspeicher und organisiert den zur Verfügung stehenden freien Speicherplatz. Wird eine Datenmanagementvariante mit einem Sekundärspeicher konfiguriert, kann optional hierzu noch ein *Caching* aktiviert werden. Des Weiteren können die Daten entweder in einem *File* oder direkt auf den Hintergrundspeicher (Sekundärspeicher) geschrieben werden.

Storage Organisation (SO): Die Speicherverwaltung (*Storage Organisation*) ist für die Strukturierung und den Zugriff auf Daten verantwortlich. Hierzu kann zwischen einer reinen Hauptspeicherverwaltung und einer dateibasierten Speicherung der Daten unterschieden werden. Zusätzlich können Zugriffspfade gewählt werden.

Records (Rec): *Records (Rec)* repräsentieren die Daten in unserem Speichermanager. Diese können mit einer fixen oder variablen Länge definiert sein.

Optionale Merkmale

Weiterhin ist es optional möglich, den Speichermanager mit einer Integritätsprüfung (*Integrity Checks (IC)*) auszustatten und die unterstützten Dateitypen zu erweitern (*File (FT)*).

Nicht analysierte Merkmale

Das komplette Merkmalsmodell in unserer sehr stark beschränkten Speicherverwaltungsfamilie hat 93 Merkmale. Spezielle Datentypen, eine Transaktionsverwaltung, Komponenten zur Wiederherstellung oder spezialisierte Indexstrukturen für stark restriktive Anwendungsszenarien wie etwa BOBINEAU et al. [BBPV00] in PicoDBMS haben wir dabei nicht weiter betrachtet. Eine erweiterte Analyse würde hunderte zusätzliche Merkmale für eine Speicherverwaltung im beschriebenen Anwendungskontext hervorbringen.

Anzahl der Varianten

Für eine grobe Bestimmung der Anzahl an möglichen Varianten müssen wir noch die nicht aufgeführten Merkmale quantifizieren. In der Tabelle 4.1 sind variable Parameter dargestellt, zum Beispiel die Anzahl der unterstützten Datentypen. Die dritte Spalte gibt die Werte für unsere experimentelle Evaluierung an. Zum Beispiel haben wir für die Berechnung der erlaubten Varianten der Speicherverwaltung vier unterschiedliche Datentypen angenommen. Für die Bestimmung der Varianten haben wir folgende Formel aus dem Merkmalsdiagramm abgeleitet (unter Nutzung einer GenVoca-Grammatik [BO92]):

$$\#SM = \underbrace{(2^f - 1)}_{FT} * \underbrace{(15)}_{BM} * \underbrace{(2^d - 1)}_{DT} * \underbrace{((m + s) * (6 * a * (2^o - 1)))}_{SO} * \underbrace{(2)}_{IC} * \underbrace{(2)}_{Rec}$$

Bei der Berechnung der theoretischen Anzahl der Varianten haben wir 8.164.800 mögliche Konfigurationen bestimmt.

Parameter	Beschreibung	Anzahl angenommer Varianten
d	data types	4
f	file type	2
m	main memory organisation	2
s	data file structures	2
a	access structure	2
o	B^* Tree	6

Tabelle 4.1: Parameter für die Berechnung des Variantenraums unseres Speichermanagers

4.3.2 Entwurf, Implementierung und Konfiguration

Entwurf

Um unseren Ansatz zu evaluieren, haben wir eine Speicherverwaltung mit Hilfe der AHEAD-Tool-Suite implementiert und auf Basis von Java umgesetzt. Im ersten Schritt wurde hierzu das Merkmalsdiagramm auf Schichten des objektorientierten Entwurfs überführt. Abbildung 4.3 zeigt einen Ausschnitt dieses Kollaborationenentwurfs. Die Schichten, die die B-Baum Zugriffsstruktur betreffen, sind von unten nach oben dargestellt. Angefangen bei den unteren Schichten, in denen die Merkmale Datensätze (*Record*), Seitenspeicher (*Page*) und der *Cache* den Basisspeichermanager erweitern, werden in den oberen Schichten die B-Baum-Struktur und verschiedene Operationen auf dem Baum hinzugefügt. (Abbildung 4.3 stellt lediglich die Einfüge-Operation im B-Baum dar). Die grau hinterlegten Klassen werden in der entsprechenden Schicht neu eingefügt. Die verfeinerten Klassen mit durchgezogenen Umrandungen erweitern in der jeweiligen Schicht diese Klasse. Die Klassen mit den gestrichelten Linien werden in der jeweiligen Schicht nicht erweitert.

Implementierung

Für unseren Speichermanager wurden 26 Klassen und einige Hilfsklassen implementiert. Im Durchschnitt haben wir drei Klassen pro Schicht verfeinert. Um beispielsweise den *seitenbasierten Speicher* zu implementieren, mussten wir drei Klassen verfeinern (*File, FreeSpaceMgr, SecStorageMgr*) und eine neue Klasse hinzufügen (*Page*).

Konfiguration

Die Variantengenerierung wird durch die AHEAD-Tool-Suite einfach gestaltet und gut unterstützt. Zur Demonstration des einfachen Konfigurationsvorgangs und der Flexibilität der Implementierung haben wir verschiedene Speicherverwaltungen für unser Szenario abgeleitet:

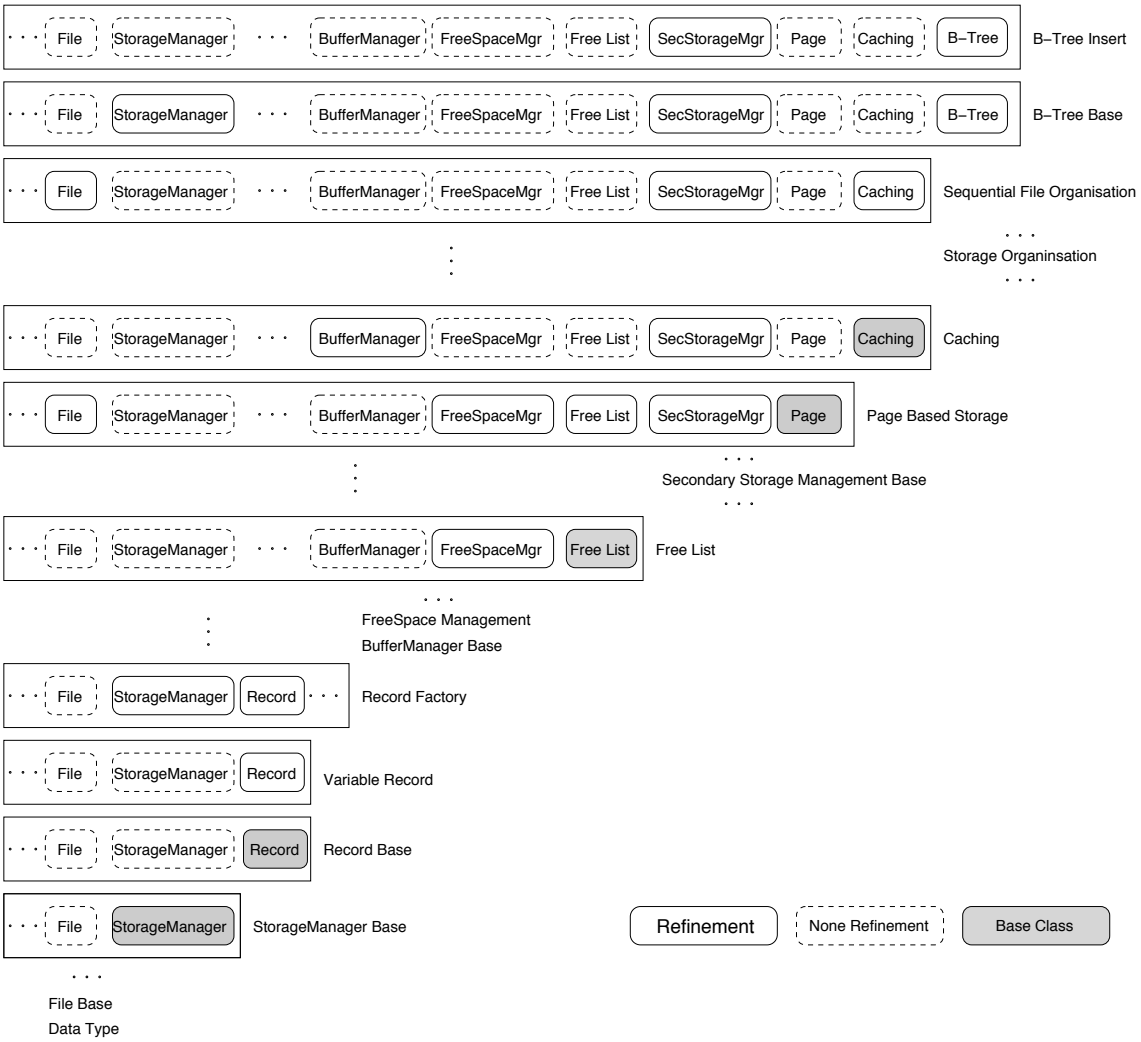

Abbildung 4.3: Ausschnitt des Kollaborationendiagramms

Sensorknoten: Wir haben zwei verschiedene Speicherverwaltungsversionen aus der Speichermanagerproduktlinie für unsere Sensorknoten generieren können. Beide Konfigurationen verwenden eine Hauptspeicherverwaltung. Die Hauptspeicherbelegung erfolgt statisch auf Grund der fest definierten Datensatzlänge, die durch den Sensortyp definiert wird. Der erste Sensortyp verwendet nur einfache Datentypen (*integer, number*) und ein einfaches Feld, um Daten zu speichern. Die resultierende Speicherverwaltung ist durch 11 Schichten schrittweise verfeinert. Für die zweite Kategorie haben wir eine *Hash-Map* anstelle des Feldes konfiguriert. Dadurch werden die Änderungs- und Suchfunktionen effizient unterstützt. Des Weiteren haben wir einen Integritätscheck für die Datensätze hinzugefügt. Diese

zweite Variante der Speicherverwaltung ergibt sich durch Verfeinerung von 16 Schichten beziehungsweise Merkmalen.

Datenkollektor: Auf Grund des Anwendungsszenarios ist die Funktionalität des Datenkollektors komplexer. Szenarienbedingt sind die Datenkollektoren mit einem Sekundärspeicher ausgestattet. Hierfür haben wir eine Sekundärspeicherverwaltung konfiguriert, die eine dateiorientierte Speicherung erlaubt und eine interne seitenbasierte Organisation nutzt. Die Daten in den Dateien sind sequenziell organisiert und als Zugriffspfad haben wir einen B^*-Baum verwendet. Zur Verbesserung der Leistung ist eine Cache-Verwaltung hinzugewählt worden. Die Datensätze können eine variable Länge aufweisen. Auf Grund der Aufgaben des Datenkollektors haben wir eine spezielle Integritätsprüfung für die Datensätze eingebaut. Der voll konfigurierte Datenkollektor wird aus 38 Erweiterungen gebildet.

Generierung der Varianten

Die korrekte syntaktische Zusammensetzung der Schichten für eine bestimmte Konfiguration wird durch eine einfache Schichtenauswahl vorgenommen. Eine zusätzliche semantische Korrektheit wird durch *Design Rule Checks* der AHEAD-Tool-Suite gewährleistet. Wir haben in unserer Fallstudie herausgefunden, dass etwa 60 Prozent aller möglichen Konfigurationen durch zusätzliche *Design Rule Checks* ausgeschlossen werden. Viele dieser *Design Rule Checks* , die den Merkmalsbaum zusätzlich begrenzen, sind erst auf Entwurfs- und Implementierungsebene hinzugekommen.

4.4 Evaluierung und Ergebnisdiskussion

Im folgenden Abschnitt werden die Ergebnisse der merkmalsbasierten Umsetzung eines hochkonfigurierbaren Speichermanagers vorgestellt. Die Ergebnisse basieren auf einer Umsetzung mittels der AHEAD-Tool-Suite. Eine Verallgemeinerung dieser ersten Ergebnisse ist nicht Ziel dieses Kapitels. Vielmehr steht eine Problemdefinition im Vordergrund. Hierfür werden die aufgetretenen Probleme in den Implementierungsphasen diskutiert. Anschließend wird aufbauend auf dieser Analyse das weitere Vorgehen aufgezeigt. Die Abbildung der Variationspunkte durch die Merkmalsorientierte Programmierung mittels der AHEAD-Tool-Suite war für unser ausgewähltes Fallbeispiel sehr gut möglich. Unsere Implementierung hat gezeigt, dass die Zerlegung der Speicherverwaltung in feingranulare Komponenten möglich ist. Im Folgenden diskutieren wir einzelne Aspekte der Umsetzung.

4.4.1 Granularität

Feingranulare Maßschneiderung

Bezüglich der Variationspunkte ist uns im Vergleich zu existierenden Lösungen wie beispielsweise Berkeley DB oder COMET DBMS eine sehr feingranulare Zerlegung des Speichermanagers gelungen. Dies ermöglicht eine bessere Adaption unseres Speichermanagers an den Anwendungskontext. Die gesamte Analyse hat für das beschriebene Szenario 93 Variationspunkte identifiziert. So kann beispielsweise der Funktionsumfang des B-Baumes vergleichsweise sehr variabel gestaltet werden. Hier half vor allem die zusätzliche Dimension der Kapselung der Merkmale und das Schneiden der Objekte, um bisherige Grenzen aufzulösen.

Einschränkungen

Generelle Aussagen lassen sich hieraus dennoch nicht ableiten. Insbesondere die als schwer zu kapselnde Funktionalität der Transaktionsverwaltung, die sowohl in Berkeley DB als auch in COMET DBMS implementiert wurde, ist in unserem Speichermanager nicht realisiert. Um hier wissenschaftlich belastbare Aussagen treffen zu können, muss auch diese Funktionalität integriert werden. Des Weiteren gibt es auch feingranulare Erweiterungen, die beispielsweise Erweiterungen von Strukturen unterhalb der Methodenebene bedürfen, wie etwa die Erweiterung einer Schleife innerhalb einer Methode mit zusätzlicher Funktionalität. Diese Art von Erweiterungen sind konzeptbedingt nicht möglich.

4.4.2 Overhead

Overhead-Analyse nicht möglich

Für eine belastbare Aussage zum *Overhead* der hier vorgestellten Untersuchung müssten umfangreiche Performanz- und Speicherverbrauchsanalysen durchgeführt werden. Diese Analyse müsste für fundierte Aussagen auf gängiger eingebetteter Hardware durchgeführt werden. Ein direkter Vergleich zu den auf eine Minimierung des *Overheads* optimierten Versionen von Berkeley DB[1] und COMET DBMS ist schon auf Grund der verwendeten Basissprachen nicht möglich. Während Berkeley DB und COMET DBMS auf Basis der Programmiersprache C entwickelt wurden, so ist unser Speichermanager in Java konzipiert.

1 Wir beziehen uns hier auf die C-Version von Berkeley DB.

Konzeptbedingt kaum Overhead-Potential zu erkennen

Konzeptbedingt ist der Java-Ansatz hier durch die Laufzeitumgebung benachteiligt. Des Weiteren ist gerade im Bereich von Datenmanagementsoftware für eingebettete Systeme die Diversität des Funktionsumfangs so stark, dass Vergleiche schwer möglich sind. Darüber hinaus sind unser Speichermanager und COMET DBMS im Gegensatz zu Berkeley DB nicht mit einer praxistauglichen Fehlerbehandlung und Protokollierungsfunktionalitäten ausgestattet. Auch wenn eine direkte Betrachtung des *Overheads* für diese Fallstudie nicht möglich ist, so wollen wir an dieser Stelle die gewonnenen Erkenntnisse auf Konzeptebene vergleichen. Generell bietet die Merkmalsorientierte Programmierung mit AHEAD-Ansatz in der gewählten Kompositionsform „*Jampack*“ keinen konzeptbedingten *Overhead* zur Objektorientierten Programmierung, da keine Indirektionen und virtuellen Funktionen benutzt werden. Vielmehr können im Gegensatz zur klassischen objektorientierten Umsetzung Erweiterungen von Objekten durch eine statische Komposition durchgeführt werden. Im Kontrast zu *Plug-In* Konzepten wird auch keine zusätzliche *Plug-In* Architektur benötigt. Für das Zusammenfügen von Merkmalen wird des Weiteren auch kein zusätzlicher „*Glue-Code*“ wie beispielsweise bei der klassischen Komponentenkomposition verlangt.

4.4.3 Querschneidende Belange

Bessere Trennung von Belangen

Die zusätzliche Möglichkeit, eine objektorientierte Struktur in Merkmale zu zerlegen, schafft einen deutlich besseren Umgang mit den querschneidenden Belangen, sodass generell von einer besseren Trennung der Belange in unserer Fallstudie gesprochen werden kann. Im Durchschnitt mussten drei Klassen pro Merkmal erweitert werden, das zeigt, dass im Durchschnitt jedes Merkmal querschneidend zu drei Klassen steht. Dennoch kann eine generelle Aussage noch nicht getroffen werden, da die als kritisch geltenden Funktionalitäten wie Transaktionsverwaltung mit Mehrbenutzersynchronisation und *Recovery*, *Logging* und Fehlerbehandlung in dieser idealisierten Studie noch nicht betrachtet wurden.

Im Zusammenhang der Modularisierung einer schichtweisen Erweiterung der Merkmalsorientierten Programmierung eines Basissystems zeigte sich, dass die bisherige in Abschnitt 3.1.2 vorgenommene Differenzierung in „Scattered“ und „Tangled Code“ nicht ausreicht. Im Folgenden werden wir deshalb eine differenzierte Betrachtung vornehmen. Die Basis dieser Analyse bilden verschiedene Überlegungen zur Kategorisierung von querschneidenden Belangen von LIEBERHERR et al. [LLO03] , LOPEZ-HERREJON et al. [LHBC05] als auch von MEZINI und OSTERMANN [MO04a]. Die problematischen Erweiterungen und die in unserer Fallstudie aufgetretenen querschneidenden Belange

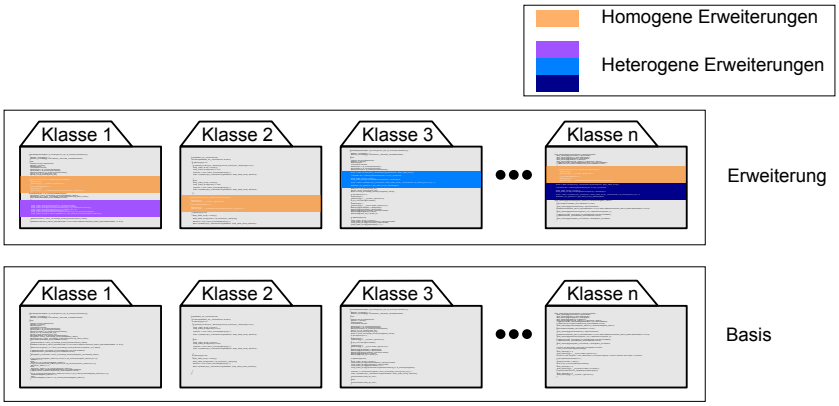

Abbildung 4.4: Homogene versus heterogene Erweiterungen

und deren Struktur können in die folgenden vier Kategorien eingeteilt werden:

Homogene und heterogene Erweiterungen: Als homogene Erweiterungen bezeichnen wir Quelltexterweiterungen, die an verschiedenen Stellen das Basisprogramm mit gleichen Quelltextfragmenten erweitern. Dagegen führen heterogene Erweiterungen an verschiedenen Stellen im Basisprogramm unterschiedlichen Quelltext ein (vgl. hierzu COLYER und CLEMENT, [CC04]). Abbildung 4.4 zeigt beide Erweiterungstypen am Beispiel. Während heterogene Erweiterungen ideal durch die Kapselung in Merkmale mit Hilfe der Merkmalsorientierten Programmierung umgesetzt werden können, gibt es bei homogenen Erweiterungen des Basisprogramms[2] Probleme. Zwar kann ein Merkmal auch den Quelltext der homogenen Erweiterungen kapseln, aber der Quelltext der Erweiterung liegt in diesem Fall dann redundant vor. Somit bietet die Merkmalsorientierte Programmierung für diese Art der Erweiterung kein adäquates Implementierungskonzept, um replizierten Quelltext zu verhindern. Homogene Erweiterungen sind in unserem idealisierten Szenario im Gegensatz zu einer Berkeley DB-Studie von TESANOVIC et. al [TSH04] und der COMET DBMS- Umsetzung [NTN+04] nicht aufgetreten. Dies liegt zum einen an dem Basisquelltext von Berkeley DB, der durch eine Präprozessoranweisung nicht physisch separiert wurde und zum anderen daran, dass wir die Funktionalität, an der TESANOVIC et al. [TSH04, NTN+04] diese Beobachtungen machten, beispielsweise Fehlerbehandlungen, Transaktionsverwaltung und Synchronisation, nicht in unserem Szenario implementierten.

2 Als Basisprogramm bezeichnen wir die Zusammenfassung aller Schichten, die durch diese Erweiterung um neue Funktionalität verfeinert werden sollen.

Statische und dynamische Erweiterungen: Eine statische Erweiterung fügt der festen Struktur eines Basisprogramms neue Elemente hinzu, indem beispielsweise neue Funktionen, Klassen, Superklassen oder Klassenvariablen die Basisstruktur des Programms erweitern. Dynamische Erweiterungen müssen differenzierter betrachtet werden. Hier lässt sich zwischen einfachen und komplexen dynamischen Erweiterungen unterscheiden. Einfache dynamische Erweiterungen fügen dem Kontrollfluss Funktionalität hinzu, indem sie den Kontrollfluss des Programms erweitern. Dies wird durch den Super-Impositionsansatz der Merkmalsorientierten Programmierung ermöglicht. Komplexere dynamische Erweiterungen können den Kontrollfluss in Abhängigkeit von vordefinierten Ereignissen und Aktionen erweitern (vgl. hierzu MEZINI und OSTERMANN [MO04a] WAND et al.[WKD04]). Diese Art der Erweiterung kann beispielsweise verwendet werden, wenn ein Merkmal, in Abhängigkeit von einem anderen optionalen Merkmal, Aktionen ausführen soll oder nicht. Auch wenn diese Art der Erweiterungen recht selten auftritt, so bietet die Merkmalsorientierte Programmierung keine geeigneten Konzepte hierfür an.

Merkmalskohäsion: Unter der hohen Merkmalskohäsion wird die Möglichkeit verstanden, die Implementierungsdetails dieses Merkmals in genau eine Implementierungseinheit mit entsprechendem Bezeichner zu kapseln. Im Idealfall wird eine eins zu eins Abbildung von abstrakten Merkmalen der Domänenanalyse zu der entsprechenden Implementierung angestrebt. Dies bedeutet, dass es eine explizite Repräsentation in der Design- und Implementierungsebene geben muss. Generell bietet die Merkmalsorientierte Programmierung konzeptbedingt eine sehr gute Möglichkeit dieser Abbildung. Dennoch zeigten sich bei der Implementierung des Speichermanagers Probleme, die eine eins zu eins Abbildung nicht ermöglichten. Wird beispielsweise ein optionales Merkmal durch ein anderes optionales Merkmal erweitert, so entsteht das Problem, dass der Interaktionsquelltext, den diese beiden optionalen Merkmale nur bei gleichzeitiger Konfiguration benötigen, beim Nichtvorhandensein eines dieser Merkmale einen Fehler produziert. Dieses Problem wird auch als „*feature optionality problem*“ in schichtbasierten Entwürfen[3] bezeichnet. Eine Lösung ist die Auslagerung dieses Interaktionsquelltextes in ein extra Merkmal. Während PREHOFER [Pre97] diese zusätzlichen (künstlichen) Merkmale/Schichten der Merkmalsinteraktionen als „*lifter*“ bezeichnet, so sprechen LIU et al. [LBN05, LBL06] in diesem Fall von „*derivatives*“ mit einer algebraischen Definition. Die Aufspaltung dieser Merkmale in künstliche Merkmale entspricht

3 Diese Betrachtung bezieht sich nur auf schichtenbasierte Implementierungen und ist nicht auf merkmalsorientierte Interaktionen auf semantischen Ebenen begrenzt.

so nicht mehr der eins zu eins Abbildung. Insbesondere bei einer starken „*Optionalisierung*“ von Merkmalen tritt dieses Problem häufiger auf. Eine umfassendere Diskussion, ein Beispiel und ein Lösungsvorschlag finden sich im Abschnitt 5.6.1.

4.4.4 Wiederverwendbarkeit

Die Wiederverwendbarkeit von Softwareeinheiten hängt von verschiedenen Faktoren ab. In Rahmen unserer Untersuchung fokussieren wir uns auf die Wiederverwendung von Merkmalen in verschiedenen Varianten eines Softwaresystems. Der Merkmalsorientierte Ansatz bietet die Möglichkeit, Einheiten der Wiederverwendung sehr feingranular zu definieren. Gerade die sehr gute Trennung von Belangen ermöglichte in unserer Fallstudie eine sehr gute Wiederverwendung von Teilen der umgesetzten Merkmale in einer anderen Programmvariante. Zusätzlich gibt es die Möglichkeit, auch die kleineren Merkmale in größere Einheiten zusammenzufassen. Dies wird durch Verschachtelung der Merkmale ermöglicht und erhöht somit eine Skalierung der wiederzuverwendenden Softwareeinheiten.

4.4.5 Umsetzung von ungeplanten Änderungen

Während der Entwicklung unseres variablen Speichermanagers zeigte sich, dass einige Erweiterungen, die nicht sorgfältig genug geplant waren beziehungsweise sich erst durch den Projektfortschritt ergaben, nur schwer in die Struktur der Kollaborationen integriert werden konnten. Dies ist insbesondere trotz einer sorgfältigen Planung in der Praxis im Produktlinienumfeld durch sich verändernde Marktbedingungen häufiger der Fall [PBL05]. In unserer vorgestellten Fallstudie führte dies zur kompletten Reorganisation und zur aufwändigen Neuimplementierung.

4.4.6 Automatisierbarkeit der Konfiguration

Die AHEAD-Tool-Suite bietet konzeptbedingt durch den Kompositionsansatz mittels Super-Imposition sehr gute Möglichkeiten zur Automatisierung des Konfigurationsprozesses. Je nach gewünschter Konfiguration konnten ohne weitere Implementierung die Varianten aus der Speichermanagerproduktlinie abgeleitet und auf Knopfdruck generiert werden. Wenn von vornherein alle erlaubten Varianten explizit modelliert werden, dann können auch die DRC Fehler bezüglich der Konfiguration automatisch erkennen. Für eine einfachere Konfiguration wäre dennoch eine Werkzeugunterstützung über den Funktionsumfang der AHEAD-Tool-Suite wünschenswert, die eine Merkmalsauswahl, je nach schon getroffener Auswahl, entsprechend intuitiv vereinfacht.

4.4.7 Praktikabilität

Ein Ziel dieser explorativen Studie war die Sammlung von Erfahrungen im Umgang mit der Merkmalsorientierten Programmierung in der Domäne der eingebetteten Systeme. Die AHEAD-Tool-Suite ist im akademischen Umfeld als Forschungsprototyp entstanden, sodass keine Studien über den praktischen Einsatz im größeren Produktlinienumfeld vorliegen. Während des Einsatzes der AHEAD-Tool-Suite im Rahmen unserer Fallstudie sind wir auf eine Vielzahl von Problemen auf Sprach- und Werkzeugebene gestoßen, die eine Etablierung dieses Ansatzes erschweren. Im Folgenden werden diese Probleme kurz vorgestellt:

Sprachebene: Eine Vielzahl von Herstellern eingebetteter Systeme liefert für die Hardware nur C bzw. C++ Compiler. Darüber hinaus sind sehr viele Bibliotheken nur für C/C++ vorhanden. Zwar gibt es auch vereinzelt Umgebungen für eingebettete Systeme, in denen eingeschränkt mit Java programmiert werden kann, doch dominieren derzeitig C/C++-Systeme die Praxis. Ein Grund ist das Bedürfnis der Entwickler, in diesem Umfeld, wenn nötig, möglichst hardware-nah programmieren zu können, um die geforderte Effizienz der Implementierung sicherzustellen. Des Weiteren werden im Umfeld von eingebetteten Systemen eine Vielzahl von weiteren, auf die Sprache von C/C++ abgestimmter Werkzeuge eingesetzt, wie beispielsweise automatisierte Testtools oder Werkzeuge zur Berechnung von maximalen Ausführungszeiten im Bereich der Echtzeitsysteme, die einen Einsatz des Java-basierten Ansatzes a la AHEAD-Tool-Suite auch in naher Zukunft unrealistisch erscheinen lassen.

Werkzeugebene: Die Handhabbarkeit der Merkmalsorientierten Programmierung ist für die Entwicklung komplexer Produktlinien ungeeignet. Auf Grund des experimentellen Entwicklungsstandes der AHEAD-Tool-Suite und der damit verbundenen sehr geringen Erfahrung der Entwickler, diese Werkzeuge mit großen Produktlinien zu nutzen, zeigt die Studie einen Bedarf an einer voll integrierten Entwicklungsumgebung (IDE). AHEAD bietet einige Werkzeuge, die die Merkmalsorientierte Softwareentwicklung unterstützen. Diese sind aber meist sehr einfacher Natur und nicht in moderne Entwicklungsumgebungen integriert. Zum einen dominieren in der AHEAD-Tool-Suite kommandozeilenorientierte Verarbeitungsschritte. Zum anderen fehlen Werkzeuge zum Debuggen, eine automatische Quellcodeergänzung, eine Verbindung zwischen Merkmalsmodell und Implementierung sowie eine Projektverwaltung, kontextabhängige Quelltextergänzungen

oder Visualisierungskonzepte für die Schichten und ihre jeweiligen Verfeinerungen.

4.5 Definition weiterer Untersuchungsgegenstände

Unsere explorativ angelegte Studie hat eine Vielzahl von offenen Punkten beim Einsatz der Merkmalsorientierten Programmierung für den Bereich der eingebetteten Systeme aufgezeigt. Darauf aufbauend definieren wir die weiteren Untersuchungsgegenstände für die Arbeit wie folgt:

FEATUREC++: Im ersten Schritt evaluieren wir die Möglichkeiten einer Integration der Merkmalsorientierten Programmierung in C++. C++ bietet schon heute verschiedene Möglichkeiten, Variabilität für hierarchische Dekompositionsformen bereitzustellen. Eine kurze Untersuchung zeigt die Einschränkung dieser Ansätze. Auf dieser Basis stellen wir unseren eigenen Ansatz FEATUREC++ für die Merkmalsorientierte Programmierung mittels C++ vor. Vorbild ist der Java-Sprachansatz *Jak* des AHEAD-Tool-Suite-Ansatzes. Neben der grundlegenden Ressourceneffizienz fokussieren wir auch auf eine bessere Integration von homogenen querschneidenden Belangen, eine verstärkte Robustheit gegen ungeplante Änderungen und eine bessere Umsetzungsmöglichkeit von optionalen Merkmalsinteraktionen.

Studien: Zur Evaluierung der Einsetzbarkeit von FEATUREC++ in der Domäne des Datenmanagements für eingebettete Systeme sollen in verschiedenen Studien unterschiedliche Aspekte untersucht werden. Hierzu werden die drei Fallstudien FAMEDBMS, ROBBYDBMS und Berkeley DB vorgestellt.

FAMEDBMS: Die erste Fallstudie soll auf Basis einer erweiterten Domänenanalyse FAMEDBMS mittels unseres Sprachansatzes FEATUREC++ neu entwickeln und evaluieren. Der Fokus dieser Entwicklung liegt auf einer wissenschaftlich getriebenen Domänenanalyse in der Domäne der eingebetteten Systeme und der Implementierung einer Referenzarchitektur. Auf die Maßschneiderung für konkrete Anwendungsfälle wird an dieser Stelle noch verzichtet. Während beim Speichermanager bisher auf die Integration von Transaktionsverwaltungs- und Anfrageoptimierungsmechanismen verzichtet wurde, sollen diese bisher nur schwer zu kapselnden Funktionalitäten integriert werden. Hieraus erhoffen wir uns Antworten über das Auftreten und die Integration von homogenen Erweiterungen.

ROBBYDBMS: In der zweiten Fallstudie zeigen wir die Umsetzung unserer Implementierung einer Produktlinie, die Datenmanagementfunktionalität für autonome, mobile Roboter auf der Basis der AVR-Hardware-Produktlinie von Atmel bereitstellt. Die Basis bildet die Referenzimplementierung von FAMEDBMS. Der Fokus dieser Studie liegt auf der stark eingeschränkten Hardware-Umgebung. Besonders diese Einschränkung soll die Fähigkeiten von FEATUREC++ beim Umgang mit stark eingeschränkten Ressourcen evaluieren. Auf Grund der sehr feingranularen Natur der Erweiterungen soll des Weiteren die untere Grenze des Einsatzbereiches von FEATUREC++ ermittelt werden.

Berkeley DB: Ein Problem der beiden vorangegangenen Studien ist die fehlende Vergleichbarkeit dieser Produktlinien zu bisherigen Ansätzen. Dies liegt am implementierten Funktionsumfang, der nicht zu existierenden Lösungen kompatibel ist. In dieser dritten Fallstudie soll nun anhand des in der Praxis etablierten Berkeley DB-Systems eine Vergleichbarkeit hergestellt werden. Hierzu wird zunächst in einem teilweise automatisierten Refaktorisierungsprozess der C-Quelltext der Originalversion in einen objektorientierten Quelltext überführt. In einem zweiten Schritt wird der Quelltext in merkmalsorientierten Quelltext auf Basis von FEATUREC++ überführt. Auf Grundlage eines vom Hersteller etablierten Benchmark werden dann die Originalversion und die Version auf Basis der Merkmalsorientierten Programmierung verglichen.

Werkzeugunterstützung: Ein weiteres wichtiges Ergebnis der explorativen Studie und unserer Erfahrungen aus den drei weiteren Studien zeigen, dass der AHEAD-Tool-Suite-Ansatz hinsichtlich der Werkzeugunterstützung unzureichend ist. Dies zeigt allerdings auch, dass nicht alle Probleme auf Sprachebene gelöst werden können, sondern dass auch auf Werkzeugebene die Entwicklung von Produktlinien unterstützt werden muss. Insbesondere für sehr große Produktlinien beziehungsweise Produktlinien mit sehr vielen Variationspunkten wird eine entsprechende Werkzeugunterstützung in Form einer integrierten Entwicklungsumgebung benötigt. Die entwickelten Werkzeuge haben wir in Form von Eclipse-Plug-Ins in FEATUREIDE zusammengefasst und als vollintegrierte Entwicklungsumgebung bereitgestellt.

In den folgenden Kapiteln dieser Dissertation wollen wir Lösungen für die zuvor beschriebenen Probleme vorstellen. Diese Lösungen sollen die Erstellung der Nanodatenmanagementfunktionalität in Form einer Produktlinie verbessern.

4.6 Zusammenfassung

Merkmalsorientierte Entwicklung eines Speichermanagers

In diesem Kapitel stellten wir die explorative Studie zur ersten Evaluierung der Merkmalsorientierten Programmierung im Umfeld des Datenmanagements für eingebettete Systeme vor. Hierzu wurde ein Speichermanager mittels der Merkmalsorientierten Programmierung auf Basis von Java umgesetzt. Die Grundlage dieser Fallstudien bildete ein idealisiertes Szenario aus dem Bereich der Sensornetzwerke für die Biotopüberwachung. Hierauf aufbauend wurde eine Domänenanalyse zur Bestimmung der Variationspunkte durchgeführt. Die anschließende Implementierung erfolgte durch Merkmalsorientierte Programmierung unter Verwendung der AHEAD-Tool-Suite auf Basis einer Java-Erweiterung. Aus der Implementierung konnten durch Konfiguration und automatisierte Komposition verschiedene Varianten abgeleitet werden.

Ergebnisse

Insgesamt konnten wir zeigen, dass die Merkmalsorientierte Programmierung auf Basis der AHEAD-Tool-Suite eine gute Ausgangsbasis für hochkonfigurierbare Datenmanagementsoftware bereitstellt. Hinsichtlich der Granularität der Variationspunkte, der Wiederverwendbarkeit und der Automatisierbarkeit der Konfigurationen wurden sehr gute Ergebnisse erzielt. Bezüglich der querschneidenden Belange zeigte sich, dass der von uns vorgestellte Entwurf solche querschneidenden Belange in Form von heterogenen Erweiterungen - durchschnittlich wurden 3 Klassen pro Merkmal erweitert - aufweist, die sich aber sehr gut durch die Merkmalsorientierte Programmierung umsetzen ließen. Homogene Erweiterungen sind in unserem idealisierten Szenario nicht aufgetreten, werden aber in den Bereichen Fehlerbehandlungen, Transaktionsverwaltung und Mehrbenutzersynchronisation vermutet und müssen deshalb genauer untersucht werden.

Unzureichende Aussagekraft der Studie

Dagegen sind belastbare Aussagen als Ergebnis dieser Fallstudie hinsichtlich des zu erwartenden Overheads durch diese Art der Umsetzung grundsätzlich nur auf einem sehr abstrakten Level möglich. Durch die beiden *Jampack*-Transformationsmechanismen der AHEAD-Tool-Suite wird dem Nutzer grundsätzlich die Möglichkeit eingeräumt, optimierend einzugreifen.

Probleme auf der Implementierungsebene

Die Grenzen des Ansatzes zeigten sich im Umgang mit optionalen Merkmalen, wenn sie auf Implementierungsebene miteinander interagieren. Diese Interaktionen können zwar gekapselt werden, genügen aber nicht mehr der geforderten Merkmalskohäsion.

Des Weiteren ist die Merkmalsorientierte Programmierung wenig robust gegen ungeplante Änderung. Sobald eine Änderung nicht in das bisherige Abstraktionslevel der ausgewählten Dekompositionsstruktur passt, können diese Erweiterungen nur schwer integriert werden.

Ausblick

In den folgenden Kapiteln werden aufbauend auf dieser explorativen Studie die Sprachерweiterung FEATUREC++, drei variable Datenbankproduktlinien und FEATUREIDE als Werkzeugunterstützung vorgestellt.

KAPITEL 5

FEATUREC++: Merkmalsorientierte Programmierung in C++

*Dieses Kapitel enthält Ergebnisse der Veröffentlichungen GPCE 2005: „FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming“ (*APEL, LEICH, ROSENMÜLLER, SAAKE *[ALRS05]), ICSE 2006 „Aspectual Mixin Layers: Aspects and Features in Concert“(*APEL, LEICH, SAAKE *[ALS06]) und IEEE-TSE-Journal 2008 „Aspectual Feature Modules“ (*APEL, LEICH, SAAKE *[ALS08]) sowie der Veröffentlichung MVCDC 2005 „Handling Optional Features in Software Product Lines“ (*LEICH, APEL, ROSENMÜLLER, SAAKE *[LARS05]). Implementierungsarbeiten wurden von* ROSENMÜLLER *„Merkmalsorientierte Programmierung in C++“ [Ros05] im Rahmen einer zu dieser Dissertation betreuten Abschlussarbeit vorgenommen.*

Im vorangegangenen Kapitel haben wir gezeigt, dass die Merkmalsorientierte Programmierung eines hochkonfigurierbaren DBMS mit Hilfe der AHEAD-Tool-Suite möglich ist, aber auch an Grenzen stößt. Zum einen werden diese Grenzen aus Sicht des Software-Engineering durch das Paradigma der Merkmalsorientierten Programmierung gesetzt, zum anderen aber auch durch die Verwendung der konkreten Implementierungssprache Java. Gerade im Bereich der stark ressourcenbeschränkten eingebetteten Systeme ist der Einsatz der Sprache Java umstritten[1]. Der überwiegende Teil der Anwendungsent-

1 Sehr minimierte Versionen von Java haben aber dennoch ihren Einsatz im Bereich der eingebetteten Systeme gefunden. Einer der Haupteinsatzgebiete sind Mobilfunktelefone, die aus Kompatibilitätsgründen untereinander auf die Sprache Java zurückgreifen.

wicklung findet deshalb in den Sprachen C oder C++ statt. In diesem Kapitel werden wir zunächst vorhandene Realisierungsmöglichkeiten der Sprache C++ vorstellen, die eine Umsetzung der Merkmalsorientierten Programmierung ermöglichen. Nach einer Analyse hinsichtlich der Anwendbarkeit für den Einsatz im Bereich des hochkonfigurierbaren eingebetteten Datenmanagements wird der eigene Sprachansatz FEATUREC++ vorgestellt.

5.1 Anforderungen an die Entwicklung von FEATUREC++

Anforderungen

Für die Entwicklung von FEATUREC++ ergeben sich aus unserer explorativen Fallstudie auf der Sprachebene die folgenden Anforderungen:

Unterstützung von verschiedenen Erweiterungstypen: Wie in Abschnitt 4.4.3 beschrieben, können *homogene* und *komplexe dynamische Erweiterungen* mit der Merkmalsorientierten Programmierung nicht adäquat umgesetzt werden. FEATUREC++ soll diese Erweiterungstypen unterstützen.

Realisierung nicht-vorhersehbarer Erweiterungen: Trotz der längeren und detaillierten Planungsphase, die die Entwicklung von Produktlinien mit sich bringt, sind nicht geplante Änderungen durch den längeren Softwareentwicklungszyklus unumgänglich. Im Ergebnis der ersten FAMEDBMS-Studie ist festzustellen, dass der Merkmalsorientierte Ansatz hier Schwächen aufweist. Hierzu muss FEATUREC++ Mechanismen „zum Aufbrechen“ *nicht-hierarchiekonformer Erweiterungen* bereitstellen.

Optionale Merkmalsinteraktion: Werden in einer Produktlinie viele optionale Merkmale angeboten, steigt die Wahrscheinlichkeit, dass diese Merkmale interagieren. Die Merkmalsorientierte Programmierung bietet zur Lösung dieses Problems nur unzureichend Support an, welche nicht dem Konzept der Merkmalskohäsion entspricht. FEATUREC++ soll hierfür eine Lösung bieten.

Zusätzlich zu den Anforderungen auf Sprachebene ergeben sich für die Umsetzung die folgenden Nebenbedingungen:

Programmcodeoptimierung durch Standard-Compiler: Ein wesentlicher Bestandteil eines Kompilierungsvorgangs ist die Optimierung des Maschinencodes. Hierbei kann je nach Zielvorgaben eine Laufzeit- oder eine Programmcodeoptimierung bezüglich der Größe vorgenommen werden. Auf Grund der heterogenen Hardwa-

re und Betriebssysteme im Umfeld von eingebetteten Systemen sind viele Compiler auch auf bestimmte Umgebungen angepasst. In die plattformspezifischen Optimierer sind des Weiteren jahrzehntelange Erfahrungen eingeflossen. Bei der Entwicklung einer eigenen Spracherweiterung ist deshalb unbedingt darauf zu achten, dass sich die neuen Sprachkonstrukte in Standard C++ übersetzen lassen. So kann die eigentliche Übersetzung in Maschinensprache durch einen beliebigen C++ Compiler durchgeführt werden. Dies erhöht die Akzeptanz und Praktikabilität des Ansatzes.

Ressourceneffizienz: Entgegen dem derzeitigen Trend in der Programmiersprachenentwicklung, bei dem die Ressourceneffizienz nicht mehr im Vordergrund steht, soll FEATUREC++ speziell für die Bedürfnisse ressourcenknapper Umgebungen entwickelt werden. Im Fokus steht dabei die statische Konfigurierbarkeit, das heißt die Merkmale des Zielsystems werden zum Zeitpunkt der Übersetzung definiert. Zur Laufzeit sollen keinerlei Abfragen zum Konfigurationsumfang der Anwendung erfolgen. Die Transformation von AHEAD-Code in Standard Java-Code erzeugt eine Vielzahl abstrakter Klassen. Eine Übertragung des AHEAD-Konzeptes auf C++ zieht somit den massiven Einsatz von virtuellen Funktionen nach sich. Dieses ist im Sinn der Performanz und der eventuellen geforderten *Echtzeitfähigkeit* unbedingt zu vermeiden.

Anlehnung am AHEAD-Tool-Suite-Modell: Die AHEAD-Tool-Suite liefert ein fundiertes Modell zur Komposition von Programmen aus hierarchiekonformen, inkrementellen Erweiterungen von Merkmalen. Durch die klare *Trennung von Konfigurationsmechanismus und Quelltext* wird eine hohe Unabhängigkeit der Quellcodefragmente erreicht. Dies erhöht die Lesbarkeit des Quelltextes und *die Möglichkeiten der externen Optimierung von Merkmalen bei der Quelltextzusammensetzung*. Des Weiteren bietet AHEAD *Mechanismen zur Prüfung der semantischen Korrektheit*, die vom Quelltext unabhängig definiert werden.

Bevor unser Ansatz der Merkmalsorientierten Programmierung FEATUREC++ präsentiert wird, evaluieren wir im folgenden Abschnitt bisherige Implementierungsmöglichkeiten der Merkmalsorientierten Programmierung mit C++.

5.2 Merkmalsorientierte Programmierung mit C++

Der folgende Abschnitt zeigt existierende Implementierungsansätze auf Basis der Sprache C++, die die Merkmalsorientierte Programmierung unterstützen. Ein Anspruch auf

Vollständigkeit der vorgestellten Ansätze wird dabei nicht erhoben. Vielmehr präsentieren die Ansätze Vertreter einer Klasse.

5.2.1 Mixin Layer

Umsetzungsmöglichkeit in C++

Eine konkrete Umsetzungsmöglichkeit von Kollaborationenentwürfen bietet C++ mit seinen Standardsprachmitteln. Die Hauptidee des auch als *Mixin Layers* [VN96] bekannten Ansatzes basiert auf dem *Template*-Mechanismus von C++. Der *Template*-Mechnismus erlaubt es dem Programmierer, durch einen Parameter die konkrete Basisklasse einer Klasse bis zum Kompilierungsvorgang offen beziehungsweise variabel zu halten. Aus diesem Grund wird der Ansatz auch parametrisierte Vererbung genannt.

Mixins in C++

Abstrakte Subklassen

Mixins oder auch *abstrakte Subklassen* [BC90] sind Klassen, die zum Zeitpunkt der Implementierung keine konkrete Basisklasse aufweisen. Das Beispiel in Abbildung 5.1 zeigt die konkrete Funktionsweise in C++. Der *Template*-Parameter `class PSUPER` ermöglicht dem Entwickler, die Basisklasse bis zum Zeitpunkt der Instanziierung des *Templates* variabel zu halten. Dieser Vorgang wird auch als verzögerte Designentscheidung bezeichnet.

```
1 template <class PSUPER>
2
3 class mixin : public PSUPER { ... };
```

Abbildung 5.1: *Template*-basierte Vererbung in C++ Standard

Variable Vererbungshierarchie durch Mixins

C++ bietet ebenfalls die Möglichkeit, durch Verkettung und Kombination beliebige lineare Vererbungshierarchien zu erzeugen. Abbildung 5.2 zeigt die beiden Basisklassen `Btree` und `Queue`. Das *Mixin* in den Zeilen 4 bis 14 stellt eine Erweiterung der Basisfunktionalität um Synchronisationsfunktionalität dar. Die Zeilen 8 bis 13 zeigen beispielsweise die Verfeinerung der Basisfunktionalität der Methode `AddElement(..)`, die eine synchronisierte Abarbeitung sicherstellt. Hierzu wird über den *Template*-Parameter `PSUPER` (Zeile 5) auf die Funktionalität der Basisklasse zugegriffen. In den Zeilen 16 bis 19 ist analog zur Synchronisation ein *Mixin* zur Integritätssicherung dargestellt.
Ab Zeile 22 werden beispielhaft fünf verschiedene Konfigurationen dargestellt. Zeile 22 zeigt einen synchronisierten B-Baum (`Btree`), Zeile 23 eine synchronisierte Warteschlange (`Queue`), während die darauf folgende Zeile eine `Queue` mit Integritätsprüfungen

```
1  class Btree
2  class Queue
3
4  template <class PSUPER>
5  class Synchronize : public PSUPER {
6  ...
7
8   void AddElement (const Element& element)  {
9
10   Lock();
11   PSUPER::AddElement(element);
12   Unlock();
13   }
14 };
15
16 template <class PSUPER>
17 class Integrity : public PSUPER {
18 ...
19 };
20
21
22 class SyncBtree : public Synchronize < Btree> { .. };
23 class SyncQueue : public Synchronize < Queue> { .. };
24 class IntegrQueue : public Integrity < Queue> { .. };
25 class SyncIntegrQueue : public Synchronize < Intregrity < Queue> > { .. };
26 class SyncIntegrBtree : public Synchronize < Intregrity < Btree> > { .. };
```

Abbildung 5.2: Verschachtelte Instanziierung von Mixins

darstellt. Zeile 25 und 26 zeigen die Implementierungen einer `Queue` und eines B-Baums, die sowohl synchronisiert als auch mit einer Integritätsprüfung ausgestattet sind.

Mixin Layers in C++

Schichten von Mixins

Während der *Mixin*-Ansatz die Skalierbarkeit einzelner Klassen bietet, löst der *Mixin Layer*-Ansatz das Problem der Skalierbarkeit auf Komponentenebene. Auch hierfür bietet C++ mit seinen Standardsprachmechanismen eine Umsetzungsmöglichkeit. Die Grundlage der *Mixin Layers* [VN96] in C++ bilden *Mixins* und die Möglichkeit, Klassen zu schachteln. Die äußere Klasse bildet dabei den Rahmen der Schicht/Komponente. Diesbezüglich entspricht `BaseDB` (vgl. Abbildung 5.3) einer Kollaboration. Kollaborations-*Mixins* mit inneren Klassen erben wie einfache *Mixins* von einer nicht bekannten Basisklasse (`BaseDB` in Zeile 3 und 5). Identisch zu den äußeren Klassen erben innere Klassen des *Mixin Layers* (`Caching::StorageManager` und `Caching::BufferManager`) von den inneren Klassen der Basisklasse. In Abbildung 5.3 erfolgt dies durch die Verwendung des Parameters `PSUPER` (Zeilen 12 - 13 und 20 - 22).

```
1  // Basisschicht
2  class BaseDB {
3    class StorageManager  { ... };
4    class BufferManager  { ... };
5    ....
6  };
7
8  // Erweiterung des DBMS um einen B-Baum
9  template <class PSUPER>
10 class BtreeBase : public PSUPER {
11 public:
12   class StorageManager : public PSUPER::StorageManager { ... };
13   class Btree  { ... };
14 };
15
16 // Kollaboration 1 mit 2 Erweiterungen und einer neuen Introduction
17 template <class PSUPER>
18 class Caching : public PSUPER {
19 public:
20   class StorageManager : public PSUPER::StorageManager { ... };
21   class BufferManager : public PSUPER::BufferManager { ... };
22    class Cache { ... };
23 };
24
25 // Konfiguration 1
26 class config_1 : public Caching< BtreeBase < BaseDB > > { };
27 // Konfiguration 2
28 class config_2 : public Caching < BaseDB > { };
```

Abbildung 5.3: Mixin Layer

Probleme

Der Einsatz von *Mixin Layers* in C++ mit Hilfe des *Template*-Mechanismus zieht einige Probleme nach sich. Ähnlich zum AHEAD-Ansatz können keine homogenen, komplexen, dynamischen und nicht-vorhergesehenen Erweiterungen realisiert werden. Ebenfalls bieten C++ *Mixin Layers* keine Lösung für Interaktionen von optionalen Merkmalen und keine Werkzeugunterstützung in IDEs. Weitere Probleme sind:

Semantische Korrektheit: Eine Überprüfung der syntaktischen Korrektheit einer Konfiguration übernimmt der C++ Compiler bei der Instanziierung der *Templates*. Ob eine erzeugte Vererbungshierarchie aber semantisch korrekt ist, kann durch den Entwickler sichergestellt werden (vergleiche hierzu *Design Rule Checks*, Abschnitt 3.6.4). Dies ist ohne *Design Rule Checks* nur im Gesamtkontext der vollständigen Vererbungshierarchie festzustellen und somit bei diesem Ansatz nur schwer vom Entwickler sicherzustellen. Gerade in großen Produktlinien besteht jedoch die Notwendigkeit, den Entwickler bei dieser Überprüfung zu unterstützen.

Konstruktorproblem: Ein weiteres Problem entsteht durch die Verwendung von Nicht-Standardkonstruktoren in *Mixins* und *Mixin Layers*: Besitzt eine Basisklasse keinen Standardkonstruktor, so ist es nicht möglich, eine Instanziierung der Basisklasse durchzuführen. Eine *Mixin*-Klasse müsste alle Konstruktoren der Basis kennen, was aber dem Konzept der Unabhängigkeit widerspricht. Dieses Problem wird in der Literatur auch als Konstruktorproblem bezeichnet [SB00, EBC00]. Eine Lösung bietet die Kombination der *Mixins* mit der statischen *Template*-Metaprogrammierung [CE00, EBC00]. Die hieraus entstehenden Probleme werden in Abschnitt 5.2.2 erörtert.

***Template*-Instanziierung:** Weiterhin problematisch gestaltet sich die Instanziierung von *Templates* in großen Produktlinien, die mit zunehmender Anzahl der verfeinerten Merkmale schnell unübersichtlich wird. Auch hier bietet die *Template*-Metaprogrammierung eine Lösung, die aber ebenfalls nicht unproblematisch umzusetzen ist [CE00], S. 710 ff..

Der vorgestellte Ansatz der *Mixin Layers* stößt bezüglich der merkmalsorientierten Konfigurierbarkeit an Grenzen, die durch die statische *Template*-Metaprogrammierung behoben werden können, die im folgenden Abschnitt präsentiert wird.

5.2.2 Statische *Template*-Metaprogrammierung in C++

Im eigentlichen Sinn bildet die statische *Template*-Metaprogrammierung [CE00], S. 397 ff., wie auch die nachfolgend beschriebene Aspektorientierte Programmierung, einen eigenständigen Ansatz zur Produktlinienimplementierung und wäre deshalb als Substitut zur Merkmalsorientierten Programmierung zu sehen. Darüberhinaus bieten beide Ansätze Möglichkeiten, den Kollaborationsansatz in C++ zu unterstützen. Die folgende Betrachtung bezieht sich deshalb nur auf die Unterstützungsmöglichkeit des jeweiligen Ansatzes für die Merkmalsorientierte Programmierung.

Funktionsweise

In den Grundzügen kann die C++ *Mixin Layer*-Implementierung mit Hilfe des *Template*-Mechanismus als eine Form der *Template*-Metaprogrammierung bezeichnet werden. Darüber hinaus kann mit Hilfe einer rekursiven *Template*-Instanziierung, einer *Template*-Spezialisierung sowie entsprechender Fallunterscheidungen dieser Mechanismus verwendet werden, um Metaprogramme zur Konfiguration der Anwendung zu schreiben. Ein mögliches Anwendungsgebiet der *Template*-Metaprogrammierung im Rahmen der Merkmalsorientierten Programmierung ist das Konstruktorproblem, bei dem mit

Hilfe dieser Meta-Programme eine Parametrisierung und Instanziierung von Nicht-Standardkonstruktoren vorgenommen wird.

Probleme

Template-Metaprogrammierung bietet ein sehr komplexes Instrument für die Erzeugung von Programmfamilien in C++. Dennoch hat die Meta-Programmierung erhebliche Nachteile [CE00], S. 710 ff.:

Debugging und Fehlernachrichten: Das *Debugging* von *Templates* in C++ ist extrem schwierig. Derzeitig wird der *Debugging*-Prozess von keinem C++ Compiler unterstützt. Des Weiteren ist es nicht möglich, mit den Sprachmitteln der Meta-Programmierung zur Übersetzungszeit Error-Nachrichten herauszuschreiben. Dieses erschwert den Entwicklungs- und Testprozess extrem.

Lesbarkeit des Quelltextes: Die Lesbarkeit des Programmtextes wird auf zwei Ebenen gestört. Zum einen erfolgt durch den Einsatz von Meta-Programmierung eine Mischung von Programmcode und Meta-Programmcode, der zur Konfiguration des Programmcodes benutzt wird. Dieses verkompliziert die Lesbarkeit des Codes, da der Programmierer ständig das Abstraktionslevel wechseln muss. Zum anderen ist die Lesbarkeit des Meta-Quelltextes selbst stark eingeschränkt, da die Metasprache in C++ nicht das Ergebnis eines sorgfältigen Sprachdesign-Prozesses ist, sondern über Jahre eher stiefmütterlich in den C++ Standard integriert wurde.

Kapazitätsgrenzen und Robustheit des Compilers: Das Übersetzen von Anwendungen, die über komplexe *Template*-Metaprogramme konfiguriert werden, kann sich erheblich verlängern. Dies ist darin begründet, dass der Metaprogrammcode interpretiert wird und die Compiler für diesen Anwendungsfall nicht ausgelegt sind. Dies führt auch regelmäßig zu Abstürzen des Compilers.

Portabilität: Nicht alle Compiler unterstützen alle Erweiterungen der *Template*-Metasprache. Dies schränkt die Portabilität der Programme ein und verhindert so eine Wiederverwendung.

Die präsentierte statische Template-Metaprogrammierung ist bezüglich der Abbildung der Merkmalsorientierten Programmierung ausreichend mächtig, zeigt aber in der Praktikabilität große Schwächen.

5.2.3 P++

Funktionsweise von P++

Mit P++ stellten SINGHAL und BATORY [SB93] eine Spracherweiterung für C++ auf Basis des *GenVoca*-Konzeptes vor. Angelehnt ist die Syntax von P++ an C++ *Mixin Layers*. Die Rollen werden durch innere Klassen repräsentiert, während die äußeren Klassen ähnlich zum C++ Standard die Komponenten zusammenfassen. Der wesentliche Unterschied von P++ zur klassischen Implementierungsvariante von *Mixin Layers* ist die Verwendung von *Realms*. Diese bilden die Schnittstellenbeschreibungen der Komponenten und enthalten Klassen mit Methodendeklarationen. Die korrespondierende Implementierung der Methoden wird von den Komponenten gestellt. Die Konfiguration der Software erfolgt wie bei C++ *Mixin Layers* durch die Instanziierung der *Templates*. Ein Quellcodetransformationssystem übersetzt den P++ Quelltext in herkömmlichen C++ Code, welcher anschließend durch einen Standard C++ Compiler in Maschinencode übersetzt wird.

Probleme von P++

Die Probleme von P++ sind ähnlich zum Standardansatz von *Mixin Layers* in C++. Der Konfigurations- beziehungsweise Instanziierungsprozess ist analog komplex zu C++ *Mixin Layers*. Der wesentliche Vorteil von P++ liegt lediglich in der Definition der Schnittstellen durch *Realms*, die eine zusätzliche Prüfung der Gültigkeit einer Konfiguration ermöglichen. Sind alle Schnittstellen sauber definiert, erhält der Entwickler eine Gesamtprüfung hinsichtlich aller definierten Schnittstelleneigenschaften.

5.2.4 Entwurfsmuster

Entwurfsmuster

Eine weitere Möglichkeit, die Merkmalsorientierte Programmierung in C++ (eingeschränkt) umzusetzen, bieten Entwurfsmuster [GHJV96]. Durch den gezielten Einsatz von Vererbung entlang einer Hierarchie sowie Objektkomposition können schrittweise Erweiterungen in ein Basisprogramm eingebracht werden. Ziel von Entwurfsmustern ist die gezielte Entwicklung variabler, wiederverwendbarer und erweiterbarer Software durch die Entkopplung von Systemmerkmalen und Verhaltensweisen sowie deren Modularisierung in assoziierten Klassen. Die meisten vorgeschlagenen Entwurfsmuster führen zum Einsatz von virtuellen Funktionen. Dies ermöglicht einen Austausch der Implementierung zur Laufzeit und zögert somit viele Konfigurationsentscheidungen bis zur Laufzeit heraus.

Probleme von Entwurfsmustern

Entwurfsmuster ziehen eine Reihe von Problemen bei der Umsetzung von hierarchischen Kompositionsmechanismen nach sich:

Performanz: Durch den Einsatz von virtuellen Funktionen kann der Compiler nur sehr eingeschränkt automatische performanz-optimierende Codetransformationen wie zum Beispiel *Inlining* oder *Loop-Unrolling* durchführen. Die hieraus entstehenden indirekten Performanz-Einbußen können kaum quantifiziert werden.

Ressourcenverbrauch: Der erwähnte Einsatz von virtuellen Funktionszeigern führt zu einem durchschnittlichen Speichermehrbedarf von 3,7 Prozent [DH96]. In Systemen, die eine maximale Erweiterung versprechen, steigt der Speicherbedarf auf bis zu 13,7 Prozent an [DH96]. Des Weiteren erzeugt der *Overheadcode* der Entwurfsmuster selbst einen erhöhten Speicherverbrauch. Neben diesem entsteht, vergleichbar zur Performanz, das Problem, dass der Compiler-Optimierer nur noch sehr eingeschränkt arbeiten kann. Dies führt zu einem indirekt höheren Speichermehrverbrauch.

Lesbarkeit: Der Quelltext einer Entwurfsmusterumsetzung kann in zwei Bereiche eingeteilt werden. Zum einen gibt es den Nutzcode. Dieser Quelltext umfasst die eigentliche Programmlogik, die variabel, wiederverwendbar oder erweiterbar gestaltet werden soll. Zum anderen sprechen wir vom Infrastrukturquelltext des Design Patterns. Dieser Code schneidet die eigentliche Programmlogik quer. Dies führt zu einer höheren Komplexität des Programmcodes.

Echtzeitfähigkeit: Viele eingebettete Systeme werden im Bereich von Echtzeitszenarien eingesetzt. Um die maximale Ausführungszeit von Programmausführungen zu berechnen, wird der Quellcode analysiert. Dies ist bei der Verwendung virtueller Funktionen nicht möglich oder enorm erschwert.

Die präsentierten Entwurfsmuster sind an das Sprachkonzept der Objektorientierten Programmierung gebunden und sind diesbezüglich in ihrer Mächtigkeit durch die Konzepte der Objektorientierten Programmierung beschränkt.

5.2.5 AspectC++

Funktionsweise AspectC++

Wie die statische *Template*-Metaprogrammierung kann auch die Aspektorientierte Programmierung zur Umsetzung der Merkmalsorientierten Programmierung verwendet werden. Für die Aspektorientierte Programmierung in C++ steht unter anderem AspectC++ [SGSP02] zur Verfügung. Um im Sinne der Merkmalsorientierten Programmierung mit AspectC++ zu entwickeln, benötigt man Aspekte, die durch *Introductions* oder auch *inter-type declarations* neue *Class-Member* zu bestehenden Klassen hinzufügen

können. Des Weiteren werden *Pointcuts* zum Markieren der zu erweiternden Methode und *execution advices* zum Ausführen des verfeinernden Codes gebraucht.

Nachteile der Aspektorientierten Programmierung

Die Aspektorientierte Programmierung hat hinsichtlich der schrittweisen Erweiterung von Programmen eine Reihe von Nachteilen. An dieser Stelle werden drei wesentliche Nachteile aufgezeigt:

Programmverständnis: Das Aufbrechen der Kollaborationen durch Aspekterweiterungen im Sinn von heterogenen Erweiterungen verschlechtert das Programmverständnis, da die objektorientierte Struktur mit der Bündlung der Erweiterungen in Aspekten aufgebrochen wird [Ste06]. Dies erschwert das Programmverständnis und behindert dadurch nachfolgende Erweiterungen.

Merkmalskohäsion: Die Implementierung eines Aspektes zur Kapselung aller Erweiterungen eines Merkmals ist oft nicht möglich [LHBL06]. Häufig werden für die Erweiterungen eines Merkmals mehrere Aspekte benötigt und auch neue Klassen hinzugefügt. Die Aspektorientierte Programmierung bietet hierfür kein adäquates Konzept, das diese Quelltextfragmente zusammenhält.

Konfiguration: Erweiterungen durch Aspekte enthalten explizit im Quelltext des Aspektes die Konfigurations- und Kompositionsanweisungen. Dies entspricht nicht der gewünschten Trennung von Quelltext und Konfigurationsanweisungen. Eine Trennung von Programmcode und Konfigurationcode in zwei Aspekte erhöht die Komplexität zusätzlich und ist nicht immer möglich. Des Weiteren ist die Komposition von Aspekten schwer oder gar nicht möglich [LHBL06].

Eine ausführlichere Diskussion der Nachteile der Aspektorientierten Programmierung bezüglich der inkrementellen Erweiterungen findet sich unter anderem in [Ste06, LHBL06, Ape07].

Vorteile

Dennoch bietet die Aspektorientierte Programmierung Vorteile gegenüber dem Merkmalsorientierten Kompositionsansatz der AHEAD-Tool-Suite. Im Folgenden wollen wir diese Vorteile kurz vorstellen:

Homogene Erweiterungen: AspectC++ bietet durch seinen *Pointcut*-Mechanismus die Möglichkeit, mehrere *Join Points* zusammenzufassen. Dies ermöglicht die Kapselung von gleichen Erweiterungen in einem Aspekt und verhindert somit replizierten Quelltext.

Komplexe, dynamische Erweiterungen: Aspekte können den Kontrollfluss auch dynamisch erweitern. Hierfür steht in AspectC++ das `cflow`-Konstrukt zur Verfügung, bei dem zur Laufzeit über die Ausführung des Advice-Codes entschieden wird. Der Mechanismus hierfür wird statisch zur Übersetzungszeit eingewoben.

Nicht-hierarchiekonforme Erweiterungen: Werden Erweiterungen benötigt, die nicht mit dem *Super*-Impositionsmechanismus der Merkmalsorientierten Programmierung in der gewählten Dekompositionsstruktur umgesetzt werden können, bieten Aspekte eine Möglichkeit, diese mit Hilfe des *Pointcut*-Mechanismus vorbei an dieser Struktur umzusetzen.

Interaktion optionaler Merkmale: Der *Pointcut*-Mechanismus von Aspektsprachen ist bezüglich nicht vorhandener optionaler Merkmale tolerant. Ist ein optionales Merkmal nicht vorhanden, greift die beschriebene Signatur des *Pointcuts* nicht und der Quelltext wird nicht zur Interaktion mit diesem optionalen Merkmal eingewoben. Eine ausführliche Diskussion zu diesem Thema findet sich im Abschnitt 5.6.1.

Der vorgestellte Ansatz der Aspektorientierten Programmierung mit AspectC++ besitzt neben den gezeigten Nachteilen viele Vorteile, die andere merkmalsorientierte Ansätze bisher nicht vorweisen konnten. Aus diesem Grund eignet sich AspectC++ für eine Integration in die Merkmalsorientierte Programmierung, wie im Folgenden gezeigt wird.

5.2.6 Diskussion

Vergleich der Ansätze

Die in den vorangegangenen Abschnitten vorgestellten Umsetzungsmöglichkeiten der Merkmalsorientierten Programmierung zeigen, dass eine Vielzahl von unterschiedlichen Umsetzungsmöglichkeiten in C++ bereits existieren. Bis auf die Aspektorientierte Programmierung reduzieren alle Ansätze direkt die Codelesbarkeit, das wiederum einen indirekten Einfluss auf die Praktikabilität, Wartbarkeit und Erweiterbarkeit hat. Des Weiteren ist, bis auf den AspectC++ Ansatz, keiner dieser Ansätze speziell für den Bereich der eingebetteten Systeme entwickelt worden oder es konnte keine Eignung im Bereich der eingebetteten Systeme nachgewiesen werden. Die Ansätze, die auf dem C++ *Template*-Mechanismus [2] aufbauen, vermischen die Metasprache zur Konfiguration mit dem eigentlichen Programmcode. Tabelle 5.1 fasst die Vor- und Nachteile zusammen. Hierzu haben wir die Java-basierte Lösung der AHEAD-Tool-Suite als Referenz hinzugefügt. Keine der vorgestellten Umsetzungsmöglichkeiten kann mit vorhandenen Mechanismen einen ähnlichen Funktionsumfang wie die AHEAD-Tool-Suite für C++

2 P++, Statische *Template*-Metaprogrammierung und *Template*-basierte *Mixin Layers*

bieten. Auffällig viele Nachteile der Merkmalsorientierten Programmierung mit AHEAD werden durch AspectC++ behoben. Eine Integration beider Ansätze bietet Potential für eine Untersuchung.

	C++ Mixin Layer	*Template*-Metaprogrammierung	P++	AspectC++	Design Pattern	AHEAD-Tool-Suite (Referenz)
Heterogene Erweiterung	+	+	+	-	-	+
Homogene Erweiterung	-	-	-	+	-	-
Komplexe dynamische Erweiterungen	-	-	-	+	-	-
Robustheit gegen ungeplante Erweiterung	-	-	-	+	-	-
Optionale Merkmalsinteraktionen	-	-	-	+	-	-
Codeoptimierbarkeit durch Standard-Compiler	+	-	+	+	+	+
Mechanismen zur Prüfung semantischer Korrektheit	-	-	-	-	-	+
Externe Optimierung der Konfiguration	-	-	-	-	-	+
Ressourcenkonsum	+	+	+	+	-	+
Lesbarkeit des Quelltextes	-	-	-	-	-	+
Echtzeitfähigkeit	+	+	+	+	-	+

Tabelle 5.1: Ergebnis der Analyse hinsichtlich merkmalsorientierter Umsetzungsmöglichkeiten in C++

5.3 FEATUREC++

Basis von FEATUREC++

Auf Grund der Vorteile, die die AHEAD-Tool-Suite bietet, werden viele Konzepte aus der Java-Basis übernommen. Die Merkmale, die den Kollaborationen entsprechen, werden analog zu AHEAD in einer Ordnerstruktur des Dateisystems gekapselt. Die Erweiterungen der Klassen werden durch *Refinements* implementiert. Eine Verschachtlung von Merkmalen, also eine Definition von Teilmerkmalen, wird über die Verzeichnishierarchie realisiert. Die Kompositionsreihenfolge der Merkmale wird analog zur

AHEAD-Tool-Suite durch *Equation*-Dateien beschrieben und mögliche Beschränkungen von Kombinationen können in *Design Rule Checks* definiert werden.

Umsetzung der Merkmalsorientierten Programmierung

Im Folgenden stellen wir die Umsetzung von FEATUREC++ vor. Zunächst werden hierzu die Konstrukte zur Merkmalsorientierten Programmierung vorgestellt:

Basisimplementierung: Die Basis für die Verfeinerung bildet die C++ Standardkonforme Klassendefinition. Analog dazu können auch `structs` definiert werden. In den Zeilen 5 und 10 der Abbildung 5.4 werden in der Basisschicht Knoten (`Node`) und ein Baum (`Tree`) definiert.

```
1
2  // Layer Base Tree
3
4  //Knotenelemente definieren
5  class Node {
6    ...
7  };
8
9  //Baum  definieren
10 class Tree {
11   void Add(Node* node) {
12   ...
13   }
14 };
```

Abbildung 5.4: Basisdefinition

```
1
2  // Layer Sync
3
4  //Synchronisation des Baumes
5  refines class Tree {
6
7    SyncObject sync;
8
9    void Add(Node* node) {
10   Lock lock(sync);
11   super::Add(node);
12   }
13   ...
14 };
```

Abbildung 5.5: Erweiterung

Erweiterungen: Erweiterungen werden durch das Schlüsselwort `refines` gekennzeichnet. In Zeile 5 der Abbildung 5.5 wird der Baum um Synchronisationsfunktionalität erweitert. Auf die Funktionalität der zu erweiternden Methoden kann mit dem Schlüsselwort `super` zugegriffen (Zeile 11). Alle *Member*-Variablen sind analog zum C++ Standard privat.

Konstruktoren: Bei den Verfeinerungen müssen Konstruktoren nicht erneut definiert werden. Wird in einer Verfeinerung kein neuer Konstruktor definiert, so wird der Konstruktor der Basis propagiert. Die Initialisierung des Konstruktors erfolgt mit dem Schlüsselwort `super`.

Vererbung: Neben den Erweiterungen lässt FEATUREC++ die Vererbung von Basisklassen und Verfeinerungen zu (Zeile 4 in Abbildung 5.6). Erbt eine Klasse oder Verfeinerung von einer anderen verfeinerten Klasse der gleichen Bibliothek, so ver-

wendet FEATUREC++ die am stärksten verfeinerte Variante der Klasse. Syntaktisch wird die Vererbung analog dem Standard verwendet.

```
1
2 // Layer Balancierter Tree
3
4 class BalTree : public Tree{
5 // Implementierung des
6 // Ausgleiches
7   ...
8 };
```

Abbildung 5.6: Vererbung

```
1
2 // Layer Balanced Tree, Logger, Serializable
3
4 refines class BalTree : public Serializable,Logger{
5 //Serialisierung Funktionalitaet
6 //Protokollierung Funktionalitaet
7   ...
8 };
```

Abbildung 5.7: Mehrfachvererbung

Mehrfachvererbung: Im Gegensatz zum Java-basierten Ansatz der AHEAD-Tool-Suite bietet C++ die Möglichkeit der Mehrfachvererbung. Diese wird auch von FEATUREC++ angeboten. Abbildung 5.7 zeigt die Verwendung der Mehrfachvererbung in FEATUREC++, bei der der balancierte Baum von der Klasse `Serializable` und `Logger` erbt.

***Templates*:** Der *Template*-Mechanismus ist eine sehr gute statische Möglichkeit zur Abstraktion von Datentypen, die so Variabilität schafft. FEATUREC++ ermöglicht das Propagieren von *Templates* über die Verfeinerungshierarchie. Hierzu werden, sobald eine Verfeinerung einen *Template*-Parameter verwendet, alle weiteren Verfeinerungen ebenfalls zu einem *Template*. Abbildung 5.8 zeigt die Verwendung eines *Templates* in einer Verfeinerung. In einer späteren Verfeinerung muss der *Template*-Parameter nicht mehr beachtet werden.

Notwendige Einschränkungen: Die Merkmalsorientierte Programmierung in FEATUREC++ verlangt einiges an Disziplin von Entwicklern. Zur übersichtlicheren Gestaltung darf pro Datei nur eine Klasse definiert werden. Diese sollte wie in Java den Namen der Klasse im Dateinamen enthalten. Der Ordnername, der das Klassenfragment eines Merkmals kapselt, muss den Namen des Merkmals repräsentieren. Eine Trennung von Deklaration und Definition in verschiedenen Dateien muss in FEATUREC++ unterbleiben. Andere Klassen der gleichen Bibliotheken dürfen nicht mit einem `Include` eingebunden werden, da durch die Ordnerstruktur der Merkmale die `Includes` nicht aufgelöst werden können. Aus diesem Grund sind alle Klassen der eigenen Implementierung per `default` eingebun-

```
1
2  /** Layer TypedTree **/
3
4  //Typisierung vom Baum
5  //Elements als Templateparameter
6
7  template<typename ELMT >
8
9  class TypedNode : public Node {
10 TypedNode(const ELMT& e)
11   :data(e) {}
12
13   ELMT data;
14 };
15
16 template  <typename ELMT>
17 refines class Tree {
18
19   void Add(const ELMT& e) {
20      super::Add(new TypedNode <ELMT>(e));
21    }
22 };
```

Abbildung 5.8: *Template*-Verfeinerung

den[3]. Klassen von anderen Bibliotheken sind, wie im C++ Standard, per `include` einzubinden.

Die vorgestellten Konzepte präsentieren den nerkmalsorientierten Kern von FEATUREC++, der im folgenden Abschnitt durch die Erweiterungen mit Konzepten der Aspektorientierten Programmierung ergänzt wird.

5.4 Erweiterungen der Merkmalsorientierten Programmierung

Integration von Merkmals- und Aspekt-orientierter Programmierung

Wie in Kapitel 4.4 aufgezeigt, hat die Merkmalsorientierte Programmierung einige Schwächen. Diametral zu diesen Schwächen bietet die Aspektorientierte Programmierung Lösungen für einige dieser Schwächen. Aus diesem Grund ist FEATUREC++ um Elemente und Konzepte der Aspektorientierten Programmierung erweitert worden. In dieser Arbeit werden nur die Ergebnisse dieser Integration vorgestellt, die für diese Arbeit relevant sind. Dazu gehören im Einzelnen homogene Erweiterungen, dynamische Erweiterungen und nicht-hierarchiekonforme Erweiterungen. Verschiedene Konzepte

3 Der FEATUREC++-Compiler bestimmt automatisch, welche `includes` und welche *Forwarddeklarationen* notwendig sind.

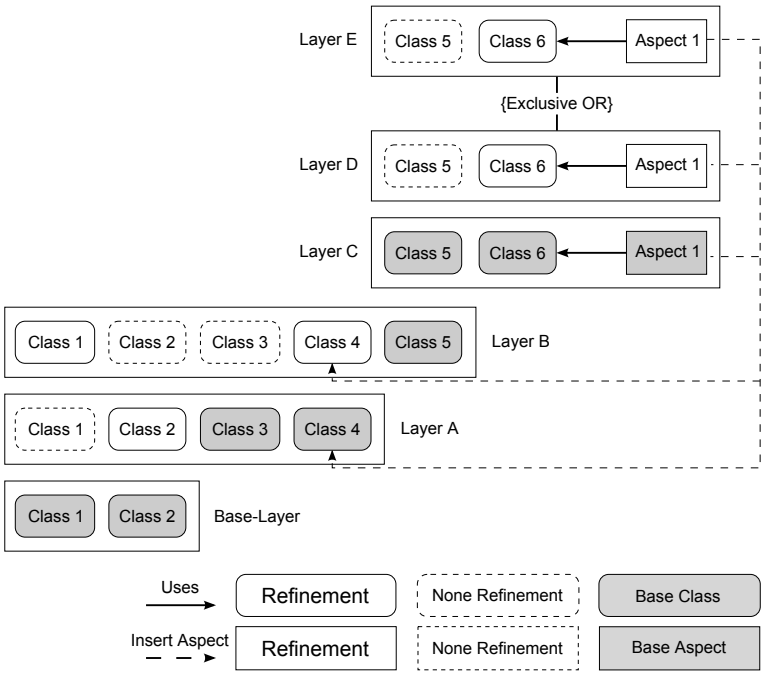

Abbildung 5.9: *Aspectual Mixin Layer*

zur Integration der Aspektorientierten Programmierung in die Merkmalsorientierte Programmierung zeigen wir in APEL et al. [ALRS05]. Die softwaretechnischen Fragestellungen im Kontext der Produktlinien und formale Aspekte der Integration werden ausführlich in der Dissertation von APEL [Ape07] und in APEL et al. [ALS06, ALS08] thematisiert.

Aspectual Mixin Layers: Integration von Aspekten

Basisidee

Die Basisidee von *Aspectual Mixin Layers* (AML) ist die Integration von Aspekten in die Merkmalsorientierte Programmierung. Abbildung 5.9 zeigt drei *Aspectual Mixin Layers* (Layer C - E). Diese Schichten beziehungsweise Merkmale enthalten neben Basisklassen und Verfeinerungen zusätzlich auch alle Aspekte und Verfeinerungen von Aspekten. Dies ermöglicht eine Komposition von Klassen und Aspekten gleichermaßen in einer Einheit und entspricht somit dem Konzept der Merkmalsorientierten Programmierung sowie dem Konzept der Uniformität der AHEAD-Tool-Suite, die eine Kapselung im Sinn der Kohäsion von sämtlichen Implementierungseinheiten fordert. *Aspectual Mixin Layer* ermöglicht eine Kohäsion sowohl von Klassenfragmenten als auch von Aspekten.

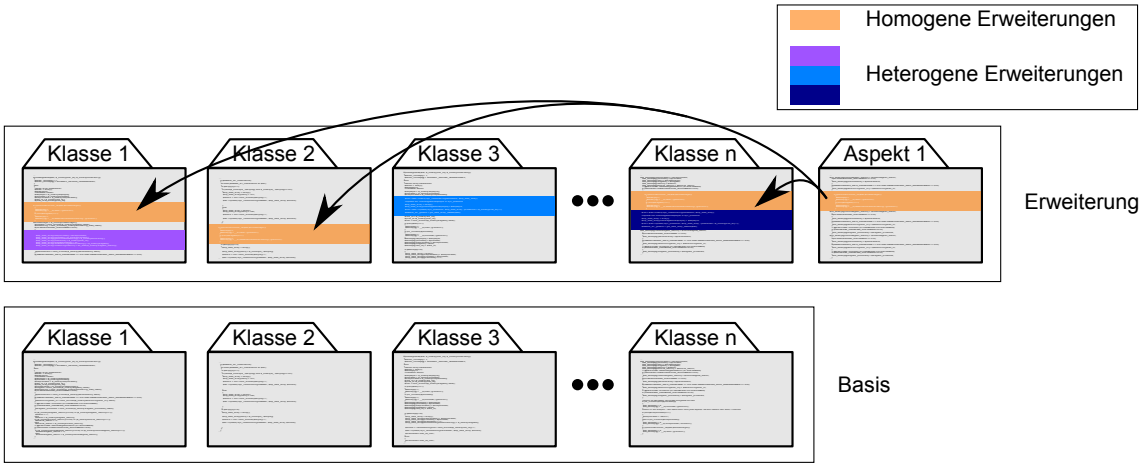

Abbildung 5.10: *Aspectual Mixin Layer* im Umgang mit homogenen und heterogenen Erweiterungen

Verwendung von Aspectual Mixin Layer

Eingesetzt werden die *Aspectual Mixin Layer* immer, wenn homogene Erweiterungen der Basis-*Mixins*, nicht-hierarchiekonforme Erweiterungen und dynamische Erweiterungen des Kontrollflusses benötigt werden. Abbildung 5.10 zeigt die Verwendung von Aspekten in einem *Aspectual Mixin Layer*. Während die heterogenen Erweiterungen klassisch durch *Mixins* erfolgen, werden die homogenen Erweiterungen durch Aspekte realisiert, indem die homogenen Quelltextfragmente an die entsprechenden Stellen gewebt werden.
Die Abbildungen 5.11 und 5.12 zeigen die Basisimplementierung eines Protokollierungsmerkmals, das die Logging-Funkionalität mit Hilfe des `LoggingAspect` an die verschiedenen Stellen der Klassen `StorageManager` und `BufferManager` einfügt.

```
1  // Layer Log
2
3  class LogFile {
4
5  void log(const char*) {
6    ...
7    ...
8    ...
9    ...
10   ...
11   }
12 };
```

Abbildung 5.11: Basisklasse zum Protokollieren

```
1  // Layer Log
2
3  aspect LoggingAspect {
4    LogFile _logFile;
5
6  pointcut log() = call("%StorageManager::%(...)"
7                     ||  "%BufferManager::%(...)");
8
9  advice log() : before() {
10   _logFile.log(tjp->signature());
11   }
12 };
```

Abbildung 5.12: Basisaspekt zur homogenen Erweiterung der Protokollierungsfunktion

Die Abbildungen 5.13 und 5.14 stellen die Verfeinerung der Protokollierungsfunktionalität dar, bei der zunächst in 5.13 das `LogFile`–Format erweitert wird. Im Aspekt `LoggingAspect` erfährt der *Pointcut* `log()` eine Erweiterung um die Klassen `BTree`, `Hash` und `Cache`.

```
1 // Layer ExLog
2
3 refines class LogFile {
4 void  setExLogFormat () {
5   ..  }
6 LogFile ()  {
7   setLogFormat ();
8   }
9 };
```

Abbildung 5.13: Verfeinerung der Protokollierung

```
1 // Layer ExLog
2
3 refines aspect LoggingAspect {
4 //Pointcut-Erweiterung
5 pointcut log() = call("%BTree::%(...)"
6                     ||"%Hash::%(...)")
7                     ||"%Cache::%(...)")
8                     ||  super :: log ());
9 };
```

Abbildung 5.14: Aspektverfeinerung der Protokollierung

Beschränkung des Wirkungsraumes von Aspekten

Das Zusammenführen der Aspektorientierten Programmierung mit der Merkmalsorientierten Programmierung verstärkt das Problem von Aspektorientierten Sprachen, die keine Beschränkung der Aspekte auf eine definierte Menge von Merkmalen zulassen. Dies führt dazu, dass die Wirkungsweisen von Aspekten zusätzlich bei jeder Erweiterung überprüft werden müssen, da es sonst zu unvorhersagbaren Effekten auf der Quelltextebene kommen kann. Unsere *Aspectual Mixin Layer* beschränken deshalb die Wirkung dieser Aspekte auf die eigene Schicht und die Schichten, die bis zu dieser Schicht verfeinert werden. Dies verhindert, dass später implementierte Merkmale die Wirkungen von Aspekten berücksichtigen müssen.

5.5 Umsetzung von FEATUREC++

Ablauf der Übersetzung

Die Umsetzung unserer Spracherweiterung in einen eigenen Übersetzer (*Compiler*) ist zu aufwendig. Deshalb wurde FEATUREC++ in Form einer Quelltexttransformation umgesetzt. Eine Quelltexttransformation beinhaltet die Umwandlung von einem Quelltext in eine andere Form. Diese wird dann durch einen Compiler in Maschinencode übersetzt. Abbildung 5.15 zeigt den prinzipiellen Ablauf von FEATUREC++. Die Umwandlung des merkmalsorientierten Programmquelltextes erfolgt dabei in den folgenden sechs Schritten:

1. Schritt: Im ersten Schritt wird durch den *Equation Parser* die gewählte Konfiguration, die analog zur AHEAD-Tool-Suite in einer *Equation*-Datei definiert wird, ausge-

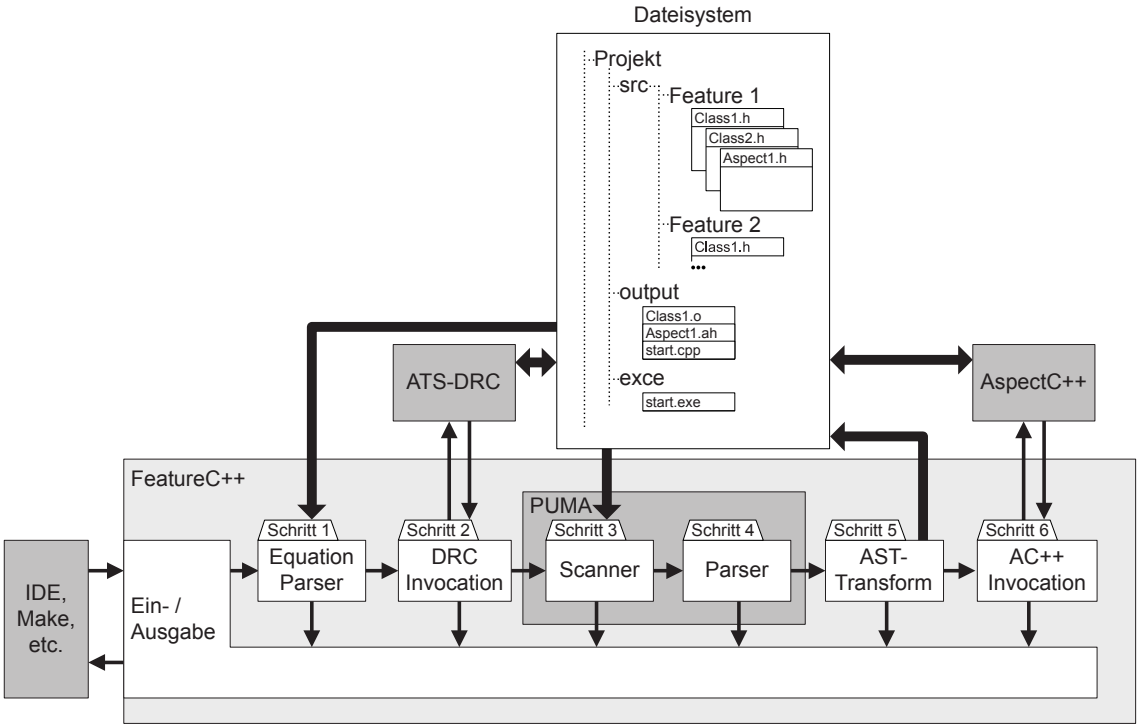

Abbildung 5.15: Ablaufübersicht der Übersetzung in FEATUREC++ [Ros05] (verändert)

wertet und die entsprechenden Quelltextfragmente werden aus den Ordnern der Merkmale zur Verarbeitung markiert.

2. Schritt: Im zweiten Schritt werden die *Design Rule Checks* durchgeführt, die die semantische Korrektheit überprüfen. Hierzu werden die in den einzelnen Ordnern eines Merkmals abgelegten `drc-Dateien` durch die Bibliotheken der AHEAD-Tool-Suite geprüft.

3. Schritt: Im dritten Schritt findet die lexikalische Analyse durch den *Scanner*[4] statt. Dabei wird der Quelltext in *Token* zerlegt. Solche *Token* können beispielsweise Schlüsselwörter, Zahlen oder Operatoren sein. Für die Umsetzung von FEATUREC++ wird für diesen und den nächsten Schritt die *Pure Manipulator* (PUMA)-Bibliothek [ULS10] verwendet. PUMA wurde zunächst zur Analyse von C++ Quelltext entwickelt und bietet dazu die Möglichkeit, *Abstract Syntax Tree* (AST) zu erstellen und diese zu transformieren. Im Rahmen der Entwicklung von AspectC++ wurde PUMA zum Parsen von aspektorientiertem Quelltext erweitert. Für unsere

4 *Scanner* und *Lexer* können synonym verwendet werden.

Umsetzung von FEATUREC++ wurde PUMA beispielsweise um die neuen *Token* `refines` und `super` ergänzt. Die Ausgabe dieses Schrittes ist ein *Token-Stream*.

4. Schritt: Im vierten Schritt erfolgt das Parsen des *Token-Streams* und die Umwandlung in abstrakte Syntaxbäume. Die AST enthalten die syntaktischen Informationen des Quelltextes und bilden die Basis für die folgenden Quellcodetransformationen.

5. Schritt: Im fünften Schritt erfolgt die Transformation der AST. Hierzu werden auf den AST die Transformationen vom merkmalsorientierten Quelltext in den objektorientierten Quelltext vorgenommen. Die Aspektverfeinerungen werden in AspectC++ konforme Vererbungshierarchien umgewandelt.

6. Schritt: Im letzten Schritt erfolgt eine weitere Transformation des Quelltextes. Hierzu werden die Aspekte durch AspectC++ in Standard C++ Quelltext transformiert. Dieser Quelltext wird dann beispielsweise durch den Microsoft C++ Compiler oder durch *gcc* in Maschinencode gewandelt.

Das Ergebnis des 6-stufigen Prozesses ist die Übersetzung von merkmalsorientierten Quelltext in nativen C++ Quelltext, der mit Standard-*Compilern* in Maschinencode umgewandelt werden kann. Im folgenden Abschnitt werden das Problem der optionalen Merkmalsinteraktionen sowie entsprechende Lösungsansätze desselben durch FEATUREC++ präsentiert.

5.6 Problem: Optionale Merkmalsinteraktion

Feature Optionality Problem

Ein Problem in der Implementierung von Softwareproduktlinien ist die Interaktion von optionalen Merkmalen. Wegen der hierarchischen Kompositionsform der Merkmalsorientierten Programmierung wird dieses Problem auch *„Feature Optionality Problem in Hierarchical Designs“* genannt. PREHOFER und LIU et al. [Pre97, LBN05] stellen ein Beispiel für diese Art der Merkmalsinteraktion am Beispiel einer *Stack*-Produktlinie vor.

5.6.1 Beispiel

Beispiel Basis-Stack

Abbildung 5.16 zeigt eine modifizierte Version dieser *Stack*-Produktlinie. Das Basismerkmal `BaseStack` implementiert die vier Basis-*Stack*-Operationen `empty()`, `push()`, `top()` und `pop()` (Zeile 3 - 17). Das zweite Merkmal `Concat` erweitert den `Stack` mit Funktionalitäten zur Kombination von zwei *Stacks*. Die `concat()`-Methode fügt diese Funktionalität hinzu (Zeile 21 - 27).

```
1  //Layer ../Stack/BaseStack/BaseStack.fcc
2
3  class stackOfChar {
4      String s;
5      void empty() {
6        s = "";
7      }
8      void push ( char a ) {
9        s = a + s;
10     }
11     void pop() {
12       s = s.substring(1);
13     }
14     char top() {
15       return s.charAt(0);
16     }
17 };
18
19 //Layer ../Stack/Concat/Concat.fcc
20
21 refines class stackOfChar {
22    void concat(stackOfChar& other) {
23      while (!other.empty()) {
24           push(other.top());
25           other.pop();}
26           }
27 };
28
29 //Layer ../Stack/Log/Log.fcc
30
31 refines class stackOfChar {
32    void concat(stackOfChar& o) {
33       cout << "concat␣2␣stacks" << endl;
34       super::concat(o);
35    }
36    void push(char a) {
37       cout << "push:␣" << a << endl;
38       super::push(a);
39    }
40    void pop() {
41       cout << "pop:␣" << top() << endl;
42       super::pop();
43    }
44 };
```

Abbildung 5.16: Merkmalsinteraktionen am Beispiel einer *Stack*-Produktlinie

Merkmal `Log`

Das dritte Merkmal `Log` erlaubt die Protokollierung von Operationen. Deshalb werden die Basis-*Stack*-Operationen wie `push()`, `pop()` und die `concat()`-Operation verfeinert. Diese gängige Implementierung führt zu einem Problem, wenn wir einen *Stack* wollen, der ohne die `concat`-Funktionalität protokolliert wird. Wenn das `concat`-Merkmal fehlt, versucht das `log`-Merkmal, ein nicht-existierendes Merkmal zu erweitern. Das `concat`-Merkmal ist *optional*. Deshalb wird dieses Problem auch als *feature-optionality problem in feature-orientierten Designs* [Pre97] bezeichnet.

Diskussion

Es gibt verschiedene Ansätze, die dieses Problem lösen. PREHOFER hat zur Lösung dieses Problems *Lifter* vorgestellt [Pre97]. *Lifter* kapseln Merkmalsabhängigkeiten und entkoppeln diese in zusätzliche Merkmale. Die Idee dabei ist, verschiedene Interaktionen in verschiedene Module zu zerlegen. Somit werden die Interaktionen und die zugrundeliegenden Basismerkmale in verschiedene zusätzliche, künstliche Schichten zerlegt und separiert. Das Abbilden von *Liftern* in der Merkmalsorientierten Programmierung erzwingt zusätzliche abstrakte Merkmale[5]. Diese künstlichen Merkmale verhindern das Konzept der *Merkmalskohäsion* [LHBC05], die möglichst eine eins-zu-eins Abbildung von Merkmalen des Entwurfs zu Implementierungseinheiten fordert.

5.6.2 Lösungsansatz: Optionales Weben

Optionales Weben

Im Folgenden stellen wir drei Möglichkeiten zum Umgang mit Erweiterungen von optionalen Merkmalen in *FeatureC++* vor. Alle drei Möglichkeiten zum Umgang mit optionalen Merkmalen können nach KÄSTNER [Käs07a] als „optionales Weben“ bezeichnet werden.

Variante I

In der ersten von uns präsentierten Variante wird der Webemechanismus der Aspektorientierten Programmierung als Grundlage zur Lösung des Problems genutzt. Die Beschreibung von Einsprungspunkten durch *Pointcuts* ermöglicht dem Programmierer unter anderem die Definition von Kontrollflusserweiterungen, die gegenüber nichtvorhandenen Quelltextfragmenten robust sind. Somit wäre es dem Programmierer möglich, alle optionalen Methodenerweiterungen in einem Aspekt auszulagern. Ist eine Methode bei einer Erweiterung nicht vorhanden, so greift der *Pointcut* nicht und die Erweiterung wird nicht in den Quelltext eingefügt. Dies erlaubt eine einheitliche Kapselung in einem Merkmal, aber die Zerlegung dieser optionalen Funktionalitäten in Aspekte und *Mixins* birgt ebenfalls Probleme in sich. So wird beispielsweise die Erweiterungshierarchie durch den Aspekt verlassen und ist damit sehr fehleranfällig,

5 LIU et al.[LBN05] bezeichnen diese (künstlichen) Merkmale als *Derivate*.

da der *Pointcut* nur auf einer Zeichenfolge des Bezeichners arbeitet und so Fehler nicht erkannt werden.

Variante II

Deshalb haben wir uns in unserer zweiten Variante dazu entschieden, die Funktionalität im *Mixin* zu behalten. Durch das Hinzufügen der Schlüsselwörter `before, after` und `around` kann eine Methode als optionale Erweiterung gekennzeichnet werden. Die Signatur, die sonst in den *Pointcut* eingebracht wurde, wird nun implizit aus dem Methodennamen übernommen.

```
1  //Layer ../Stack/Log/Log.fcc
2
3  refines class stackOfChar {
4
5     void concat(stackOfChar& other) : before()
6      {
7          cout << "concating␣2␣stacks" << endl;
8      }
9     void push(char a)
10    {
11         cout << "push:␣" << a << endl;
12         super::push(a);
13    }
14    void pop()
15    {
16         cout << "pop:␣" << top() << endl;
17         super::pop();
18    }
19 };
```

Abbildung 5.17: Optionale Methodenverfeinerung

Abbildung 5.17 zeigt noch einmal das Protokollierungsmerkmal, wobei wir die `concat()`-Methode implizit optional durch Nutzung des `before`-Schlüsselwortes (Zeile 5) gestaltet haben. Des Weiteren weist das Schlüsselwort `before` darauf hin, wann die Funktionalität der Erweiterung eingebunden wird. Das `super`-Schlüsselwort (Zeile 12 und 17) wird nach wie vor für die Erweiterung von obligatorischen Merkmalen verwendet. Diese Unterscheidung zwischen obligatorischen und optionalen Methodenerweiterungen von Merkmalen ermöglicht es uns, während der Kompositionsphase der Merkmale, Fehler zu finden. Alternativ zu `before` oder `after` ist das `around`-Schlüsselwort verwendet worden. Abbildung 5.18 zeigt die beispielhafte Verwendung des `around`-Schlüsselwortes.

Variante III

Eine weitere Möglichkeit (Variante III) der Kennzeichnung von optionalen Merkmalen ist das Einführen eines neuen Schlüsselwortes `optional`, welches innerhalb der

```
1  ....
2  void concat(stackOfChar& other) : around()
3  {
4   cout << "begin concating 2 stacks" << endl;
5   proceed(other);
6   cout << "end concat" << endl;
7  }
8  ....
```

Abbildung 5.18: Optionale Methodenverfeinerung durch `around()`

Klassenverfeinerungen die optionalen Methodenerweiterungen mit dem Standard Super-Impositionsmechanismus (vgl. Abbildung 5.19) kennzeichnet. So kann der Kompositionsprozess im Fall einer nicht vorhandenen Basismethode analog zum Aspekt diese Methode ignorieren. Des Weiteren haben wir bisher nur eine einzige Klasse betrachtet. Die Verfeinerung von optionalen Merkmalen kann aber auch neue Klassen einführen, die nur in Abhängigkeit vom optionalen Merkmal benötigt werden. Grundlegend kann eine „*Death-Code*"-Analyse diese Quellcodefragmente ausfindig machen. Diese Analysen sind aber auch in großen Produktlinien sehr aufwendig. Um hier eine Vereinfachung zu erreichen, bieten wir unser neues Schlüsselwort `optional` auch zum Einfügen von neuen Klassen an, die nur zur Verfeinerung von optionalen Merkmalen benötigt werden. Somit kann eine Schicht durch Nutzung der `refines optional class`-Anweisung eine optionale Klasse einführen und explizit ankündigen, dass diese Klasse nur mit einer optionalen Merkmalserweiterung genutzt wird. Wir haben den Zugriff auf diese optionalen Klassen nur auf die Klassen beschränkt, die an diesem *Mixin Layer* zum Verfeinern der optionalen Methoden beteiligt sind.

```
1  ....
2  refines optional void concat(stackOfChar& o)
3  {
4   cout << "begin concating 2 stacks" << endl;
5   super::concat(o);
6   cout << "end concat" << endl;
7  }
8  ....
```

Abbildung 5.19: Optionale Methodenverfeinerung durch Super-Imposition und Optional-Schlüsselwort

5.6.3 Diskussion

Diskussion optionales Weben

Während Variante I ohne zusätzliche neue Schlüsselwörter mit den Standard Sprachmitteln von *Aspectual Mixin Layer* auskommt, bedarf es bei der zweiten Variante schon der Erweiterung der *Mixin*-Syntax. Diese Integration ermöglicht im Gegensatz zur ersten Variante eine hierarchiekonforme Erweiterung. Sie bildet eine bessere Grundlage für folgende Erweiterungen. Des Weiteren werden durch die implizite Definition der Methodensignatur Fehler vermieden. Die dritte Variante stellt die volltransparente Integration des optionalen Webens in die Merkmalsorientierte Programmierung dar. Durch die „Warnung“ `optional` wird klar signalisiert, dass diese Erweiterung nur im Zusammenhang mit einem optionalen Merkmal greift. So wird auch bei späteren Erweiterungen dieses Merkmals gewarnt, dass dieses Quelltextfragment nur durch Interaktion mit einem anderen optionalen Merkmal existiert. Mit Hilfe von Werkzeugunterstützung können bei der Implementierung je nach gewählter Konfiguration die nicht benötigten optionalen Quelltextfragmente bei Bedarf über einen Sichtenmechanismus ausgeblendet werden.

5.7 Verwandte Arbeiten

Merkmalsorientierte Programmierung

Neben dem AHEAD-Ansatz gibt es eine Reihe von schichtenbasierten Ansätzen auf der Basis der Programmiersprache Java. Dies sind beispielsweise *Delegation Layers* [Ost01], *Jiazzi* [MFH01], *Java Layers* [CL01]. *Jiazzi* ermöglicht eine Komposition der Komponenten auf Binärcode. Delegation Layers realisieren mit einem entsprechenden *Overhead* die Komposition zur Laufzeit.

Metaprogramming in C++

FOG [WM00], *OpenC++* [CM93, Chi95] und *MPC++* [IHS+96] bieten für C++ verschiedene Metaobjektprotokolle, die eine Konfiguration auf einem Metalevel unterstützen, aber wenig praktikabel sind.

Kombination von Aspekt- und Merkmalsorientierter Programmierung

Vergleichbare Ansätze zu FEATUREC++ bezüglich der Kombination von Aspektorientierter Programmierung mit Schichtenbasierten Entwürfen bieten *Adaptive Plug-and-Play Components* [ML98], *Aspectual Components* [LLM99], *Aspectual Collaborations* [LO03] und *CaesarJ* [MO04a, AGMO06] auf Java-Basis. Für einen Vergleich zu FEATUREC++ greifen wir auf *CaesarJ* zurück, der die wichtigsten Eigenschaften dieser Vertreter vereint. Vergleichbar zu FEATUREC++ bietet *CaesarJ* Kollaborationen als Basis für die Softwareentwicklung an. Diese werden in *CaesarJ* durch *Aspect Components* auf Sprachebene im Quellcode miteinander komponiert.

Durch unseren AHEAD-Ansatz können wir außerhalb des Quelltextes die Komposition der Software durchführen, das neben der Lesbarkeit des Quelltextes auch Optimierungspotential bietet. Im Gegensatz zu FEATUREC++ ermöglicht *CaesarJ* dafür bei Bedarf zur Laufzeit einen Austausch der Module. Dieses ist in FEATUREC++ auf Grund der statischen Kompositionsform nicht möglich. Des Weiteren wurden bei der Entwicklung von *CaesarJ* keine Anforderungen von eingebetteten Systemen berücksichtigt.

Erweiterte Präprozessoransätze

KÄSTNER erweitert in seinen Arbeiten [Käs07a, Käs10] den Präprozessoransatz mit Hilfe von Werkzeugunterstützung so weit, dass bei der Entwicklung ein Sichtenmechanismus unter anderem mit Hintergrundfarben eine virtuelle Trennung der Belange erreicht. Durch die virtuelle Trennung von Belangen versucht *Kästner* die Lesbarkeit zu verbessern.

Erweiterungen und Einsatzgebiete von FEATUREC++

ROSENMÜLLER ergänzt in seiner Dissertation [Ros11] FEATUREC++ insofern, dass je nach Anforderungen der Anwendungsszenarien dynamische *binding units* Merkmale nicht nur statisch zum Zeitpunkt der Programmerstellung, sondern auch dynamisch zum Programmstart beziehungsweise zur Laufzeit entsprechend den Anforderungen eingebunden werden können. Damit ist es dem Entwickler möglich, unter Performanzaspekten statische oder dynamische Merkmalsauswahl anzubieten. Dies kann in der Entwicklung vollkommen transparent gestaltet werden.

FeatureHouse

Mit FEATUREHOUSE zeigen *Apel* et al. [AKL09] die konsequente Weiterentwicklung der Merkmalsorientierten Programmierung. FEATUREHOUSE kann Quellcodefragmente von Java, C#, C, Haskell, JavaCC, Alloy und UML komponieren, aber nicht C++.

5.8 Zusammenfassung

Merkmalsorientierte Programmierung in C++

In diesem Kapitel haben wir die Umsetzung der Merkmalsorientierten Programmierung für C++ mit dem Fokus auf eingebettete Systeme gezeigt. Die Grundlage hierfür bildete die AHEAD-Tool-Suite und die Ergebnisse aus unserer explorativen Fallstudie. Hierzu wurden zunächst verschiedene Umsetzungsmöglichkeiten der Merkmalsorientierten Programmierung in C++ untersucht. Da bisherige Techniken in C++ unzureichend sind, wurde FEATUREC++ auf Grundlage der AHEAD-Tool-Suite neu entwickelt. FEATUREC++ erlaubt die statische Komposition von Merkmalen in Form von Klassenfragmenten. Des Weiteren wurde der Java-Ansatz um C++ spezifische Konstrukte erweitert. FEATUREC++ bietet Mehrfachvererbung, *Templates* und Typenvarianz mit Typisierung zum Zeitpunkt der Programmerzeugung.

Integration der Aspektorientierten Programmierung

Über den konzeptionellen Möglichkeiten der AHEAD-Tool-Suite integrierten wir Elemente der Aspektorientierten Programmierung. Wir nennen diesen Ansatz *Aspectual Mixin Layers*. Dies gibt dem Entwickler die Möglichkeit, auf Konzepte der Aspektorientierten Programmierung zurückzugreifen, wenn Konzepte der Merkmalsorientierten Programmierung keine adäquaten Lösungen bieten.

Homogene Erweiterungen

Die Integration von Aspekten in Merkmalen ermöglicht den Umgang mit homogenen Erweiterungen in schichtenbasierten Entwürfen. Dabei werden die homogenen Erweiterungen mit dem *Pointcut*-Mechanismus der entsprechenden *join points* beschrieben und mit dem *Advice*-Code wird der entsprechende Quelltext an diesen Stellen eingewebt.

Nicht-hierarchiekonforme Erweiterungen

Werden nicht-hierarchiekonforme Erweiterungen, also Erweiterungen, die nicht mit dem Super-Impositionsmechanismus in der gewählten Dekompositionshierarchie umgesetzt werden können, benötigt, können Aspekte diese Erweiterungen im entsprechenden Merkmal kapseln.

Optionale Merkmale

Des Weiteren bietet FEATUREC++ die Möglichkeit optionale Merkmale im Gegensatz zum *Lifter*-Ansatz zu kapseln. Hierzu nutzen wir Techniken des optionalen Webens.

KAPITEL 6

FEATUREIDE: Werkzeugunterstützung

*Dieses Kapitel enthält Ergebnisse der Workshop-Veröffentlichung OOPSLA Eclipse Technology eXchange „*FEATUREIDE*: An Extensible Framework for Feature-Oriented Software Development“ (*LEICH, APEL, MARNITZ, SAAKE *[LAMS05]), der ICSE-Demo 2009 „*FEATUREIDE*: Tool Framework for Feature-Oriented Software Development“ (*KÄSTNER, THÜM, SAAKE, FEIGENSPAN, LEICH, WIELGORZ, UND APEL *[KTS+09]) und Journals Science of Computer Programming 2012 „*FEATUREIDE*: An Extensible Framework for Feature-Oriented Software Development“ (*THÜM, KÄSTNER, BENDUHN, MEINICKE, SAAKE, UND LEICH *[TKB+12])*. Implementierungsarbeiten wurden von MARNITZ *„Werkzeugunterstützung für die Merkmalsorientierte Softwareentwicklung“* [Mar05] einer im Rahmen dieser Dissertation betreuten Abschlussarbeit vorgenommen.

Ein Problem, das sich in unserer explorativen Studie zeigte, ist die fehlende Werkzeugunterstützung für die Merkmalsorientierte Softwareentwicklung. Hierbei sollte eine integrierte Entwicklungsumgebung alle Phasen des Softwareentwicklungsprozesses konsistent und benutzerfreundlich zusammenführen. Im folgenden Kapitel zeigen wir die Konzeption und Umsetzung einer integrierten Werkzeugunterstützung für die Merkmalsorientierte Softwareentwicklung. Ausgehend von Problemen auf Prozess-, Implementierungs- und Konfigurationsebene bei der Umsetzung von komplexen Produktlinien skizzieren wir verschiedene Herausforderungen des merkmalsbasier-

ten Entwicklungsprozesses. Daraufhin stellen wir unseren Ansatz FEATUREIDE[1] als eine integrierte Entwicklungsumgebung auf Eclipse-Basis[2] vor.

6.1 Analyse der Werkzeugunterstützung

Im folgenden Abschnitt präsentieren wir eine Analyse der wesentlichen Probleme, die ohne einen entsprechenden Werkzeugeinsatz die Entwicklung komplexer Produktlinien erschweren. Beginnen wollen wir mit den Problemen auf Entwicklungsprozessebene. Darauf folgend stellen wir die Anforderungen an die Merkmalsanalyse vor. Geschlossen wird dieser Abschnitt mit einer Diskussion von Problemen auf der Implementierungsebene.

6.1.1 Probleme des Entwicklungsprozesses

Es gibt ein weites Spektrum unterschiedlicher Entwicklungsebenen, das während der Planung, Analyse, Konzeption, Umsetzung, Produktgenerierung und vor allem im gesamten Prozess der Wartung und Weiterentwicklung einer Produktlinie durch einen konsistenten Informationsfluss integriert werden muss. Unsere Beobachtungen in der explorativen Studie zeigten, dass es bei den eingesetzten Werkzeugen der AHEAD-Tool-Suite kaum eine Integration des Prozesses gab. Des Weiteren gibt es keine kontextorientierten, integrierten Sichten oder Softwarevisualisierungen zur intuitiven Navigation und Verwaltung der verschiedenen Quelltextfragmente von Merkmalen. Somit ist die Handhabung von Produktlinien mit mehr als 200 Merkmalen, wie unsere feingranularen Dekompositionen aus dem Bereich des Nanodatenmanagements zeigten, nicht umsetzbar. Ein Hauptziel unserer Entwicklung ist es, Fehler durch einen konsistenten Entwickungsprozess zu verhindern. Hierzu sollen möglichst viele Schritte des Entwicklungsprozesses durch Automatisierung und integrierte Konsistenzprüfungen unterstützt werden.

Inkonsistente Zustände zwischen den Entwicklungsphasen

Die Merkmalsorientierte Softwareentwicklung ist ein phasenorientierter Entwicklungsprozess, der sich an das Domänen-Engineering von CZARNECKI und EISENECKER [CE00], S. 21 anlehnt. Ohne Werkzeugunterstützung müssen viele Informationen über die verschiedenen Phasen des Entwicklungsprozesses redundant eingegeben werden.

1 http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/
http://marketplace.eclipse.org/content/featureide

2 http://www.eclipse.org

Dies verursacht Fehler und inkonsistente Zustände. Ein Beispiel hierfür ist, dass die Informationen der Merkmalsanalyse nicht mit den Design- und Implementierungsphasen im Entwicklungsprozess durch Werkzeugunterstützung verbunden sind. Dadurch werden die bereits definierten Beschränkungen und Abhängigkeiten der Merkmalsanalyse nicht für die Komplexitätsreduktion von Arbeitsschritten der späteren Phasen genutzt. In den darauf folgenden Phasen muss der Programmierer diese Beziehungen manuell integrieren. Des Weiteren ist die Softwareentwicklung selten ein zyklusfreier Prozess. Entdeckt beispielsweise ein Entwickler in der Implementierungsphase eine weitere Einschränkung bezüglich der Kombination zweier Merkmale, so muss das Ergebnis eigentlich in die Analysephase zurückpropagiert werden. Auf Basis von vollständigen Modellen der Analysephase ist es besser möglich, den Konfigurationsprozess zu unterstützen.

Projektverwaltung

Ein zentrales Element einer Entwicklungsumgebung ist die Projektverwaltung mit allen benötigten Einstellungen, wie etwa *Compilerflags*, Ausführungsumgebungen, Umgebungsvariablen, Verzeichnisverwaltung und Quelltextorganisation. Auf Grund der vielen Varianten, die durch unseren Produktlinienansatz erzeugt werden können, gilt es, hierfür auch eine Projektverwaltung einzubauen.

6.1.2 Problem der Merkmalsanalyse

Die Aufgabe der Merkmalsanalyse ist es, mit verschiedenen Stakeholdern die Variationspunkte einer Software zu bestimmen. Hier trägt die Unterstützung der Domänenanalyse durch eine graphische Repräsentation der Merkmalsdiagramme in verschiedenen Darstellungsformen zu einem besseren Verständnis bei. Die klassische Sicht nach KANG et al. [KCH+90] ist am besten für die Kommunikation mit den Endanwendern geeignet. Baumansichten in Form von erweiterten Ordnerstrukturen sind häufig dann geeignet, wenn die Produktlinien sehr viele Variationspunkte enthalten. Für den Austausch mit anderen Systemen und zur automatischen Validierung wird eine textuelle Notation benötigt.

6.1.3 Probleme bei der Merkmalsorientierten Programmierung

Das Lesen oder Schreiben von Quelltexten ist noch immer der gebräuchlichste Weg zum Entwickeln und Verstehen von Software. Eine der Hauptaufgaben eines Entwicklungswerkzeuges ist es, den Entwickler möglichst schnell durch Navigation und Sichten an die

richtige Stelle im Quellcode zu führen. Obwohl die Sprachen Java und C++ sowie ihre merkmalsorientierten Erweiterungen Jak und FEATUREC++ sich nur in einer Handvoll von Schlüsselwörtern unterscheiden, ist es immer noch eine große Herausforderung, sie zu benutzen. In klassischen für die Objektorientierte Programmierung ausgerichteten Entwicklungsumgebungen ist die Klassenansicht das dominierende Navigations- und Darstellungselement. Im Gegensatz dazu liegt das Hauptinteresse in der Merkmalsorientierten Programmierung auf den Merkmalen und ihren Rollen. Merkmale beinhalten verschiedene Software-Artefakte, die etwas zur Funktionalität beisteuern [3] Im Folgenden stellen wir die konkreten Probleme bei der Implementierung von merkmalsorientierten Quelltexten vor:

Quelltextnavigation: Merkmale interagieren häufig auf Quelltextebene miteinander. Aus diesen Interaktionen resultieren Abhängigkeiten zwischen verschiedenen Merkmalen in der Weise, dass Merkmale andere Merkmale ausschließen oder benötigen. In der AHEAD-Tool-Suite oder in FEATUREC++ haben Merkmale keine direkte textliche Präsenz im Quelltext. Dies verhindert, dass Abhängigkeiten explizit für den Entwickler auf Quelltextebene dargestellt werden. Der Entwickler benötigt zur Kompensation eine andere Sicht, die diese Abhängigkeiten darstellt. Die Merkmalsorientierte Programmierung erweitert die Dekomposition einer Software zusätzlich um eine Dimension. Die Software wird für den Entwickler nicht nur in Klassen aufgeteilt, sondern orthogonal hierzu erhält der Entwickler noch eine Trennung des Quellcodes in die einzelnen Merkmale. Dies führt in sehr feingranularen Dekompositionen zu sehr kleinen, auf viele Merkmale verteilten Quellcodefragmenten. Fokus und Kontext-Techniken [PBS93] aus dem Bereich der Softwarevisualisierung und verschiedene Sichten können helfen, diese Fragmente besser darzustellen. Des Weiteren haben wir bei unseren Entwicklungen festgestellt, dass typische Mechanismen wie eine Quellcodeergänzung bei der Entwicklung von Merkmalen unerlässlich sind.

Programmierung, Debugging und Errorhandling: Standardfunktionalitäten für die Objektorientierte Programmierung, wie sie *Eclipse* für Java oder C++ bietet, zum Beispiel Vervollständigung des Quellcodes, *on-the-fly* Syntaxprüfungen, Klassennavigation und eine *Debugging*-Hilfe, sind für die Merkmalsorientierte Programmie-

3 Obwohl die Merkmalsorientierte Softwareentwicklung das Verfeinern anderer Fragmente wie UML-Diagrammen und Dokumentationen erlaubt, konzentrieren wir uns in diesem Kapitel auf Implementierungseinheiten, das heißt auf Klassen.

rung nicht verfügbar. Des Weiteren müssen Fehler aus dem transformierten C++ Quelltext in die einzelnen Merkmale zurückpropagiert werden.

6.1.4 Probleme im Konfigurationsprozess

Wie unser Produktlinienansatz in der explorativen Studie zeigt, können im Bereich des Nanodatenmanagements schnell Produktlinien mit mehr als 100 Variationspunkten entstehen. Zu den typischen Beschränkungen in einem Merkmalsdiagramm ergeben sich weitere Beschränkungen durch *Design Rule Checks*. Die bisherigen Werkzeuge der AHEAD-Tool-Suite erlauben nach der Auswahl der Merkmale eine Überprüfung, ob diese Konfiguration gültig ist. Bei mehr als 100 Merkmalen mit zusätzlichen Beschränkungen ist dieser Prozess ungeeignet.

6.2 Umsetzung in FEATUREIDE

Auf Grund der guten Erweiterbarkeit, der recht großen plattformübergreifenden Verbreitung und des Open-Source-Ansatzes von Eclipse haben wir FEATUREIDE als *Eclipse Plug-Ins* implementiert. Fast alle Komponenten der *Eclipse-Workbench* sind in Form von Erweiterungspunkten in einem Framework (*Framewoks Extension Points*) erweiter- und anpassbar gestaltet. In den folgenden Abschnitten werden die Umsetzungen in FEATUREIDE der in vorangegangenen Abschnitten beschriebenen Probleme vorgestellt.

6.2.1 Umsetzung der Entwicklungsprozessunterstützung

Ein Hauptziel von FEATUREIDE ist die Bewältigung der Komplexität des merkmalsorientierten Entwicklungsprozesses. Hierzu verwenden wir Visualisierungs- und Interaktionstechniken, zum Beispiel *Detail und Overview*, *Detail on Demand* und graphische Hinweise. Ein Großteil der Funktionalität ist ähnlich der Standard-Java- und C++ Entwicklungsunterstützung in *Eclipse* umgesetzt. Dies haben wir im Hinblick auf eine bessere Akzeptanz der Merkmalsorientierten Programmierung getan, das zu einer besseren Weiterverbreitung des Programmierungsparadigmas dienen soll.

Vermeidung inkonsistenter Entwicklungsschritte

Die Verhinderung inkonsistenter Zustände wird durch *Mapping-Funktionen* erzwungen, die alle Phasen des Entwicklungsprozesses miteinander verbinden. Die zentralen Elemente dieser Mapping-Funktionen sind die Merkmale und deren Beziehungen sowie

Beschränkungen. Dadurch werden alle Ergebnisse der Merkmalsanalyse in die Designphase propagiert und alle Merkmale werden auf Kollaborationen abgebildet. Diese Propagierung hilft dem Programmierer, eine konkrete Verfeinerungsreihenfolge der Kollaborationen während des Entwurfsprozesses zu definieren. Des Weiteren werden zusätzliche Einschränkungen auf die korrespondierenden Schichten übertragen. Wenn sich beispielsweise zwei Merkmale ausschließen, ist diese Beziehung auch in den Folgephasen bekannt (zum Beispiel implementierte Schichten). Somit ist es nicht erlaubt, die Funktionalität dieser Schichten zu benutzen oder zu verfeinern. Als ein Ergebnis der Entwurfsphase werden die Beziehungen zwischen den Merkmalen zunehmend erweitert. Unter Nutzung dieser Information wird die Implementierungsstruktur, zum Beispiel Dateiordner, Implementierungsdateien und *Design Rule Check*-Dateien, generiert. Trotz der ausgedehnteren Planungsphase in der Produktlinienentwicklung ist der Entwicklungsprozess selten geradlinig. Üblicherweise ergeben sich häufig während der Entwurfs- und Implementierungsphase bisher nicht berücksichtigte Einschränkungen, die nicht konsistent zu existierenden Informationen aus vorhergehenden Phasen sind.

Projektverwaltung

FEATUREIDE bietet eine Reihe von Funktionalität zur Verwaltung von Projekten an, die mit Merkmalsorientierter Programmierung umgesetzt werden. Hierzu integriert die Projektverwaltung elementare Komponenten einer Entwicklungsumgebung wie Dateinavigatoren, *Wizards*, Quelltexteditoren, Komponentenversionsmanagement. So wird beispielsweise der Nutzer beim Anlegen von FEATUREC++ Projekten mit einem *Wizards* durch die notwendigen Parametereinstellungen begleitet. Des Weiteren werden sämtliche Quelltextpfade und die übersetzten Konfigurationen gespeichert. Zusätzlich bietet FEATUREIDE in der Projektverwaltung für sämtliche Varianten, die eine gültige Konfigurationsdatei haben, die Möglichkeit einer periodischen, automatischen Übersetzung. Dies kann helfen, in allen benötigten Konfigurationen schneller Fehler zu finden.

6.2.2 Unterstützung der Merkmalsanalyse

Merkmalsanalyse

Zentrales Element der Merkmalsanalyse ist die graphische Repräsentation. Abbildung 6.1 zeigt den graphischen Merkmalseditor. Des Weiteren ist es möglich, die Variabilität der Merkmalsmodelle textuell zu bearbeiten. Hier repräsentieren wir die Merkmalsmodelle in einem XML-Format.

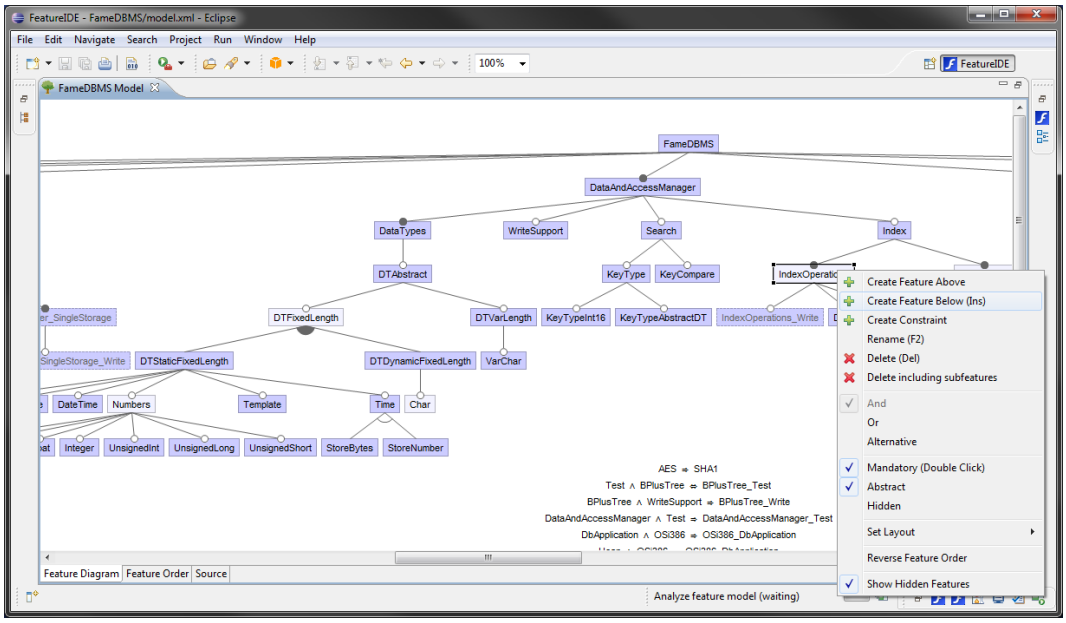

Abbildung 6.1: Merkmalsbaum

Für einen Austausch mit anderen Werkzeugen ermöglichen wir den Im- und Export von Merkmalsmodellen. Hierfür bieten wir beispielsweise Schnittstellen für *GUIDSL* der AHEAD-Tool-Suite [Bat05], *Feature Modeling Plug-In* [AC04], *S.P.L.O.T.* [MBC09] und *SPLConqueror* [SRK+08] an.

Erweiterte Design Rule Checks

Zu den klassischen Merkmalsdiagrammen können auch *Cross-Tree*-Beschränkungen hinzugefügt werden. Diese erweitern die *Design Rule Checks* der AHEAD-Tool-Suite und können mit Hilfe von Editoren (vergleiche hierzu Abbildung 6.2) formuliert werden. Dabei wird sowohl die Syntax der Beschränkungen als auch die Semantik überprüft. So können etwa vergessene Klammern, aber auch unerreichbare Merkmale oder redundante Beschränkungen herausgefiltert werden.

6.2.3 Unterstützung der Programmierung

Das größte Problem bei der Merkmalsorientierten Programmierung ergibt sich aus der Tatsache, dass die Verfeinerungshierarchie durch optionale und alternative Merkmale hoch variabel ist. Prinzipiell erfolgt die Entwicklung von Merkmalen zur Vereinfachung gegen eine gültige Konfiguration. Kontextbasierte Sichten für die Analysen von Abhängigkeiten sind entscheidend für den Entwickler. Im Folgenden werden die wichtigsten

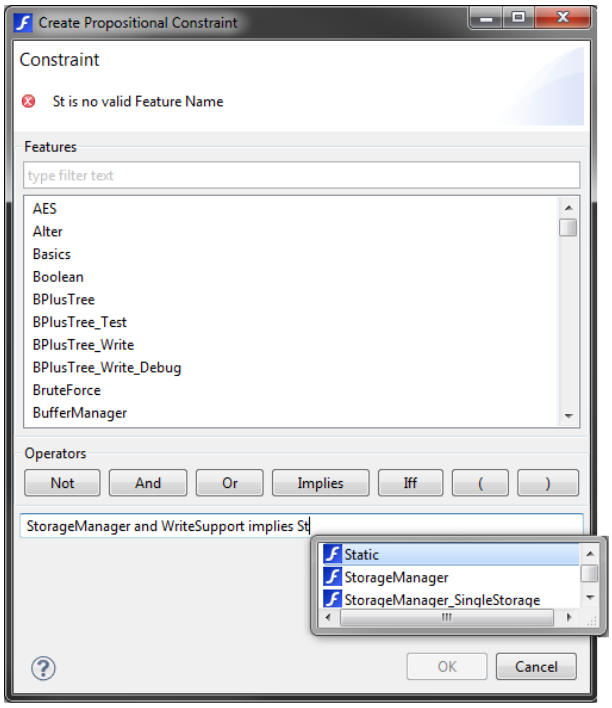

Abbildung 6.2: *Design Rule Checks*-Editor

Funktionen, die die Merkmalsorientierte Programmierung unterstützen, aufgezeigt:

Outline View und Kollaborationsansichten: Zur Überwindung der Komplexität haben wir verschiedene Sichten auf die Implementierungseinheiten benutzt. Mit der Kollaborationsansicht hat der Programmierer eine gute Möglichkeit, einen Überblick über die Merkmale und die Verfeinerungshierarchie der Klassen zu bekommen. Das Kollaborationsdiagramm ist die Hauptansicht, denn sie bietet eine gute Übersicht über die globale Struktur. Des Weiteren ermöglicht diese Ansicht die Navigation durch die Implementierungseinheiten und den Sprung zu den Implementierungen. Verschiedene Filter erlauben dem Nutzer die Éinstellung der Komplexität der Darstellung (Dateiname, Beziehung zu anderen Merkmalen, Verfeinerungsstufe). Abbildung 6.3 zeigt die Kollaborationsansicht.

Syntax-Highlighting und Content Assist: FEATUREIDE erweitert das C++ Syntax-Highlighting um die notwendigen Schlüsselwörter `refines` und `super` von FEATUREC++. Diese Funktionalität steht auch für die Java-Erweiterungen Jak und FEATUREHOUSE zur Verfügung. Des Weiteren ermöglicht der *Content Assist* die automatischen Quelltextergänzungen. Hierbei hilft zur Reduktion der Komplexität

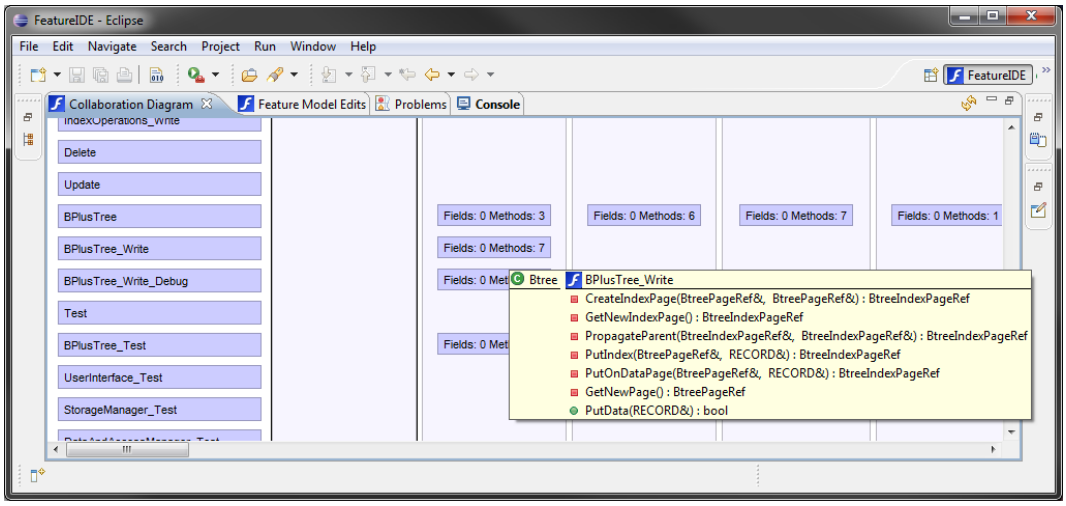

Abbildung 6.3: Zentrale Kollaborationsansicht

bei der Auswahl von Ergänzungsvorschlägen, dass die Implementierung immer gegen aktuell ausgewählte Konfigurationen erfolgt.

Error-Propagation und On-the-fly Errorchecks: Das Zurückpropagieren von Fehlern nach dem Übersetzungsvorgang in das zugehörige Klassenfragment ist für die Praktikabilität der Merkmalsorientierten Programmierung sehr wichtig. Die hierfür notwendige Funktionalität bringt FEATUREC++ in Form von `Line`-Direktiven bei der Transformation mit. FEATUREIDE wertet diese aus und kann damit im Originalquelltext auf die Zeile des Fehlers verweisen. Die Möglichkeit, dass bereits während der Erstellung des Quelltextes Fehler erkannt werden, hilft deutlich Zeit durch das ständige Übersetzen zu sparen. Hierzu muss die unterliegende Sprache, also für FEATUREC++ C++ eine inkrementelle Übersetzung bieten. Hierfür bietet *Eclipse* das *Development Tooling (CDT)* an. Durch die ständige inkrementelle Übersetzung können Syntaxfehler zeitnah angezeigt werden.

6.2.4 Unterstützung des Konfigurationsprozesses

Merkmalsdiagramme bieten eine abstrakte und intuitive Darstellung der Variationspunkte einer Produktlinie. Deshalb sind diese Diagramme der perfekte Ausgangspunkt zur Verbesserung des Konfigurationsprozesses. Abbildung 6.4 zeigt die zwei möglichen Ansichten, die für eine Konfiguration notwendig sind.

Konfigurationseditor

Wir haben festgestellt, dass gerade in größeren Produktlinien die Baumansicht in Form einer erweiterten Dateiansicht eine schnelle Konfiguration ermöglicht. Hierzu hilft der

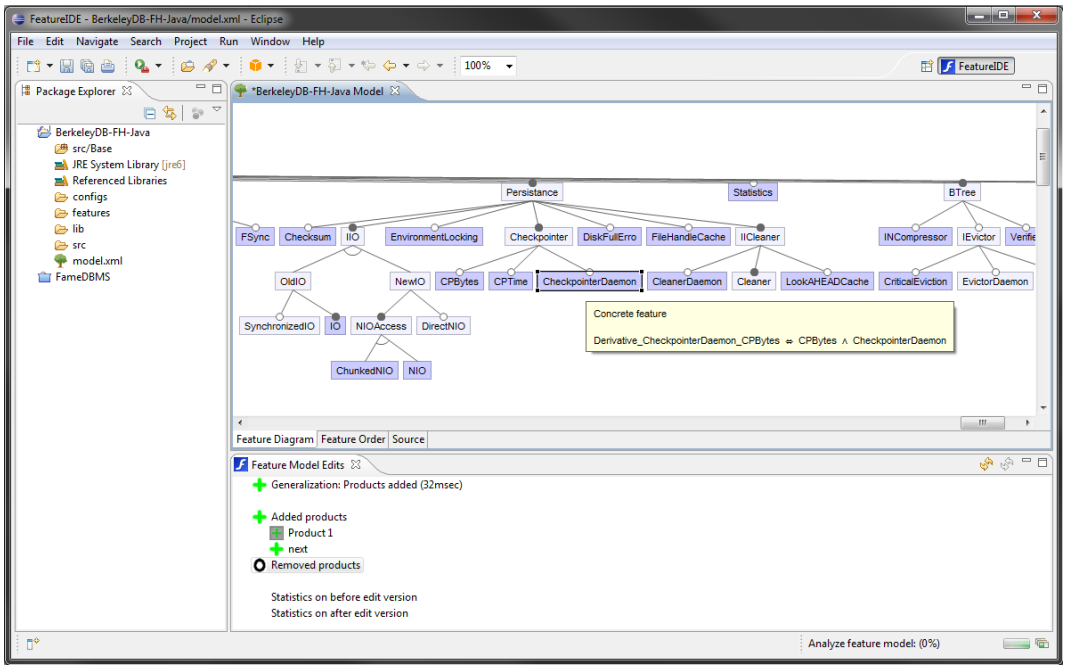

Abbildung 6.4: Merkmalsauswahl

Konfigurationseditor bei der Erstellung von erlaubten Kombinationen, indem ihm nur die gültigen Merkmale zur Auswahl angeboten werden. Wählt der Nutzer beispielsweise von einem alternativen Merkmal eine Variante aus, so werden die anderen alternativen Merkmale bei der weiteren Konfiguration für eine Auswahl ausgeschlossen. Wird ein Merkmal ausgewählt, das auf Grund zusätzlicher Beziehungen weitere Merkmale benötigt, so werden diese automatisch hinzugewählt. Dies kann bei vielen zusätzlichen *Cross-Tree*-Referenzen viel Zeit sparen. In Abbildung 6.5 zeigen wir auf der linken Seite einen einfachen Konfigurationseditor. Auf der rechten Seite der Abbildung präsentieren wir einen erweiterten Konfigurationseditor, der auch explizit die Negativauswahl eines Merkmals erlaubt, dass heißt, dass ein bestimmtes Merkmal nicht für eine weitere Konfiguration berücksichtigt werden soll. Dies hilft, den Suchraum in großen Produktlinien zu verkleinern und es verhindert, dass etwa durch eine automatische Auswahl dieses Merkmal angewählt wird.

6.3 Verwandte Arbeiten

Im Bereich der Merkmalsmodellierung und -konfiguration gibt es im kommerziellen Bereich zwei dominierende Werkzeuge. Dies ist zum einen *Pure::variants* von der Firma

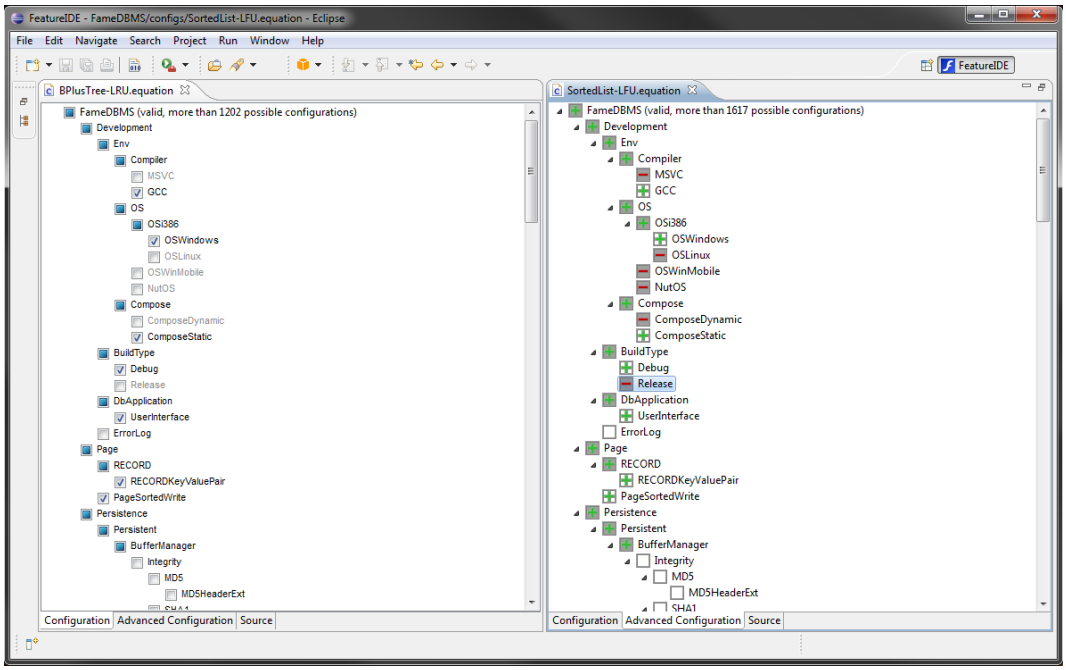

Abbildung 6.5: Merkmalsauswahl mit einfacher (links) und erweiterter (rechts) Funktionalität

pure::systems[4] [BPSP03] und *Gears* von der Firma *Big Lever*[5]. Beide Werkzeuge unterstützen sowohl die Modellierung als auch die Konfiguration, aber nur sehr rudimentär die Implementierungsphase. Erweiterungen für die Merkmalsorientierte Programmierung sind nicht vorgesehen und auf Grund von Software-Lizenzbedingungen wenig attraktiv. Im *Open-Source* Umfeld sind *KConfig*[6] und CDL[7] weiter verbreitet. Beide Ansätze unterstützen eine textuelle Notation zur Modellierung und Konfiguration von Variabilität und ermöglichen so die Beschreibung gültiger Varianten.

6.4 Zusammenfassung

In diesem Kapitel wurde die Entwicklung Werkzeugunterstützung FEATUREIDE vorgestellt. Dabei integrierten wir FEATUREIDE in Form von mehreren *Eclipse Plug-Ins* in das *Eclipse-Framework*. FEATUREIDE bietet neben einer Prozessunterstützung, die vor allem einen automatisierten und damit einen konsistenten Informationsaustausch zwischen den Entwicklungsphasen des Domänen-Engineering sicherstellt, auch Hilfen auf der Im-

4 http://www.pure-systems.com
5 http://www.biglever.com
6 http://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt
7 http://www.gaisler.com/doc/ecos-2. 0-cdl-guide-a4.pdf

plementierunsgsebene, wie etwa Quelltextergänzungen oder spezielle Sichten, die dem Kollaborationenentwurfsgedanken genügen. Des Weiteren wird durch FEATUREIDE der Konfigurationsprozess unterstützt.

Weitere Einsatzgebiete

Neben der Anwendung zur Unterstützung der Merkmalsorientierten Programmierung für das Umfeld vom Nanodatenmanagement für eingebettete Systeme hat sich FEATUREIDE zum einem Standardwerkzeug im Bereich der Lehre, insbesondere in Produktlinienveranstaltungen, etabliert. FEATUREIDE wird derzeitig unter anderem in Austin, Magdeburg, Marburg, Namur, Passau, Santa Cruz und Torino eingesetzt.
Neben den schon zahlreich in FEATUREIDE integrierten Produktlinienansätzen (wie zum Beispiel: AHEAD, FEATUREHOUSE, FEATUREC++, DeltaJ [SBB+10] und AspectJ) wollen wir zukünftig FEATUREIDE auch für AspectC++, AspectC#, Hyper/J und dem CPP anbieten.

KAPITEL 7

Studien: Datenmanagement für eingebettete Systeme

*Dieses Kapitel enthält Ergebnisse der Veröffentlichungen des BTW-Workshops 2007 „Konfigurierbarkeit für ressourceneffiziente Datenhaltung in eingebetteten Systemen am Beispiel von Berkeley DB“(*ROSENMÜLLER, LEICH, APEL *[RLA07]), des DKE-Journals 2009 „Tailor-Made Data Management for Embedded Systems: A Case Study on Berkeley DB„ (*ROSENMÜLLER, APEL, LEICH, SAAKE *[RALS09]), des SETMDM-Workshops 2008 „*FAME*DBMS: Tailor-made Data Management Solutions for Embedded Systems„(*ROSENMÜLLER, SIEGMUND, SCHIRMEIER, SINCERO, APEL, LEICH, SPINCZYK, SAAKE *[RSS+08]), des FOSD-Workshops 2009 „*ROBBY*DBMS - A Case Study on Hardware/Software Product Line Engineering„ (*LIEBIG, APEL, LENGAUER, LEICH *[LALL09]), des Datenbank-Spektrums 2007 „ Von Mini- über Micro- bis zu Nano-DBMS: Datenhaltung in eingebetteten Systemen“(*ROSENMÜLLER, LEICH, APEL, SAAKE *[RLAS07]) und des ACPIS Workshops 2007 „Highly Configurable Transaction Management for Embedded Systems“ (*PUKALL, LEICH, KUHLEMANN, ROSENMÜLLER *[PLKR07]). Implementierungsarbeiten und Messdatenerhebungen wurden von* PUKALL *„FAME-DBMS: Entwurf ausgewählter Aspekte der Transaktionsverwaltung“ [Puk06],* GITTLER *„Merkmalsorientierte Refaktorisierung von Berkeley DB“ [Git08],* LIEBIG *„Untersuchung der Anwendung erweiterter Programmierparadigmen für die Programmierung eingebetteter Systeme“ [Lie08] und* KUNZE *„Untersuchung und Nutzung mehrdimensionaler Indexstrukturen in einem Datenbankmanagementsystem für tief eingebettete Systeme“ [Kun11] im Rahmen dieser Dissertation betreuten Abschlussarbeiten vorgenommen.*

In diesem Kapitel stellen wir die Ergebnisse der Evaluierung unseres Sprachansatzes FEATUREC++ in der Domäne der eingebetteten Datenmanagementsysteme vor. Im Fokus des Kapitels stehen drei Fallstudien. Die erste Fallstudie zeigt unter „Laborbedingungen“ einen neuen Entwurf der FAMEDBMS-Produktlinie und zwei Erweiterungen dieser grundlegenden Basis. Das Anwendungsgebiet wurde im Wesentlichen auf Grundlage einer akademisch dominierten Analyse der Domäne untersucht. Die zweite Fallstudie präsentiert die Entwicklung einer Produktlinie auf Basis von FAMEDBMS an einer konkreten Anwendungsdomäne. Das Anwendungsszenario entstammt dem Bereich der mobilen autonomen Roboter. Die Produktlinie ROBBYDBMS zeigt dabei die Anwendbarkeit in einem extrem ressourcenbeschränkten Anwendungsgebiet. In der dritten Fallstudie wird an einem in der Praxis etablierten DBMS, Berkeley DB, die Praktikabilität des Ansatzes untersucht und anhand eines Performanztests evaluiert. Das Kapitel gliedert sich wie folgt:

FAMEDBMS: Zunächst werden Ergebnisse der FAMEDBMS-Fallstudie auf Basis von FEATUREC++ präsentiert. Das Hauptziel der FAMEDBMS-Fallstudie besteht darin aufzuzeigen, dass eine feingranulare Umsetzung von Datenmanagementfunktionalität mit FEATUREC++ möglich ist. Hierauf aufbauend werden zwei Erweiterungen von FAMEDBMS vorgestellt. In der ersten Erweiterung haben wir FAMEDBMS um variable Transaktionsverwaltungsfunktionalität erweitert. Diese gilt als eine nur schwer zu kapselnde Funktionalität, da sie stark mit anderer Funktionalität im Datenmanagementbereich interagiert. Die zweite Erweiterung ist ein Anfrageoptimierer. Die Anfrageoptimierung ist ebenfalls stark mit der restlichen Funktionalität des Systems verbunden.

ROBBYDBMS: Im Fokus der ROBBYDBMS-Fallstudie stehen Untersuchungen, ob der Einsatz von FEATUREC++ bei der Entwicklung von Datenmanagementfunktionalität im Bereich der tief eingebetteten Systeme möglich ist. Wir gehen davon aus, dass das gewählte Szenario, das dem Bereich des Nanodatenmanagements zugeordnet werden kann, die untere Schranke des Einsatzumfeldes von Datenmanagementsoftware darstellt, die mit FEATUREC++ zu entwickeln ist. ROBBYDBMS zeigt dabei eine extrem auf *Footprint* optimierte und angepasste Erweiterung von FAMEDBMS auf, die auf einem ressourcenbeschränkten eingebetteten Robotersystem zum Einsatz kommt. Die Programmgröße des Datenmanagementcodes wird in dieser Studie auf einen Bereich von 1 bis 20 KB, je nach konfigurierten Merkmalen, skaliert.

Berkeley DB: Während die beiden vorangegangenen Studien auf kompletten Neuentwicklungen basieren und im Funktionsumfang sowie in der Variabilität kaum vergleichbar zu anderen Systemen sind, soll diese Fallstudie einen Vergleich zu einem kommerziell eingesetzten DBMS bieten. Ziel der Studie ist es aufzuzeigen, dass sich die Leistungseigenschaften hinsichtlich der Performanz und des Programmcodes durch den Einsatz von FEATUREC++ nicht verschlechtern. Im Gegensatz dazu wollen wir sogar zeigen, dass sich durch die gezielte Auskonfiguration von Merkmalen, die nicht benötigt werden, das Systemverhalten bezüglich der Performanz und Programmgröße sogar messbar positiv entwickelt. Hierzu verwenden wir einen Benchmark, den die Entwickler von Berkeley DB eigens zu Demonstrationszwecken der Leistungsfähigkeit ihres Systems entwickelt haben. In dieser Fallstudie gilt es aufzuzeigen, dass eine stärkere Modularisierung durch FEATUREC++ keine negative Auswirkung auf die Performanz und die Programmcodegröße hat.

Fokussierug der Fallstudien hinsichtlich der Ziele

Ausgerichtet an den grob formulierten Zielsetzungen der Arbeit, die im Abschnitt 2.6 vorgestellt wurden, werden unsere Werkzeugansätze FEATUREC++ und FEATUREIDE hinsichtlich dieser Ziele evaluiert. Eine eins zu eins Zuordnung der drei folgenden Ziele der Arbeit zu den drei Fallstudien ist nicht möglich, wie die Abbildung 7.1 veranschaulicht.

1. Domänenanalyse im Umfeld des Nanodatenmanagements
2. Software-Engineering für eingebettete Systeme
3. Ressourcenoptimierung

Ausgehend von unserer explorativen Studie werden wir die genannten Ziele hinsichtlich der Problemfelder Granularität, *Overhead*, Wiederverwendbarkeit, querschneidende Belange, Automatisierbarkeit der Konfiguration, Praktikabilität und Umsetzung von ungeplanten Änderungen verfeinert untersuchen.

7.1 1. Fallstudie: FAMEDBMS

Gesamtprojekt FAMEDBMS

Die Nachfolgeversion des im Kapitel 4 vorgestellten FAMEDBMS-Speichermanagers entstand in einem Gemeinschaftsforschungsprojekt der Otto-von-Guericke-Universität Magdeburg, der Technischen Universität Dortmund sowie der Friedrich-Alexander Universität Erlangen-Nürnberg und wurde darüber hinaus auch von der Universität

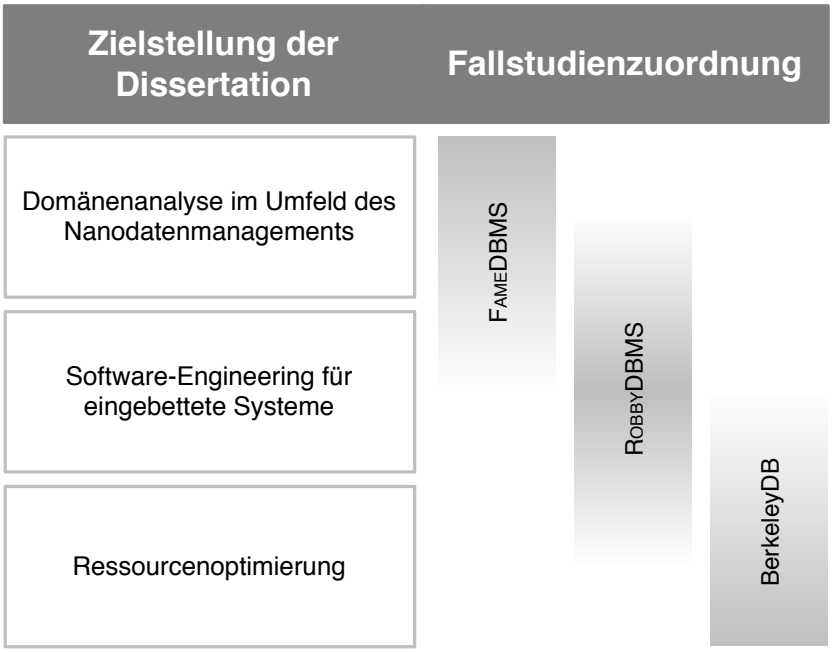

Abbildung 7.1: Zuordnung von Zielen der Dissertation zu den Fallstudien

Passau begleitet. Das Ziel des FAMEDBMS-Projektes war die Entwicklung von Techniken und Werkzeugen zur Analyse, zum Entwurf, zur Implementierung und Anpassung von Datenmanagementfunktionalität und ihrer Infrastruktursoftware sowie ihre Bewertung. Die im Rahmen dieser Dissertation vorgestellten Ergebnisse reflektieren nur den Bereich des Datenmanagements, der für die weiteren Fallstudien in diesem Kapitel relevant ist. Ein großer Teil der im Gemeinschaftsforschungsprojekt bearbeiteten Themen, wie beispielsweise die Werkzeugentwicklung, nicht-funktionale Eigenschaften und die Integration von Betriebssystemfunktionalität, werden in dieser Arbeit nicht betrachtet.

7.1.1 Zielsetzung der Fallstudie

Ziele der FAMEDBMS-*Fallstudie*

Das primäre Ziel dieser Fallstudie ist die Entwicklung von FAMEDBMS als auf der Grundlage einer umfassenden Domänenanalyse im Umfeld des Nanodatenmanagements. Möglichst abstrakt sollen Variationspunkte im Datenmanagementbereich von eingebetteten Systemen untersucht und Abhängigkeiten zwischen verschiedenen Merkmalen analysiert werden. Obwohl der Bereich des Datenmanagements in seinen Grundlagen gut analysiert und ausreichend dokumentiert ist, zeigt die erste Fallstudie eines FAMEDBMS-Speichermanagers, dass die Abhängigkeiten der Merkmale untereinander nur unzureichend bekannt beziehungsweise dokumentiert sind. Gegenüber der ersten

Fallstudie gilt es des Weiteren die Anzahl der optionalen Merkmale zu erhöhen. Ebenfalls werden zwei spezielle Teildomänen im Datenbankumfeld genau untersucht, die wir in der ersten Fallstudie nicht betrachtet haben, um erste Aussagen über zu erwartende Probleme zu erhalten. Diese sind zum einen die Anfrageverarbeitung und zum anderen die Transaktionsverwaltung:

Anfrageverarbeitung: Die Anfrageverarbeitung ist eine der zentralen Komponenten eines klassischen Datenbanksystems, die die Performanz des Systems enorm beeinflusst. Als zentrale Komponente ist die Anfrageverarbeitung von Ad-hoc-Anfragen seit mehreren Jahrzehnten immer wieder Gegenstand von Forschung und Entwicklung (Vergleiche hierzu: *Starburst* [MP87, HFLP89, HS90], *Volcano* [GM93], *Cascades* [Gra95] und *XXL*[BBD+01]). Aus softwaretechnischer Sicht ist die Modularisierung der Anfrageoptimierung auf Grund der vielen Abhängigkeiten zu fast allen anderen Komponenten eines Datenmanagementsystems eine Herausforderung. Auch wenn die variable Anfrageverarbeitung auf Grund des zu erwartenden *Overheads* kein zentraler Bestandteil von Datenmanagementkomponenten im Nanodatenbankenbereich ist, so erhoffen wir uns dennoch Aussagen über die Grenzen unseres Ansatzes. Ziel der hier vorgestellten variablen Anfrageverarbeitung ist es herauszufinden, ob auf Basis einer variablen Anfragesprache (vgl. hierzu ROSENMÜLLER et al. [RKS+09]) beziehungsweise einer variablen Anfrage-API wiederverwendbare variable Strukturen einer Anfrageverarbeitung herausgearbeitet werden können. Des Weiteren gilt es, die Grenzen dieses Ansatzes aufzuzeigen. Ziel ist es, die Machbarkeit, Skalierbarkeit und Praktikabilität unseres Ansatzes im Umfeld der Anfrageverarbeitung zu untersuchen.

Transaktionsverwaltung: Die zweite Teildomäne, die im Umfeld von erweiterbaren Datenmanagementlösungen als problematisch gilt, ist die Transaktionsverwaltung mit *Recovery*-Komponenten. Betrachtet man die klassische 5-Ebenen-Architektur, so können diese Teile der Komponenten allen Ebenen zugeordnet werden. Genau diese starken Abhängigkeiten erschweren die Variabilität enorm. Des Weiteren werden in der Teildomäne der Transaktionsverwaltung viele homogene Erweiterungen vermutet (vgl. hierzu TESANOVIC et. al [TSH04]), die aus unserer Sicht insbesondere den Einsatz vom *Aspectual Mixin Layers* erforderlich machen wird.

7.1.2 Umsetzung FAMEDBMS

Variationspunkte von FAMEDBMS

Die Abbildung 7.2 zeigt die wichtigsten Variationspunkte von FAMEDBMS. Die Anfragebearbeitung und die Transaktionsverwaltung werden dabei nicht detailliert dargestellt, sondern folgen in späteren Abschnitten. Die wesentlichen Komponenten von FAMEDBMS sind der *Storage Manager*, der *Data and Access Manager* und das Merkmal *Development*, das die Abstraktion von Betriebssystemfunktionalität und Kompositionsbeziehungsweise Übersetzungseigenschaften kapselt. Optional kann FAMEDBMS um einen *Transaction Manager* [SRS+09] und um abstrahierte Zugriffsmöglichkeiten erweitert werden. Dabei hat der Nutzer die Möglichkeit, entweder eine einfache *Access-API* oder eine Anfrageverwaltung mit Optimierungskomponenten zu verwenden. Im Folgenden wird der wesentliche Funktionsumfang der einzelnen Merkmale kurz beschrieben:

Storage Manager: Das Merkmal *Storage Manager* organisiert die Variationspunkte des *Buffer Manager* und der *Storage Organisation*. Dabei wird dem Nutzer die Wahl zwischen einer reinen Hauptspeicherdatenbank (*InMemory*) und einer Sekundärspeicherdatenbank (*Persistent*) geboten. Wird eine Sekundärspeicherdatenbank gewählt, so kann ein *BufferManager* umfangreich konfiguriert werden. Der *Buffer Manager* organisiert eine statische oder dynamische Speicherallokierung. Des Weiteren kann die Speicherung betriebssystemabhängig dateibasiert oder nicht dateibasiert erfolgen. Zur Sicherstellung der Integrität können *MD5* oder *SHA1* Signaturen verwendet werden. Ebenfalls ist eine verschlüsselte Speicherung der Daten möglich.

Data and Access Manager: Durch den Zugriffsmanager (*Data and Access Manager*) werden die von FAMEDBMS angebotenen Datentypen (*Data Types*), eine Schreibunterstützung (*Write Support*) und Indexstrukturen (*Index*) sowie Vergleichsoperationen (*Search*) bereitgestellt. Durch das Merkmal *Data Type* werden sowohl Standarddatentypen als auch abstrakte Datentypen (*DTAbstract*) sowie Relationen (*Relation*) bereitgestellt. Insbesondere durch den merkmalsorientierten Kompositionsansatz können die abstrakten Speicherformen wie beispielsweise Listen und Felder in vielen Varianten wiederverwendet und in neue abstrakte Datentypen gekapselt werden.

Abbildung 7.2: Variationspunkte von FAMEDBMS

Als Indexstrukuren bietet FAMEDBMS derzeitig einen B-Baum (*BPlusTree*), eine sortierte Liste (*SortedList*) und einen T-Baum (*TTree*), der insbesondere im Bereich der Hauptspeicherdatenbanken Vorteile bringt [LC86, LNT00]. Des Weiteren kann die Erweiterung des T*-Baums [CK96] in Echtzeitsystemen Vorteile bringen, da Zugriffe im Vorfeld exakt berechnet werden können.

Development: Das Merkmal *Development* kaspselt alle zur Umsetzung relevanten Funktionalitäten, die nicht der Kernfunktionalität des Datenmanagements zugeordnet werden können. Dazu zählen unter anderem Umgebungsparameter für verschiedene Betriebssysteme und Übersetzeranweisungen, die sonst typischerweise durch Präprozessoranweisungen gekapselt werden. FAMEDBMS kann sowohl für Windows und Linux-Systeme verwendet werden (*OSi386*) als auch für *NutOS* und Windows Mobile. Die typische Trennung von *Debug* und *Release* Funktionalität wird ebenfalls durch das Merkmal *Development* gekapselt. Des Weiteren bietet FAMEDBMS die Möglichkeit der statischen und dynamischen Komposition von Merkmalen.

Wiederverwendung von FAMEDBMS

Die Variationspunkte und die Umsetzung von FAMEDBMS auf Grundlage von FEATUREC++ wird von anderen Projekten als Basis verwendet. Hierzu zählen *Celluar DBMS* [uR11], *AutoDaMa* [TSP+11, TSP+12] und unser ROBBYDBMS, das wir im Abschnitt 7.2 genauer vorstellen. Des Weiteren haben wir FAMEDBMS experimentell um variable Transaktionsverwaltungseigenschaften und Anfrageverarbeitungskomponenten erweitert. Diese stellen wir in den folgenden Abschnitten vor.

7.1.3 Transaktionsverwaltung

Transaktionsverwaltung für FAMEDBMS

Der Funktionsumfang einer Transaktionsverwaltungskomponente im Bereich der eingebetteten Systeme variiert stark. So kann beispielsweise eine einfache Transaktionverwaltung die externe Veränderung von Datenobjekten in der Pufferverwaltung bei Lese- und Schreibzugriffen verhindern. Dies ist beispielsweise notwendig, wenn transaktionale Aktionen auf Datenobjekten vorgenommen werden. Im Bereich der eingebetteten Systeme kann dies entweder durch klassisches Sperren oder durch das zeitweise Abschalten von *Interrupts* erreicht werden. Abbildung 7.3 zeigt das Domänenmodell einer Transaktionsverwaltung auf FAMEDBMS-Basis für eingebettete Systeme.

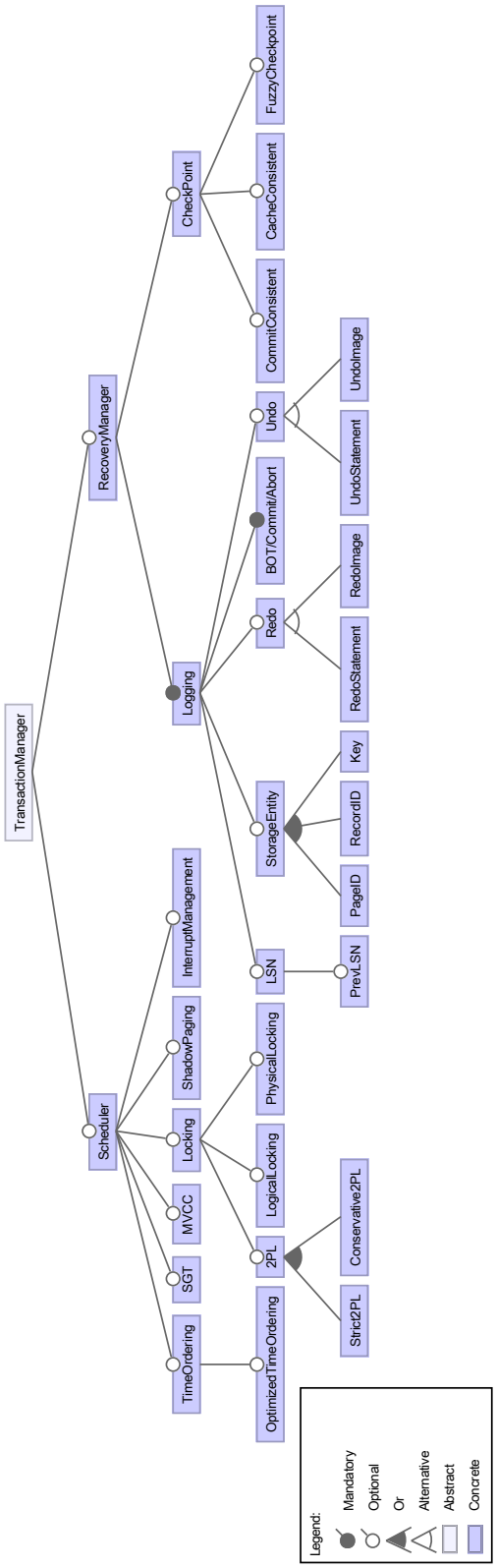

Abbildung 7.3: Variationspunkte des FAMEDBMS-Transaktionsmanagers

Scheduler: Im Merkmal *Scheduler* werden die Serialisierbarkeit und Verfahren zur Rücksetzbarkeit organisiert. Für die Serialisierungen können zum einen das einfache (*Time Ordering*) und das erweiterte Zeitmarkenverfahren (*Optimized Time Ordering*) verwendet werden. Zum anderen kann ein Serialisierbarkeitsgraphentester (*SGT*) gewählt werden. Für extrem ressourcenbeschränkte Umgebungen kann eine Serialisierung durch ein einfaches Interruptmanagement (*Interrupt Management*) realisiert werden. Des Weiteren kann ein Multiversionskontrollprotokoll (engl. *Multiversion Concurrency Control* (MVCC)) zur Entkopplung von lesenden und schreibenden Transaktionen ausgewählt werden. Dies führt insbesondere bei verhältnismäßig wenig schreibenden Transaktionen zu einem besseren Durchsatz. Die Rücksetzbarkeit kann mit verschiedenen Varianten des Zwei-Phasen-Sperrprotokolls *2PL* (engl. *Two Phase Locking* (2PL)), dem *S2PL* (engl. *Strict Two Phase Locking* (S2PL)) und dem *C2PL* (engl. *Conservative Two Phase Locking* (C2PL)) erreicht werden. Generell kann das Sperren sowohl auf logischen (*Logical Locking*) als auch auf physischen (*Physical Locking*) Objekten erfolgen.

Recovery Manager: Die Datenwiederherstellung (*Recovery Manager*) wird im Fehlerfall durch die Protokollierung von Daten organisiert. Das Merkmal *Storage Entitiy* ermöglicht dabei die Festsetzung des Granularitätslevels für die Wiederherstellung. Die Wiederherstellung kann physisch auf Seitenebene (*Redo Images*) oder auf Basis der logischen Wiederholung der Anweisungen (*Redo Statement*) erfolgen. Zur Unterstützung der Wiederherstellung können Sicherungspunkte (*Check Points*) konfiguriert werden. Hierbei kann zwischen transaktionskonsistenten (*Commit Consistent*), aktionskonsistenten(*Cache Consistent*) und unscharfen Sicherungspunkten (*Fuzzy Check Points*) gewählt werden.

Implementierung und Ergebnisse

Umsetzung der Transaktionsverwaltung

Die Umsetzung der Transaktionsverwaltung auf Basis einer im maximalen Funktionsumfang bekannten Struktur ist weit weniger schwierig als am Anfang angenommen wurde. Die Fähigkeiten heterogener Erweiterungen, die durch die Merkmalsorientierte Programmierung in FEATUREC++ bereitgestellt werden, reichen in den meisten Fällen aus, einen variablen Speichermanager um Transaktionsfähigkeit zu erweitern. Bis auf wenige Ausnahmen konnten bei der Erweiterung von FAMEDBMS kaum homogene Erweiterungen festgestellt werden. Eines der Beispiele für eine klassische aspektorientierte Lösung war die Einbringung eines Logging-Mechanismus für das *Recovery*-Verfahren. Abbildung 7.4 zeigt die Umsetzung der *Logging*-Funktionalität in die verschiedenen Sperrmechanismen des *Schedulers*. Alternative Lösungen zu diesen Varianten, die weit

weniger homogene Erweiterungen benötigt hätten, wären möglich gewesen, hätten aber im klassischen Kollaborationsentwurf zusätzliche Indirektionen bedeutet. Des Weiteren konnten die Aspekte in diesem Zusammenhang für eine feingranularere Erweiterung verwendet werden. Implizit wurden die optionalen Merkmalsinteraktionen auch in diesem Fall durch den Aspektmechanismus gekapselt.

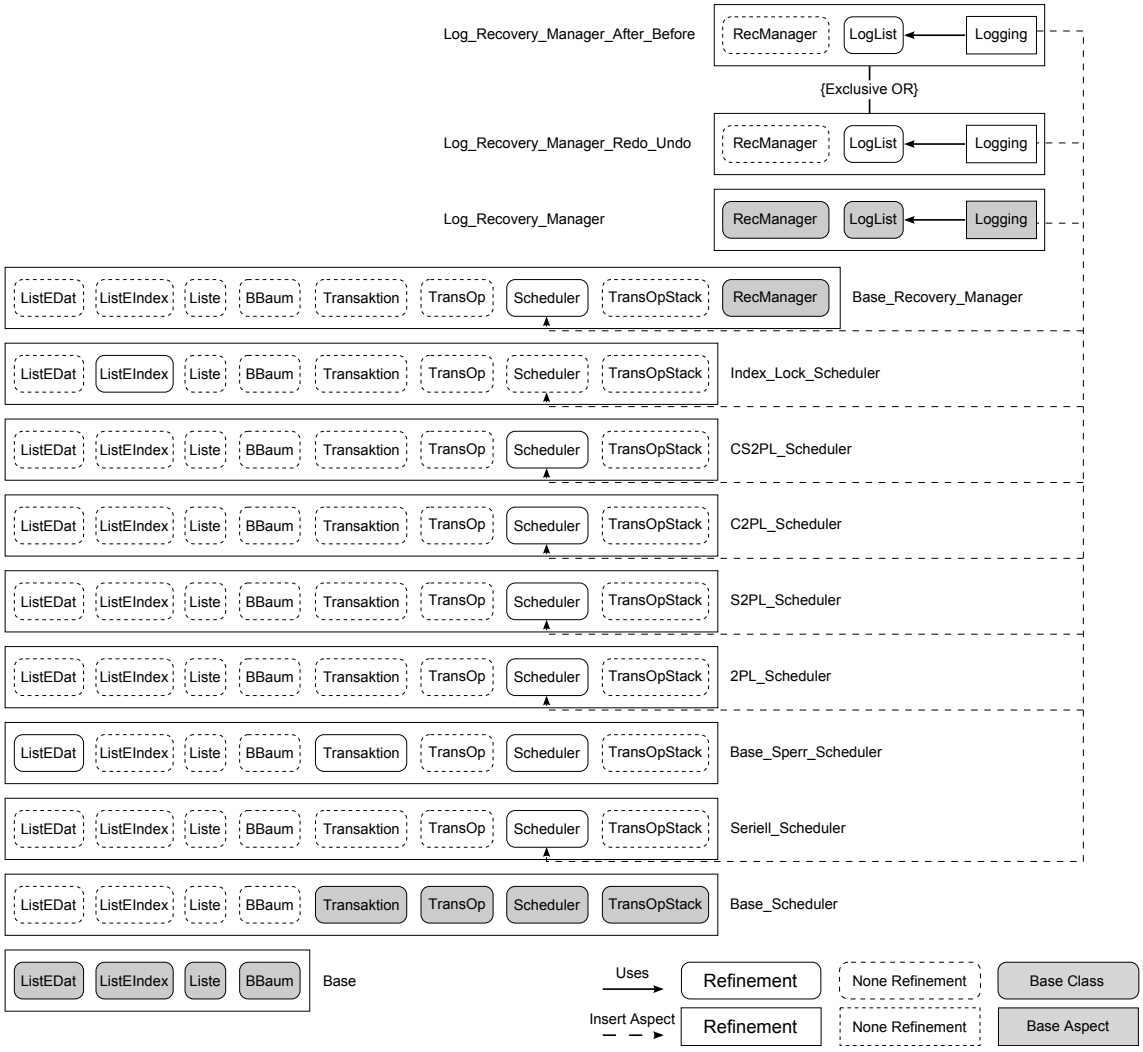

Abbildung 7.4: Umsetzung einer Transaktionsverwaltung mittels *Aspectual Mixin Layers* PUKALL [Puk06] (verändert)

Ergebnisse

Während aus unserer Sicht die *Aspectual Mixin Layers* wenig für die interne Implementierung variabler Transaktionsverwaltungsfunktionalität verwendet worden sind, ist in diesem Zusammenhang ein interessantes Einsatzgebiet für unsere neue Art der Implementierung die Anbindung des Anwendungsprogramms an die Transaktionsverwaltungsfunktionalität. *Aspectual Mixin Layer* können hierbei insbesondere für die voll transparente Integration von Transaktionsverwaltungsfunktionalität in der Anwendung in Form einer Transaktions-API verwendet werden. In Ansätzen bietet COMET DBMS [NNT+04, TNH+03, NTNH03] Aspekte zur Integration in das Anwendungsprogramm. GROPENGIESSER et al. [GS11] zeigen wie Aspekte der Transaktionsverwaltungsfunktionalität in *Cloud*-Anwendungen integriert werden können. SCHRÖTER [Sch12] erweitert diesen Ansatz mit *Aspectual Mixin Layer*. Im Zusammenhang mit *Aspect Refinements* kann hier die domänenspezifische Wiederverwendung von Transaktionsanbindungen realisiert werden.

7.1.4 Anfrageverarbeitung und -optimierung

Variationspunkte einer Anfrageverarbeitung und -optimierung

Die Entwicklung einer variablen Anfrageoptimierung ist komplex. Dies liegt schon an der Interaktion der Merkmale auf semantischer Ebene. Auf Basis unserer variablen Struktur mit vielen optionalen Merkmalen des Speichermanagers gibt es in den verfeinernden Merkmalen der Anfrageverarbeitung viele Interaktionen mit optionalen oder alternativen Merkmalen des zugrundeliegenden Speichermanagers. Zur Vereinfachung wurde in unserer Fallstudie zunächst eine fixe Struktur, also eine konkrete Konfiguration, eines Speichermanagers gewählt. Abbildung 7.5 zeigt den ersten Vorschlag für Variationspunkte einer einfachen Anfrageverarbeitung mit einem Optimierer. Dabei wurden die Variationspunkte wie folgt gesetzt:

Query Prozessor: Im Merkmal *Query Prozessor* werden die Variationspunkte zur Übersetzung (*Translation*), zum Sprachumfang von *SQL*, zu den anfrageverarbeitungsbezogenen Katalogdaten (*Data Dictionary*) und zur internen Repräsentation von Anfragen (*Internal Query Representation*) organisiert. Dabei ist es möglich, dass der Entwickler auf eine Ad-hoc-Übersetzung seiner Anfrage verzichtet und nur auf physische Operationen zurückgreift, um den *Overhead* der Übersetzung und der logischen Anfrageoptimierung zu eliminieren. Dies ist allerdings nur bei statischen Anfragen möglich. Hierfür muss der Entwickler dann die Anfragen mit Hilfe der Elemente zur internen (physischen) Anfragerepräsentation formulieren. Die Hauptmerkmale der Anfrageverarbeitung können wie folgt weiter variiert werden:

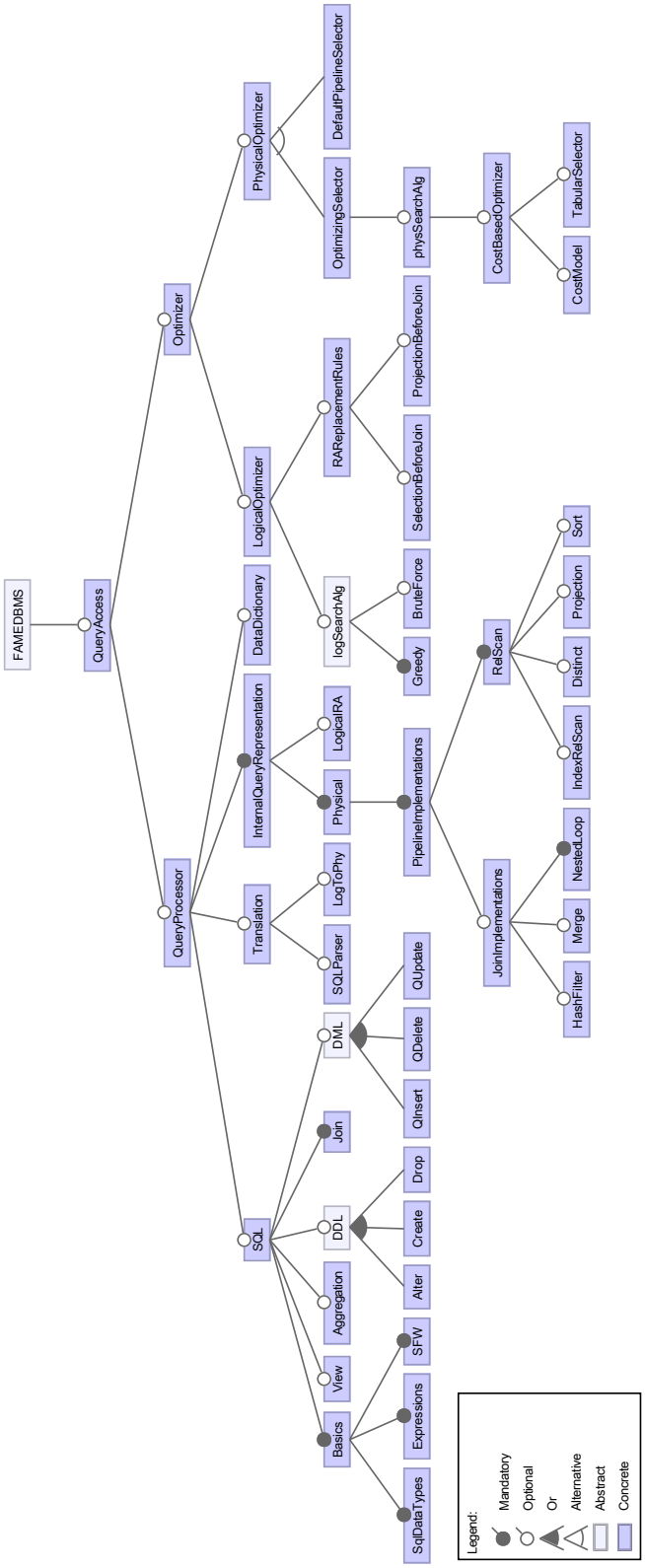

Abbildung 7.5: Variationspunkte eines einfachen Optimierers

SQL: *SQL* ist in seinem Standardfunktionsumfang für die meisten Anwendungen im Bereich der eingebetteten Systeme überdimensioniert. Aus diesem Grund bietet das Merkmal *SQL* eine Basisvariante (*Basics*) mit einem auf eine Relation beschränkten Funktionsumfang. Optional können Verbundoperationen (*Join*), Sichten (*View*) und Aggregationen (*Aggregation*) zur Basisfunktionalität erweitert werden. Des Weiteren können Datenmanipulationsoperationen (*DML*), (*QInsert*), (*QDelete*) und (*QUpdate*) hinzugefügt werden. Bei Unterstützung variabler Schemata können Relationen angelegt (*Create*), erweitert (*Alter*) und gelöscht (*Drop*) werden. Erweiterungen, wie beispielsweise die Anfrageverarbeitung von Datenströmen, sind derzeitig nicht in FAMEDBMS integriert. Eine ausführlichere Diskussion zur variablen Gestaltung der Anfragesprache SQL führen wir in ROSENMÜLLER et al. [RKS+09].

Translation: Je nach Konfiguration kann es zur Laufzeit notwendig sein, dass Ad-hoc-Anfragen übersetzt und in eine logische Repräsentation überführt werden müssen. Dies ist Aufgabe des Merkmals *SQLParser*. Der Parser übersetzt in Abhängigkeit des unterstützten SQL-Umfangs die Anfragen. In einem zweiten Schritt werden diese logischen Repräsentationen der Anfragen durch das Merkmal (*LogToPhy*) in physische Präsentationen transformiert.

Internal Query Representation: Das Merkmal *Internal Query Representation* organisiert die Basis der logischen (*Logical RA*)[1] und physischen (*Physical*) Repräsentation der Anfragen. Die Basis der physischen Anfragerepräsentation bildet ein „Relationen-Scan“ (*RelScan*), der optional zum „Index-Relationen-Scan“ (*IndexRelScan*) mit Duplikateliminierung (*Distinct*), Sortierfunktionalität (*Sort*) oder Projektionseigenschaften (*Projection*) erweitert werden kann. Darüber hinaus können, wenn in der Anfragesprache verwendet, Verbundimplementierungen (*Join Implementation*) konfiguriert werden. Die Basis der Verbundimplementierung bildet verschachtelte Schleifen (*Nested Loop*). Optional können die Substitute des sortierten Verbundes (*Merge*) oder des *Hash*-Filterverbundes (*Hash Filter*) hinzugenommen werden.

Data Dictionary: Zentraler Bestandteil einer Anfrageoptimierung ist der Katalog (*DataDictionary*). In Abhängigkeit der Konfiguration wird der Funktionsumfang des Kataloges bestimmt.

Optimizer: Das Merkmal *Optimizer* organisiert die Variationspunkte des Anfrageoptimierers. Wird ein Optimierer verwendet, besitzt er zwingend einen physischen

1 Die Basis hierfür wird durch die Relationenalgebra gebildet.

Optimierer (*Physical Optimizer*). Optional kann noch ein logischer Optimierer (*Logical Optimizer*) hinzugewählt werden.

Logical Optimizer: Für den logischen Optimierer (*Logical Optimizer*) werden zum einen auf Basis der Relationenalgebra Termersetzungsregeln (*RA Replacement Rules*) angeboten und zum anderen Optimierungsstrategien (*LogSearchAlg*) zum Aufbau des Suchraumes bereitgestellt. Als Optimierungsstrategie steht in der Basis der *Greedy*-Ansatz zur Verfügung. Dieser kann dann mit einer *BruteForce*-Strategie erweitert werden.

Physical Optimizer: Die Aufgabe der physischen Optimierung ist die Erzeugung und Optimierung von physischen Ausführungsplänen auf Basis von *Pipelining*-Operationen. Der physische Optimierer (*Physical Optimizer*) bietet die Möglichkeit einer einfachen Vorauswahl (*Default Pipeline Selector*) eines gültigen Anfrageplanes. Des Weiteren gibt es die Möglichkeit, physische Anfragepläne auf Basis einer kostenbasierten Optimierung (*Cost Based Optimizer*) zu erstellen.

Grenzen und Probleme

Ein wesentliches Problem bei der variablen Gestaltung einer Anfrageverarbeitung ist die Explosion des Variantenraumes durch das Hinzufügen von einzelnen Sprachkonstrukten zur definierten SQL-Basis. Hierfür wird bei der Optimierung eine Erweiterung der logischen Termersetzungsregeln benötigt, die sich bei unseren Versuchen nicht in Form einer schrittweisen Verfeinerung kapseln ließen. Mit Hilfe von *Design Rule Checks* könnten sich bestimmte Kombinationen von Regeln ausschließen lassen, aber mit jeder Regel steigt die Anzahl der zusätzlichen Bedingungen, die durch *Design Rule Checks* überprüft werden müssen.

7.1.5 Diskussion der Ergebnisse

Diskussion

Im Folgenden wollen wir die wichtigsten Erkenntnisse aus der FAMEDBMS-Fallstudie im Kontext des variablen Nanadatenmanagements für eingebettete Systeme diskutieren.

Variable Referenzarchitektur: Die Domäne der Datenbanksysteme ist sowohl wissenschaftlich als auch praktisch umfangreich evaluiert. Es existieren eine Vielzahl von Varianten und Algorithmen für verschiedene Problemstellungen im Datenmanagementumfeld. Implizit sind auch einige Abhängigkeiten von unterschiedlichen Implementierungsvarianten bekannt, doch es finden sich in der wissenschaftlichen Literatur keine systematisch modellierten Abhängigkeiten. FAMEDBMS wurde als ein Vorschlag für eine Referenzarchitektur von Variationspunkten und als ein

Implementierungsvorschlag auf Basis einer wissenschaftlich-orientierten Domänenanalyse entwickelt. Hohe Variabilität, die explizit in Merkmalsdiagrammen und Kollaborationsschichten definiert ist, stand im Vordergrund der Entwicklung. Die in Abbildungen 7.2 - 7.5 gezeigten Variationspunkte der Architekturen sind das Ergebnis von mehreren Interaktionszyklen und dient für unsere weiteren Entwicklungen als Referenzarchitektur.

Granularität der Variationspunkte: Derzeitig hat FAMEDBMS[2] mehr als 180 Variationspunkte, die teilweise sehr feingranular implementiert sind. Mehr als 40 Prozent der Merkmale umfassen weniger als 100 Zeilen Quellcode. Etwa 60 Prozent der Merkmale sind optionale Merkmale und können, wenn sie nicht benötigt werden, „herauskonfiguriert" werden. Bei der Implementierung durch FEATUREC++ konnte vollständig auf Präprozessorkonfigurationen verzichtet werden. Nicht im Fokus dieser Entwicklung war die Minimierung des Ressourcenbedarfs der einzelnen Merkmale. Es war zwar das Ziel, möglichst feingranulare Merkmale für den Bereich des Nanodatenmanagements zu schaffen, doch der saubere Entwurf und die klare Trennung der Merkmale aus Software-Engineering-Sicht standen im Vordergrund. Des Weiteren wurden insbesondere durch das Einbinden von Standardbibliotheken viele Merkmale künstlich aufgebläht. Bei einem konsequenten Minimieren der Merkmale müsste auch die in den Merkmalen nicht verwendete Funktionalität weggelassen werden.

Querschneidende Belange: Die Kapselung der Erweiterung in Merkmalen entflechtet ein Großteil der querschneidenden Belange. Bis auf wenige Erweiterungen im Bereich der Transaktionsverwaltung sind die Erweiterungen durchweg heterogener Natur. In der Grundversion des vorgestellten Speichermanagers treten gar keine homogenen querschneidenden Belange auf, sodass der Speichermanager in der Basisversion ohne *Aspectual Mixin Layers* entwickelt werden konnte. Neben dem wohl überlegten Entwurf ist vor allem die sehr feingranulare Struktur und damit hochspezialisierte Umsetzung der Merkmale hierfür verantwortlich. Dies führt dazu, dass keine Erweiterungspunkte für homogene Erweiterungen im Quelltext entstehen. Da FAMEDBMS unter Laborbedingungen entworfen wurde, gab es auch keine ungeplanten Änderungen, die ein häufiger Grund für nicht-hierarchiekonforme Änderungen sind. Auf Grund der vielen optionalen Teilmerkmale der Transaktionsverwaltung wurden einige optionale Merkmalsinteraktionen mit optionalem Weben realisiert. In der Grundversion haben wir auf

2 inklusive Transaktionsverwaltung und Anfrageverarbeitung

den Einsatz von optionalen Webetechniken aus technischen Gründen verzichtet, da für einige Zielsysteme der von uns in FeatureC++ eingesetzte AspektC++ Übersetzer nicht auf allen Zielplattformen im Bereich der eingebetteten Systeme verwendbar ist.

Varianten von FameDBMS

In der von uns vorgestellten Umsetzung dient *FameDBMS* als Ausgangsbasis für weitere Entwicklungen. Neben unserer eigenen Entwicklung *RobbyDBMS*, die wir im folgenden Abschnitt vorstellen, baut auch der *Celluar DBMS*-Ansatz von Rahman [uR11] und *AutoDaMa* von Thüm et al. [TSP+11] auf dem Kern von FameDBMS auf und nutzt die explizit definierten Variationspunkte zur Umsetzung von eigenen domänenspezifischen Änderungen.

7.2 2. Fallstudie: RobbyDBMS

Einsatzgebiet von RobbyDBMS

Mobile Roboter sind in vielen Einsatzgebieten, wie beispielsweise in Krankenhäusern zum Gütertransport [RKF98] oder als Reinigungs- und Serviceroboter [PRSF00], längst aus der Forschung in die Anwendung übergegangen. Auch wenn die bisher verrichteten Aufgaben vergleichsweise einfach sind, benötigen sie für ihren Bereich komplexe Datenverarbeitung, um beispielsweise eine Umgebungserfassung, Lokalisierung, Kollisionserkennung und Koordination durchzuführen. Diese Aufgaben ergeben sich aus der Tatsache, dass teilweise Umgebungen existieren, die schwer oder gar nicht zugänglich sind, wie beispielsweise im Bergbau, unter Wasser oder im Kanalbau. Die folgende Fallstudie zeigt die Entwicklung von RobbyDBMS, das Datenmanagementfunktionalitäten für ein autonomes Robotersystem bereitstellt. Die Domänenanalyse von FameDBMS ist sehr weit gefasst. Eine konkrete Fokussierung auf ein bestimmtes Anwendungsgebiet war im Entwurf nicht vorgesehen. In der RobbyDBMS-Fallstudie steht nun eine konkrete Plattform im Mittelpunkt. Die starken Hardware-Restriktionen der mobilen Plattform sollen Aufschluss darüber geben, ob FeatureC++ in diesem Bereich noch einsetzbar ist.

7.2.1 Zielsetzung der Fallstudie

Ziele

Im Gegensatz zur FameDBMS-Fallstudie, die eine weitgefasste Domänenanalyse ohne konkrete Anpassung auf den konkreten Anwendungsfall beinhaltet, wird die Produktlinie RobbyDBMS stärker eingeschränkt. Die Ziele der Fallstudie können wie folgt aufgeteilt werden:

Domänenanalyse „Datenmanagement für Robotersysteme“: Im ersten Schritt soll für das konkrete Szenario eine Domänenanalyse durchgeführt werden. Dabei wird bezüglich einer gegebenen Hardwareproduktlinie für den Einsatzbereich der mobilen autonomen Robotersysteme variable Datenmanagementfunktionalität bereitgestellt. Die Basis der Entwicklung soll die FAMEDBMS-Produktlinie bilden.

Programmgröße. Die im Rahmen der Fallstudie verwendete Hardware mit wenigen Kilobyte Programmspeicher stellt eine sehr starke Restriktion für die zu entwickelnden Programme dar. Ziel der Fallstudie ist es herauszufinden, wie für die ausgewählte Hardware minimale Datenmanagementfunktionalität im Kontext von mobilen autonomen Robotersystemen bereitgestellt werden kann. Die Fallstudie soll die untere Schranke für den Einsatzbereich unserer Variabilitätsmechanismen, die wir mit FEATUREC++ bereitstellen, ermitteln und Aussagen zur Programmgröße (*Footprint*) für diese Szenarien im Bereich des variablen Nanodatenmanagements treffen.

Skalierbarkeit von FEATUREC++: Die Erweiterungen der Merkmalsorientierten Programmierung finden im Wesentlichen auf der Ebene der Methodenverfeinerungen statt. Während sich FEATUREC++ für die in FAMEDBMS eingesetzte Granularitätsstufe der Merkmale ideal eignet, gilt es herauszufinden, inwieweit FEATUREC++ bei feingranularen Erweiterungen ebenfalls noch einsetzbar ist und die Vorteile des merkmalsorientierten Ansatzes mit FEATUREC++ in diesem Bereich noch skaliert.

Im folgenden Abschnitt wird ein Szenario vorgestellt, das als Grundlage für unsere Untersuchungen dienen soll.

7.2.2 Szenario der Fallstudie

Szenario

Für das Szenario wird ein autonomer mobiler Roboter mit verschiedenen Sensoren zur Umgebungs- und Kollisionserkennung ausgestattet. Hierzu sollen beispielsweise Odometriesensoren zur Bestimmung zurückgelegter Entfernungen verwendet werden. Mit ihrer Hilfe lässt sich die Position des Roboters im Raum ungefähr ermitteln. Des Weiteren sollen durch den Einsatz von weiteren Sensoren beispielsweise Temperatur und Lichtverhältnisse bestimmt werden, um weitere Umgebungscharakteristika zu erfassen. Die Daten für unser Szenario können wie folgt zusammengefasst werden:

- Sensordaten zur Umgebungserkennung und -erkundung,
- Daten zur Kalibrierung von Sensoren und Aktoren,

Abbildung 7.6: Robonova

Abbildung 7.7: Crash-Bobby

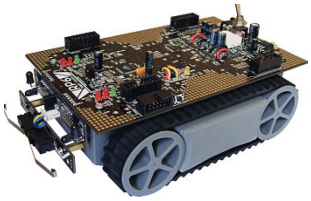

Abbildung 7.8: RP6

- Konfigurations- bzw. Metadaten,
- Störungsprotokolle und
- Wartungs- und Servicedaten.

AVR Hardware

Diese Daten sollen, in Abhängigkeit vom Vorhandensein unterschiedlicher Sensoren, verschiedene Messwerte in Datenreihen erfassen, verarbeiten und gegebenenfalls auch speichern. Die Grundlage hierfür bieten eingebettete Systeme, basierend auf Mikroprozessoren der Modellreihe *AVR* von *Atmel1*. Die Hardware ist ebenfalls in Form einer Produktlinie realisiert. Die Abbildungen 7.6 bis 7.8 zeigen Anwendungsmöglichkeiten autonomer mobiler Roboter (Robonova[3], Crash-Bobby[4] und RP6[5]) auf Basis der im Szenario eingesetzten Hardware-Mikroprozessorproduktlinie.

Technische Daten

Eine Besonderheit der eingesetzten Hardware ist die geringe Leistungsfähigkeit der Mikroprozessoren und die geringe Ausstattung der Programm- und Arbeitsspeicher der Systeme. Im Einsatz für das Szenario sind Systeme mit niedrig getakteten 8-Bit Prozessoren mit 8 beziehungsweise 16 MHz und ein Programmspeicher von 32 oder 128 KB sowie Arbeitsspeicher von 2 bis 4 KB je nach Konfiguration. Für die persistente Speicherung besitzen die Robotersysteme EEPROM-Speicher, die mindestens 1 KB groß sind. Durch die Einschränkungen im Funktionsumfang und die begrenzte Hardware-Ausstattung

3 Bildquelle: http://www.lvbots.com/gallery/bots.php
4 Bildquelle: http://www.qfix-robotics.de
5 Bildquelle: http://www.arexx.com

mit geringer Leistungsfähigkeit können Kostenvorteile erzielt und Energieeinsparungen realisiert werden.

7.2.3 Umsetzung von RobbyDBMS

Domänenanalyse

Zur Umsetzung des Szenarios wurde als Ausgangslage die variable Struktur und die Softwarearchitektur von FAMEDBMS verwendet. Neben dem „Minimieren“ der Merkmale von FAMEDBMS sollten noch mehr Komponenten als bei FAMEDBMS optional gestaltet werden. Aus den Umgebungsparametern ergeben sich die in Abbildung 7.9 beschriebenen Merkmale, deren Funktionsumfang im Folgenden kurz erläutert wird:

StorageManager: Der Speichermanager organisiert und überwacht das Zusammenspiel von permanenter Speicherung und Pufferung (`BufferManager`) der Daten auf unterschiedlichen Hardwaresystemen (`StorageDevice`). Im Folgenden werden die zwei Merkmale genauer vorgestellt:

BufferManager: Die klassische Aufgabe der Pufferverwaltung ist die Optimierung der Lese- und Schreibzugriffe auf den Datenbestand, der durch die Zugriffslücke zwischen dem stabilen und flüchtigen Speicher entsteht. Des Weiteren wird häufig im Bereich der tief-eingebetteten Systeme der permanente Datenspeicher auf Grund der eingesetzten Technologie (Flash-Speicher) hinsichtlich der Schreibzyklen begrenzt. Um deren Anzahl zu reduzieren beziehungsweise die Schreiboperationen zu optimieren wird die Pufferverwaltung eingesetzt. Weiterhin kann die Pufferverwaltung den benötigten Speicher zur Übersetzungszeit statisch oder zur Laufzeit dynamisch allokieren. Die zwei Vorteile der statischen Reservierung sind zum einen die einfachere Handhabung, die eine fehleranfällige, platzaufwendige und performanzbremsende Verwaltung des benötigten Speichers übernimmt. Zum anderen sind für bestimmte sicherheitskritische Systeme dynamische Allokationen des Speichers nicht erlaubt. Im Gegensatz dazu ermöglicht die dynamische Allokation zur Datenspeicherung mehr Flexibilität und die Größe der Tupel ist nicht zur Übersetzung festzulegen.

StorageDevice: Das Merkmal `StorageDevice` kapselt die hardware-spezifischen Erweiterungen für das *Robby* und *Bobby-Board*.

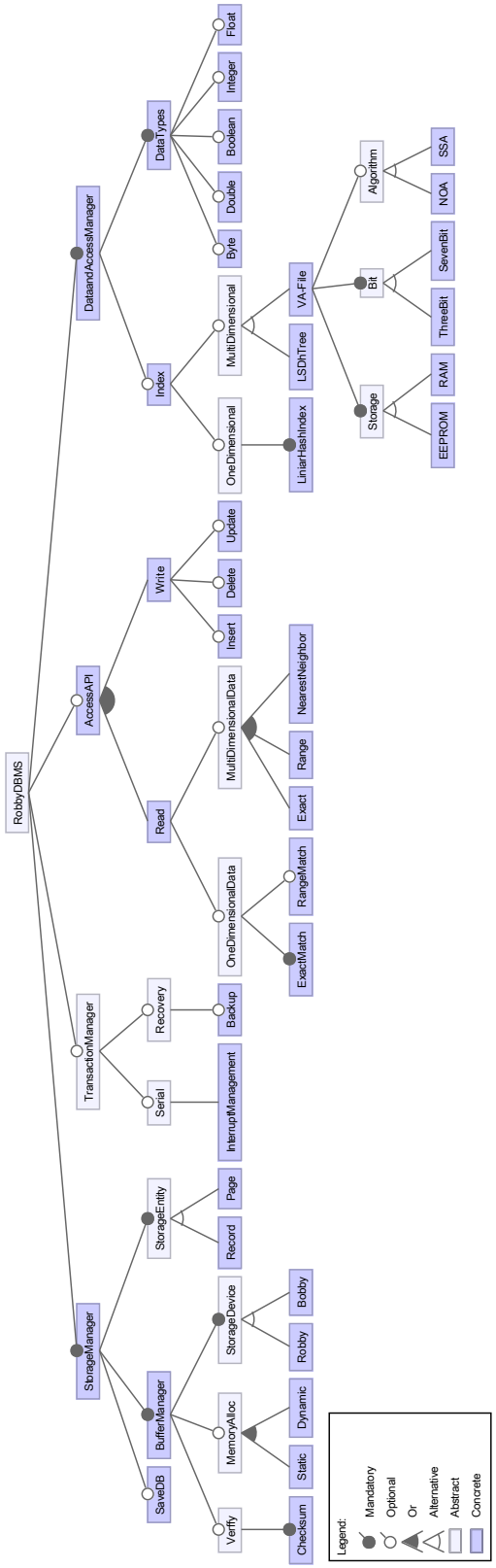

Abbildung 7.9: Variationspunkte von ROBBYDBMS

Transaction: Das Merkmal `Transaction` erweitert die Pufferverwaltung um eine rudimentäre Transaktionsverwaltung. Dabei werden die von der Pufferverwaltung zwischengespeicherten Daten durch eine transaktionale Kapselung vor Veränderungen von anderen Operationen geschützt. Die Transaktionsverwaltung kann sowohl für Lese- als auch Schreibzugriffe eingesetzt werden.

Persist: Die Änderungen von Datensätzen werden bei ROBBYDBMS zunächst im Arbeitsspeicher vorgenommen. Hardwarefehler oder ein Zusammenbrechen der Stromversorgung führen zwangsläufig zum Verlust der Daten. Das Merkmal *Persist* ermöglicht das periodische Wegschreiben von Daten in den permanenten Datenspeicher.

Serial: ROBBYDBMS erzwingt bei der Abarbeitung der Transaktionen eine serielle Ausführungsreihenfolge. Konkurrierende Transaktionen werden bis zum Ende der laufenden Transaktion aufgeschoben. Die Umsetzung der Transaktionsverwaltung ist durch das Abschalten der Interrupts realisiert.

Checksum: Das optionale Merkmal `Checksum` erkennt fehlerhafte Datensätze. Dies kann zum einen durch abgebrochene Speichervorgänge infolge eines Stromausfalles geschehen, und zum Anderen kann der permanente Datenspeicher nach längerem Gebrauch seine Speicherfähigkeit verlieren. In beiden Fällen müssen fehlerhafte Daten durch eine Prüfung mit einer zum Datenelement gespeicherten Prüfsumme identifiziert werden, um inkonsistente Zustände des Gesamtsystems zu vermeiden.

Index: Das Merkmal *Index* ermöglicht das effiziente Auffinden von Daten im Speicher und optimiert somit die Suche gegenüber der rein sequentiellen Suche im Datenbestand. Am häufigsten werden hierzu Baumstrukturen oder Hash-Verfahren eingesetzt. Für ROBBYDBMS sind statisches Hashing, LSDh-Bäume und das VA-File als Indexstrukturen [SSH11] implementiert.

Ergebnisse der Umsetzung

Die vollständige Umsetzung implementiert mehr als 50 Merkmale. Einige dieser Merkmale ergeben sich aus der Merkmalsinteraktion von optionalen Merkmalen (*Lifter*), die in dieser Variante nicht mit optionalem Weben umgesetzt wurden. Abbildung 7.10 zeigt einen Ausschnitt des Kollaborationendiagramms. Die Implementierung umfasst insgesamt 23 Klassen. Die zentrale Klasse von ROBBYDBMS ist der `StorageManager`, der von einem Großteil aller Merkmale erweitert wird. Alle Erweiterungsimplementierungen von ROBBYDBMS sind heterogen. *Aspectual Mixin Layers* kamen nicht zum Einsatz.

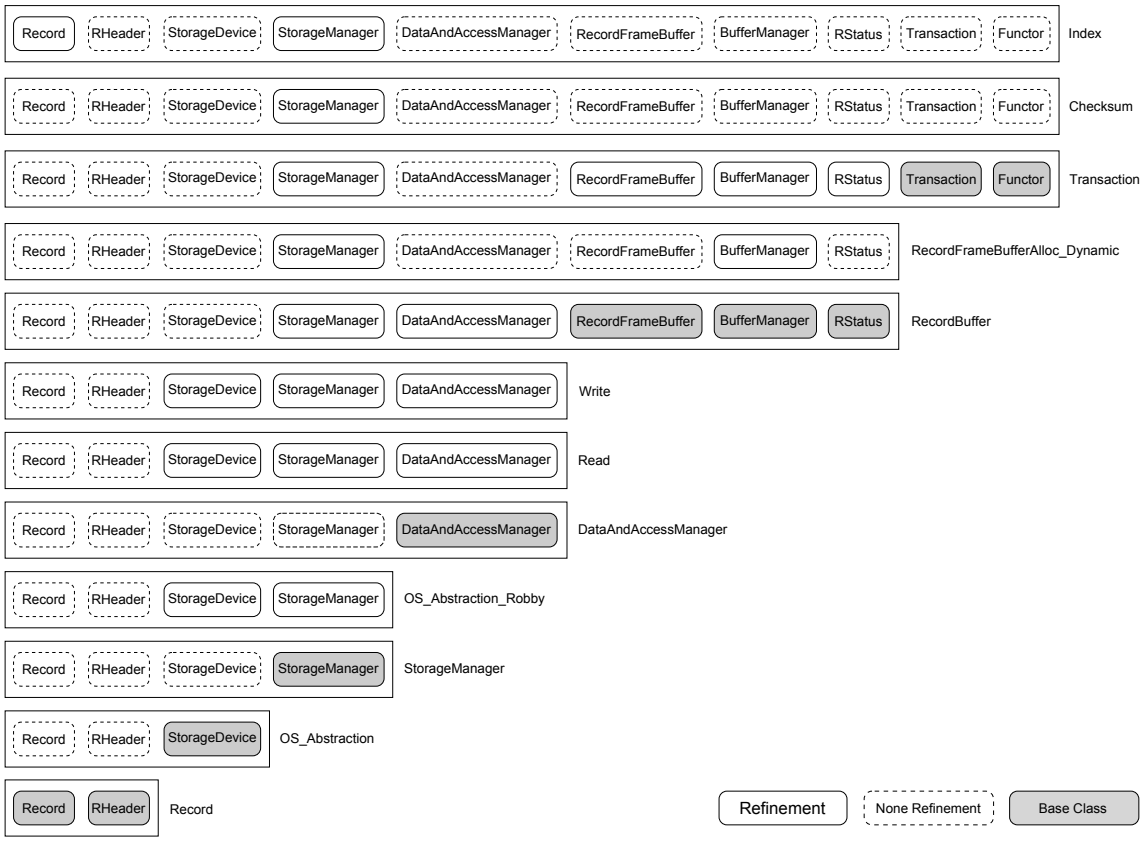

Abbildung 7.10: Ausschnitt des Kollaborationendiagramms ROBBYDBMS [Lie08](verändert)

7.2.4 Evaluierung von RobbyDBMS

Im folgendem Abschnitt evaluieren wir RobbyDBMS bezüglich der eingangs definierten Ziele der Fallstudie. Hierzu werden wir Bezug auf die Domänenanalyse, die Programmgröße und die Skalierbarkeit des Konfigurationsansatzes mittels FeatureC++ nehmen.

Domänenanalyse „Datenmanagement für Robotersysteme“

Bewertung der Domänenanalyse

Verglichen mit FameDBMS handelt es sich bei RobbyDBMS um eine Extremvariante, die den sehr starken Hardwarerestriktionen angepasst wurde. Die Merkmale sind wesentlich feingranularer strukturiert. Einige Merkmale erweitern den Funktionsumfang mit wenigen 100 Byte. Generell werden für verschiedene Aufgaben unterschiedliche Anforderungen an dieDatenmanagementfunktionalität in diesem Bereich gestellt. Dies reicht von einer einfachen Lese- und Zugriffsunterstützung für Daten zur Kalibrierung von Sensoren und Aktoren bis zur Unterstützung von Wiederherstellungsprozessen im Fehlerfall, mehrdimensionalen Indexstrukturen, die nächste Nachbarschaftsanfragen zur Umgebungserkennung und -erkundung unterstützen und einer Transaktionsverwaltung, die Atomarität, Isolation und Konsistenz bietet. In unserer Domänenanalyse identifizierten wir insgesamt 42 Variationspunkte (vgl. hierzu Abbildung 7.9). Auf dieser Grundlage können Anwendungen des Roboters verschiedene Messdaten ihrer Sensoren speichern und verarbeiten oder eine Umgebungs- und Kollisionserkennung durchführen.

Programmgröße

Bewertung der Programmgröße

Das wesentliche Kriterium für die Evaluierung von RobbyDBMS ist die Programmgröße und der initial belegte Arbeitsspeicher. Für unsere Evaluierung haben wir den `avr-gcc` Cross-Compiler in der Version 4.0.2 eingesetzt. Abbildung 7.11 zeigt die Verarbeitungsreihenfolge. Der `avr-gcc` Cross-Compiler besitzt mehr als 100 Optimierungsparameter, die den Übersetzungsprozess steuern. Für die Fallstudie wurde mit der Optimierungsstufe `-Os` übersetzt. Entgegen der meisten anderen Optimierungsparameter, die einen Kompromiss zwischen Programmlaufzeit und -größe darstellen, weist die Option `-Os` den Compiler an, die Programmgröße zu reduzieren. Die Tabelle 7.1 präsentiert verschiedene Konfigurationen von RobbyDBMS. Die Übersicht zeigt, ausgehend von einem minimalen System, verschiedene DBMS-Varianten und ihre unterschiedlichen Programmgrößen und den initial belegten Arbeitsspeicher[6]. Die minimale

6 Die Bestimmung des tatsächlich verbrauchten Arbeitsspeichers zur Laufzeit setzt eine genaue Analyse der Varianten voraus, die nicht Gegenstand dieser Fallstudie ist.

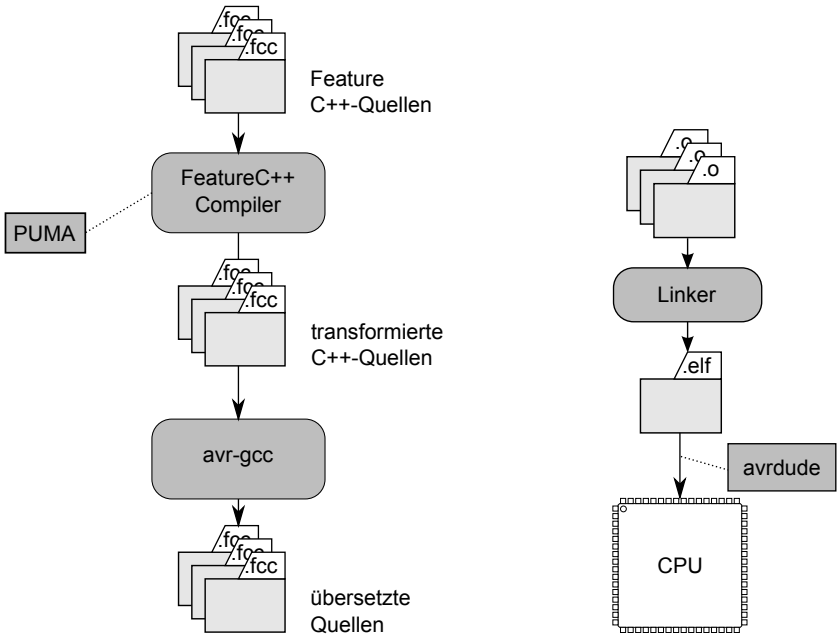

Abbildung 7.11: Verarbeitungsreihenfolge bei der Übersetzung ROBBYDBMS

Variante (Konfiguration 1) ist eine, nur auf Leseoperationen ausgerichtete „Datenbank“ mit 658 Byte. Dagegen belegt die Variante sechs, die Lese- und Schreiboperationen anbietet und dynamische Speicherallokationen verwendet, 2,2 KB Programmspeicher und initial 10 Byte Arbeitsspeicher. Die Konfiguration (Variante 7) mit einem Index auf Basis des statischen Hashens, einer Transaktionsverwaltung und einer dynamischen Speicherallokation benötigt schon 3,3 KB Programmspeicher. Werden Konfigurationen (Variante 9 und 10) mit mehrdimensionalen Indexstrukturen auf Basis vom VA-File oder dem LSDh-Baum benötigt, so steigt der Programmspeicher auf 12,4 und 14,7 KB. Dabei unterstützen die mehrdimensionalen Indexstrukturen sowohl Bereichsanfragen als auch Anfragen, die die nächste Nachbarschaftssuche benötigen. Die größte Version mit allen möglichen optionalen Merkmalen ist 18,6 KB groß.

Skalierbarkeit von FEATUREC++

Feingranulare Erweiterung

Die durch das Szenario vorgegebenen starken Hardware-Einschränkungen sorgten für sehr feingranulare Erweiterungen. In unserer Umsetzung von ROBBYDBMS haben wir Folgendes festgestellt: Je feingranularer die Erweiterungen waren, desto häufiger mussten die Erweiterungen auf Anweisungsebene innerhalb einer Methode durchgeführt werden. Dies ist zum Beispiel notwendig, um auf lokale Variablen zuzugreifen oder um bestimmte Abschnitte einer Methode zu erweitern (beispielsweise durch feingranulares

Variantennummer	Leseoperationen	Schreiboperationen	Dyamische Speicherallokation	Transaktionsverwaltung	Integritätsüberprüfung	Hash-Index	VA-File-Index	LSDh-Baum	Programmcode in Byte	Arbeitsspeicher in Byte
1	+	-	-	-	-	-	-	-	658	0
2	+	+	-	-	-	-	-	-	996	0
3	+	+	-	-	+	-	-	-	1.278	0
4	+	+	-	-	-	+	-	-	1.686	0
5	+	+	-	-	+	+	-	-	2.002	0
6	+	+	+	-	-	-	-	-	2.294	10
7	+	+	+	-	+	+	-	-	3.330	10
8	+	+	+	+	+	+	-	-	3.734	12
9	+	+	+	-	-	-	+	-	12.452	41
10	+	+	+	-	-	-	-	+	14.774	50

Tabelle 7.1: Übersicht der Programmgröße von verschiedenen ROBBYDBMS Varianten in Byte

Sperren). Sowohl die reine Merkmalskomposition als auch unsere aspektorientierten Erweiterungen von FEATUREC++ bieten hierfür keine adäquaten Konzepte. Eine mögliche Lösung für den Zugriff auf lokale Variablen ist die Änderung der lokalen Variable in eine Instanzvariable. Während lokale Variablen bevorzugt in Registern des Prozessors gehalten und dadurch Lade- sowie Speicheroperationen vermieden werden, führt eine alternative Umsetzung mit Instanzvariablen häufig zu einer Vergrößerung des Programms und gegebenenfalls zu einer geringeren Performanz.

Hook-Methoden

Eine andere Möglichkeit bietet der Einsatz von *Hook*-Methoden, die explizit den Erweiterungspunkt in den Methoden der zu erweiternden Klasse markieren. Diese leere Methode kann später durch eine Verfeinerung oder einen Aspekt erweitert werden. Wird die *Hook*-Methode nicht durch eine Konfiguration erweitert, so erfolgt keine Berücksichtigung dieser leeren Methode durch den Compiler und erzeugt somit einen Overhead. Für den Zugriff auf die lokalen Variablen werden die *Hook*-Methoden um die

lokalen Variablen als Parameter erweitert. Wird die *Hook*-Methode dann durch einen Aspekt oder eine Verfeinerung erweitert, können diese Variablen beliebig manipuliert werden. Abbildung 7.12 zeigt den Zugriff einer Verfeinerung (Zeile 25) auf die lokale Variable `ADDRESS_u16t&` (Zeile 11) der Basisklasse des `StorageManagers` und die Manipulation der Variable durch die Erweiterung in Zeile 26.

```
1  //Base StorageManager define Hook-Methode
2  class StorageManager {
3     private:
4       //Empty Hook-Methode
5       void hook_modifyCurAddress(ADDRESS_u16t& cadd)
6       {
7         return;
8       };
9       ADDRESS_u16t search(RID_u16t key)
10      {
11        ADDRESS_u16t cur = 0;
12        while (cur < (sd.MAXSTORAGE-1))
13        { ...
14          // extension point; access to local data
15           hook_modifyCurAddress(cur);
16        }
17      }
18    ...
19 };
20 //Refinement StorageManager using Hook-Methode
21 refines class StorageManager {
22    private :
23       void hook_modifyCurAddress(ADDRESS_u16t& cadd)
24       {
25          super::hook_modifyCurAddress(cadd);
26          cadd++;
27       }
28  };
```

Abbildung 7.12: Definition und Verfeinerung einer *Hook*-Methode

Ähnlich findet auch die feingranulare Erweiterung von Quelltext innerhalb einer Methode statt. Sollen innerhalb einer Schleife bestimmte Quelltextblöcke durch eine Verfeinerung gesperrt werden, so werden an den Stellen, an denen gesperrt beziehungsweise entsperrt werden soll, *Hook*-Methoden angelegt. Diese können dann durch Sperrfunktionalität erweitert werden. Durch die Verwendung von *Hook*-Methoden konnten alle feingranularen Erweiterungen umgesetzt werden, sodass keine Verwendung von Präprozessoranweisungen erfolgen musste.

Nachteil der Hook-Methoden

Konzeptbedingt haben die *Hook*-Methoden einen Nachteil. Abweichend vom sonstigen Konzept der Merkmalsorientierten Programmierung muss bereits in der zu verfeinernden Klasse dieser Erweiterungspunkt explizit berücksichtigt werden. Dies stört den Lesefluss und widerspricht der Unabhängigkeit der Merkmale. Ein sparsamer Umgang mit dieser Art der Erweiterungen ist deshalb anzuraten.

7.2.5 Diskussion der Ergebnisse

Ergebnisdiskussion

Während FAMEDBMS für eine recht breit angelegt Domäne konzipiert ist, die auf einer stark theoretischen Analyse des Anwendungsgebietes der eingebetteten Systeme basiert, liegt der Fokus von ROBBYDBMS auf Szenarien von mobilen autonomen Robotern mit einer stark eingeschränkten Hardware-Produktlinie. Die Basis-Architektur liefert FAMEDBMS. Die Ergebnisse von ROBBYDBMS hinsichtlich der in Abschnitt 7.2.1 definierten Zielsetzungen können wie folgt zusammengefasst werden.

Domänenanalyse „Datenmanagement für Robotersysteme“: Die Hardware-Mikroprozessorproduktlinie AVR von Atmel stellt eine Ausgangsbasis für verschiedene Varianten von autonomen mobilen Robotern dar. Das aufgabenbezogene autonome Agieren von solchen Robotersystemen verlangt die Verarbeitung von größeren Datenmengen[7]. Auf Basis der FAMEDBMS-Produktlinie stellt ROBBYDBMS eine Produktlinie von 42 Merkmalen mit Datenmanagementfunktionalität dar. Hierbei werden besonders Implementierungen für eine effiziente und abstrahierte Speicherung und Weiterverarbeitung von Daten, Zugriffsstrukturen wie beispielsweise multidimensionale Indexstrukturen, eine einfache Transaktionsverwaltung, die hardware-äquivalent eine einsprechende Interrupt-Behandlung durchführt und damit Atomarität, Isolation und Konsistenz sichert, Recoveryfunktionalität oder eine einheitliche Zugriffs-API bereitgestellt. Eine Besonderheit von ROBBYDBMS ist dabei die sehr feingranulare Domänenanalyse.

Programmgröße: Die durch das Szenario vorgegebenen Hardwarerestriktionen zwingen zu einem konsequenten Umgang mit den Ressourcen. Für einen Programmspeicher von 32 bis 128 KB ist die im ROBBYDBMS-Projekt vorgestellte Lösung anwendbar. Die Größen der verschiedenen Varianten lassen noch genügend Freiraum, um die Anwendungssoftware für die Roboter zu implementieren. Ausgehend von einer abstrahierten Lese- und Schreibunterstützung für die verwendete Hardwareproduktlinie können über eine API, die weniger als 1 Kilobyte Programmspeicher

7 bezogen auf die Rechenleistung und den Speicherplatz

benötigt, unter anderem verschiedene Zugriffsstrukturen, ein dynamisches Speichermanagement, *Recovery*-Funktionalität oder eine einfache Transaktionsverwaltung hinzu konfiguriert werden. Eine vollumfängliche Konfiguration ist mit 18,6 KB immer noch vergleichsweise klein.

Skalierbarkeit von FEATUREC++: Auf Grund der sehr starken Hardwarerestriktionen ergaben sich sehr feingranulare Umsetzungen von Merkmalen mit Merkmalsgrößen von teilweise wenigen 100 Byte. Im Hinblick auf diese Ressourcenoptimierung war es hierfür zum Teil notwendig, auf Anweisungsebene innerhalb von Methoden Erweiterungen von Merkmalen einzufügen. Dies ist weder im merkmalsorientierten Kern noch in den aspektorientierten Erweiterungen von FEATUREC++ vorgesehen. Durch die Einführung von *Hook*-Methoden, die explizite Variationspunkte in die zu verfeinernden Quellcodefragmente setzen, können die feingranularen Umsetzungen ressourcenschonend realisiert werden. Der Nachteil der Invasivität des Ansatzes bezüglich der zu verfeinernden Quellcodefragmente wird durch die Performanzvorteile aus unserer Sicht aufgehoben. Die Fallstudie ROBBYDBMS zeigt, dass durch Merkmalsorientierte Programmierung mittels FEATUREC++ hochskalierbare Datenmanagementlösungen im Bereich der eingebetteten Systeme anwendbar sind. Trotz der teilweise sehr feingranularen Erweiterungen konnten wir vollständig auf Präprozessoranweisungen für die Konfigurationen von Merkmalsvariationen verzichten.

Fazit

Mit einigen Einschränkungen bezüglich der Variationspunkte auf Statementebene, konnten wir mit dieser Fallstudie zeigen, dass sich mit FEATUREC++ sehr feingranulare Merkmale des Nanodatenmanagements für tief eingebettete Systeme umsetzen lassen. Für die Basis der Umsetzung wurde die Implementierung von FAMEDBMS genutzt, die somit implizit ihre Wiederverwendbarkeit unter Beweis stellt. Im Folgenden wollen wir in einer weiteren Fallstudie eine bessere Vergleichbarkeit erreichen.

7.3 3. Fallstudie: Berkeley DB

Eine Schwierigkeit der vorangegangenen Fallstudie des AHEAD-basierten Speichermanagers, von FAMEDBMS und ROBBYDBMS ist die Vergleichbarkeit bezüglich der Performanz der Systeme. Auf Grund des sehr unterschiedlichen Funktionsumfangs von kommerziellen Lösungen beziehungsweise der eingeschränkten Verfügbarkeit von Forschungsprototypen fällt ein Vergleich bezüglich der Performanz schwer. Um eine möglichst objektive Aussage bezüglich der Performanz zu ermöglichen, wird auf Basis

von Berkeley DB eine mit FEATUREC++ realisierte Variante entworfen, welche den gleichen Funktionsumfang besitzt, aber noch stärkere Variabilität umfasst. Berkeley DB gilt als eines der bekanntesten und erfolgreichste DBMS im Bereich der eingebetteten Systeme und des eingebetteten Datenmanagements. Die Grundidee war die Entwicklung einer konfigurierbaren Bibliothek von Datenmanagementfunktionalität, die von einer Anwendung genutzt werden kann.

7.3.1 Zielsetzung der Fallstudie

Ziele

Das Ziel dieser Fallstudie ist es, eine vergleichbare Analyse zwischen der etablierten C-Version von Berkeley DB und einer refaktorisierten Variante von Berkeley DB auf Basis von FEATUREC++ durchzuführen. Hierbei sollen die folgenden vier Punkte untersucht werden:

Variabilität und Anpassbarkeit: Die von Berkeley DB ausgelieferte C-Version ist in Grenzen mit Hilfe von Präprozessoranweisungen variabel und an den Anwendungskontext anpassbar. Zunächst sollen die bisherigen Variationspunkte durch Kompositionsmechanismen von FEATUREC++ ersetzt werden. Zudem werden dann weitere Variationspunkte ermittelt und implementiert. Diese Variationspunkte sollen dazu beitragen, dass der bisherige fixe Funktionsumfang anschließend variabel ist.

Performanz: Moderne Programmierkonzepte werden im Bereich der eingebetteten Systeme häufig mit dem Hinweis auf Performanznachteile abgelehnt, die durch das höhere Abstraktionslevel erzeugt werden. Ziel der von uns durchgeführten Refaktorisierung von Berkeley DB soll eine im Funktionsumfang identische Version von Berkeley DB sein, die die Variationspunkte mit FEATUREC++ realisiert. Auf dieser Basis soll ein vergleichender Performanztest durchgeführt werden. Für diesen wird der von Berkeley DB selbst entworfene Benchmark verwendet, der die Leistungsfähigkeit des Systems demonstriert. Bei gleichem Funktionsumfang soll es dadurch möglich sein, den *Overhead* der FEATUREC++ zu quantifizieren. Des Weiteren gilt es zu untersuchen, ob durch den zusätzlich optional gestalteten Funktionsumfang die Performanz des Systems sogar gesteigert werden kann.

Programmgröße: Speziell im Einsatzbereich der eingebetteten Systeme spielt die Größe des ausführbaren Codes eine entscheidende Rolle. Nach der Refaktorisierung von Berkeley DB gilt es zu überprüfen, ob die Verwendung von FEATUREC++ Version zu einem *Overhead* beim ausführbaren Code führt. Des Weiteren soll überprüft

werden, ob durch die weitere Herauslösung von optionaler Funktionalität eine Verschlankung des Systems möglich ist.

Verständlichkeit des Codes: Ein Problem der Originalversion von Berkeley DB ist der massive Einsatz von Präprozessoranweisungen, der zu einer Verschlechterung der Lesbarkeit und der Verständlichkeit des Quelltextes führt. Bezüglich der Fallstudie soll untersucht werden, ob sich die Lesbarkeit des Quelltextes erhöht. Da eine direkte Messung nur mit sehr hohem experimentellen Aufwand möglich ist, sollen hier nur qualitative Aussagen auf Basis von typischen Indikatoren, wie etwa Quellcodereplikationen oder die Vermischung von Konfigurationsanweisungen (beispielsweise Präprozessoranweisungen) mit dem Quelltext, verwendet werden.

Zur Vermeidung von Einschränkungen bezüglich der Vergleichbarkeit wird Berkeley DB bei unserer merkmalsorientierten Refaktorisierung nicht im Funktionsumfang verändert.

7.3.2 Refaktorisierung von Berkeley DB

Allgemeines Vorgehen

Die Basis von FEATUREC++ bildet C++. Aus diesem Grund ist es notwendig, eine Refaktorisierung des C Quelltextes vorzunehmen und diesen zunächst in objektorientierten Quellcode zu überführen. Im zweiten Schritt wurde der objektorientierte Quelltext in merkmalsorientierten Quelltext auf Basis von FEATUREC++ überführt. Auf Grund der Größe von Berkeley DB wurde der Prozess der Überführung automatisiert. Hierbei war von Vorteil, dass Berkeley DB schon auf einem objektbasierten Ansatz in C realisiert wurde. Ein grundlegend neues objektorientiertes Design konnte auf Basis der Mächtigkeit der C Version nicht umgesetzt werden. Auf die Verwendung virtueller Methoden wurde komplett verzichtet, um deren Auswirkung auf die Ausführungsgeschwindigkeit zu verringern. Im Folgenden werden die vier Schritte des automatisierten Prozesses beschrieben:

1. *Konvertierung von C Dateien in Klassen:* Im ersten Schritt wird für jede C Datei eine Klasse definiert. Die in C Syntax vorliegenden Funktionen wurden in C++ Syntax transformiert und als statische Methoden den Klassen hinzugefügt. Alle Aufrufe der bisherigen C Funktionen wurden entsprechend umgewandelt. Der resultierende C++ Quelltext enthielt 214 Klassen.

2. *Konvertierung von* `structs`*:* Im zweiten Schritt werden `structs` in Klassenvariablen gewandelt. Auf Grund des objektbasierten Designs der Berkeley DB C Version können die, aus den `structs` gewandelten Datenfelder den Klassen zugeordnet

werden, die diese Datenfelder mit den entsprechenden Methoden aus dem ersten Schritt bearbeiten.

3. *Konvertierung von Funktionen, die auf* `structs` *arbeiten:* Im dritten Schritt werden die Operationen, die auf den Klassenvariablen der zuvor gewandelten `structs` arbeiten, in normale Methoden gewandelt. Zeiger auf diese `structs`, die als Argumente dieser Funktionen bisher mit übergeben wurden, können entfernt werden, da sie implizit durch den `this`-Zeiger angeboten werden.

4. *Entfernen überflüssiger Funktionszeiger:* Im letzten Schritt werden die überflüssigen Funktionszeiger, die in den einzelnen `structs` die Objektbasierte Programmierung emulierten, entfernt. Dadurch kann auch der Quelltext entfernt werden, der diese Funktionszeiger bisher initialisiert hat.

Notwendige Einschränkungen

Im ersten Schritt wurden die vorhandenen Merkmale und deren statische Konfiguration mit Hilfe von Präprozessoranweisungen beibehalten. Die nun vorhandene objektorientierte C++ Basis verbesserte in Ansätzen die Wiederverwendung und Verständlichkeit des Programmcodes. Wir verzichteten bei dieser Art der Refaktorisierung bewusst auf ein komplettes Redesign der objektorientierten Struktur, das sicherlich eine bessere Wiederverwendung zu Ungunsten der Vergleichbarkeit ermöglicht hätte. Ein ausführliches Reengineering hätte die Gefahr von Fehlern unweigerlich erhöht. Dennoch haben wir sieben zentrale Klassen wie beispielsweise die Klassen des B-Baums überarbeitet. Durch die Einführung von Vererbung konnten wir zudem von Wiederverwendung Gebrauch machen. Auf die Verwendung von virtuellen Methoden wurde aus Performanzgründen komplett verzichtet.

Überführung in merkmalsorientierten Quelltext

In einem zweiten Schritt wurde der C++ Quelltext in eine merkmalsorientierte Repräsentation auf Basis von FEATUREC++ überführt, um eine umfassende Konfigurierbarkeit der Software zu erreichen. Auch hierbei wurde auf eine semiautomatische Transformation zurückgegriffen. Grundlage für die merkmalsorientierte Zerlegung ist der folgende zweistufige Prozess:

Initiale Merkmalsextraktion: Die Basis für die initiale Merkmalsextraktion bietet im ersten Schritt die bereits vorhandene Struktur von Ordnern, in denen die C Dateien von Berkeley DB organisiert sind. So sind beispielsweise alle Dateien, die für das Merkmal *Hash* benötigt werden, in einem Ordner organisiert. Im Sinne der Merkmalsorientierten Programmierung stellt dies aber nur den initialen Zustand dar, da nicht alle Fragmente des Quellcodes in diesem Ordner organisiert sind.

Im zweiten Schritt wurde eine manuell gesteuerte Zuordnung von Klassen zu Merkmalen vorgenommen. Dabei lag der Fokus auf der Isolation von optionalen Merkmalen, um die bisherige Basis von Berkeley DB weiter zu verschlanken. Hierzu wurde die isolierte Funktionalität zunächst einem Merkmal und einer korrespondierenden Klasse des Merkmals zugeordnet. Des Weiteren wurden die, sofern nicht benötigten, *Includes* entfernt und die Implementierung mit der Teilfunktionalität einzelner Fragmente des Merkmals zusammengefügt. Dieser Schritt enthält bisher noch keine Verfeinerungen.

Merkmalsorientierte Dekomposition der Klassen: Die bisherige merkmalsorientierte Zerlegung von Berkeley DB ist in der Granularität auf Klassenebene beschränkt. Mit der merkmalsorientierten Dekomposition der Klassen wird die Funktionalität von einem Merkmal, das auf mehrere andere Merkmale verteilt ist, kohärent in einem Merkmal gekapselt. So sind beispielsweise Quellcodefragmente des Merkmals Transaktionsverwaltung auf ca. 30 Klassen und 11 Merkmale verteilt. Auf Grund der fehlenden Verfügbarkeit eines geeigneten Werkzeuges wurde dieser Prozess für ausgewählte Merkmale manuell durchgeführt. Aufbauend auf dem ersten Schritt wurde nur ein Teil (ca. 13.000 Lines of Code (LOC)) von Berkeley DB in merkmalsorientierten FEATUREC++ Quelltext transformiert. Der Fokus lag dabei auf den Klassen des B-Baums und auf Merkmalen, die im Hinblick auf Performanz interessant erschienen. Diesbezüglich wurden insbesondere die Merkmale Replikation und Transaktionsverwaltung betrachtet. Dabei wurde der Quellcode in Basisimplementierungen und Verfeinerungen aufgespalten. Dadurch konnten die entsprechenden Präprozessoranweisungen entfernt werden. Einige Erweiterungen wurden mit *Hook*-Methoden realisiert. Dies war insbesondere dann der Fall, wenn feingranular in den Kontrollfluss eingegriffen werden musste. Funktionszeiger und deren bedingte Initialisierung, die bisher durch Präprozessorstatements auf die konfigurierte Funktionalität verwiesen, konnten in diesem Schritt entfernt werden, da durch den Kompositionsprozess der Merkmalsorientierten Programmierung automatisch auf die entsprechende konfigurierte Funktionalität referenziert wird.

Unsere merkmalsorientierte Umsetzung wollen wir im folgendem Abschnitt bezüglich der Anpassbarkeit, Quellcodegröße, Programmcodegröße und der Performanz evaluieren.

7.3.3 Evaluierung

In diesem Abschnitt werden die Ergebnisse der Evaluierung unserer merkmalsorientierten Version von Berkeley DB vorgestellt. Im Fokus der Evaluierung stehen die Anpassbarkeit, der Quelltextumfang, die Programmgröße und Performanz.

Anpassbarkeit

Steigerung der Variabilität

Die Abbildung 7.13 stellt einen Ausschnitt des Merkmalsmodells der refaktorisierten Variante von Berkeley DB auf Basis von FEATUREC++ dar. Gezeigt werden dabei nur die optionalen und alternativen Variationspunkte. Gegenüber der originalen C-Version (11 optionale Merkmale) konnten wir 13 weitere Merkmale extrahieren und somit deren statische Konfigurierbarkeit ermöglichen. Die bisherigen Merkmale und deren Funktionsumfang wurden nicht modifiziert. Insgesamt wurde Berkeley DB in 35 Merkmale modularisiert. Basierend auf diesen Merkmalen und den *Constrains* können mehr als 400.000 verschiedene Varianten konfiguriert werden. Dies stellt eine signifikante Erweiterung des Konfigurationsspektrums dar.

Quelltextgröße

Verringerung des Quellcodeumfangs

Ein Nebeneffekt der Überführung in merkmalsorientierten Quelltext war dabei die Verringerung des Quelltextes bei gleichem Funktionsumfang um mehr als 10.000 Zeilen Quelltext. Verantwortlich hierfür waren beispielsweise der Wegfall von Funktionsdefinitionen zur Initialisierung von Funktionszeigern und die C++ API, die bisher einen Zugriff von C++ auf die C-Version ermöglichte. Tabelle 7.2 und Abbildung 7.14 zeigt die Funktion und den Quelltextumfang der wesentlichen Merkmale. Die bisher existierenden Merkmale wurden dabei nicht weiter modifiziert. Die Basis, die nicht weiter aufgeteilt wurde, umfasst 33.346 Zeilen Quellcode. Das entspricht einem Gesamtanteil am Quellcode von etwa 40 Prozent. Im Gegensatz zu den restlichen Merkmalen ist die Basisfunktionalität verpflichtend für alle Konfigurationen. Die restlichen Merkmale sind optional beziehungsweise alternativ und können somit bei Bedarf herauskonfiguriert werden. Eine weitergehende Analyse kann die Basis noch weiter verkleinern. Dies zieht aber eine umfangreichere Refaktorisierung der Basis nach sich, die im Rahmen der Untersuchung insbesondere bezüglich der Performanz ein zu großes Risiko von Fehlern in sich birgt und damit eine Vergleichbarkeit nicht uneingeschränkt zulässt. Bei den weiteren Merkmalen ist auffällig, dass insbesondere die Indexstrukturen B-Baum, *Hash*-Index und *Queue* mit knapp 31 Prozent einen auffällig großen Anteil am Quelltext haben.

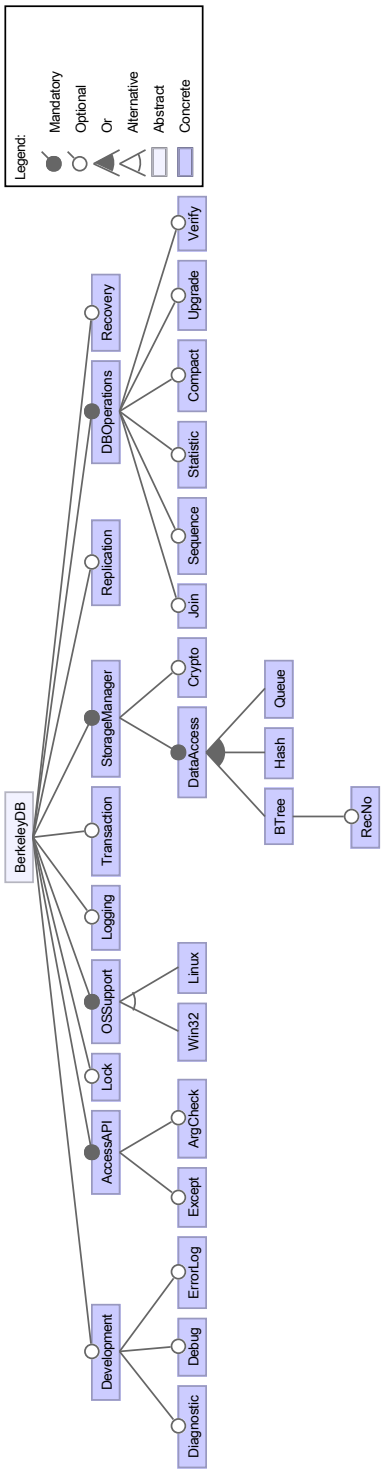

Abbildung 7.13: Ausgewählte Variationspunkte von Berkeley DB (nur alternative und optionale Merkmale)

Merkmal	Funktion	LOC	Anteil
Base	Basisfunktionalität	33.346	40,52 %
B − Tree	Indexstruktur	12.363	15,02 %
Hash	Indexstruktur	8.449	10,27 %
Replication	Replikation von Daten	6.087	7,40 %
Queue	Indexstruktur	5.009	6,09 %
TXN	Transaktionsverwaltung	3.622	4,40 %
Log	Logging	3.248	3,95 %
Crypt	Verschlüsselung	2.222	2,70 %
Verify	Konsistenzcheck für Datendateien	1.982	2,41 %
Recovery	Wiederherstellungsmanagement	2.474	3,01 %
Sonstiges	(Compact, Upgrade, Statistic, usw.)	3.487	4,24 %

Tabelle 7.2: Umfang und Anteil des Quelltextes

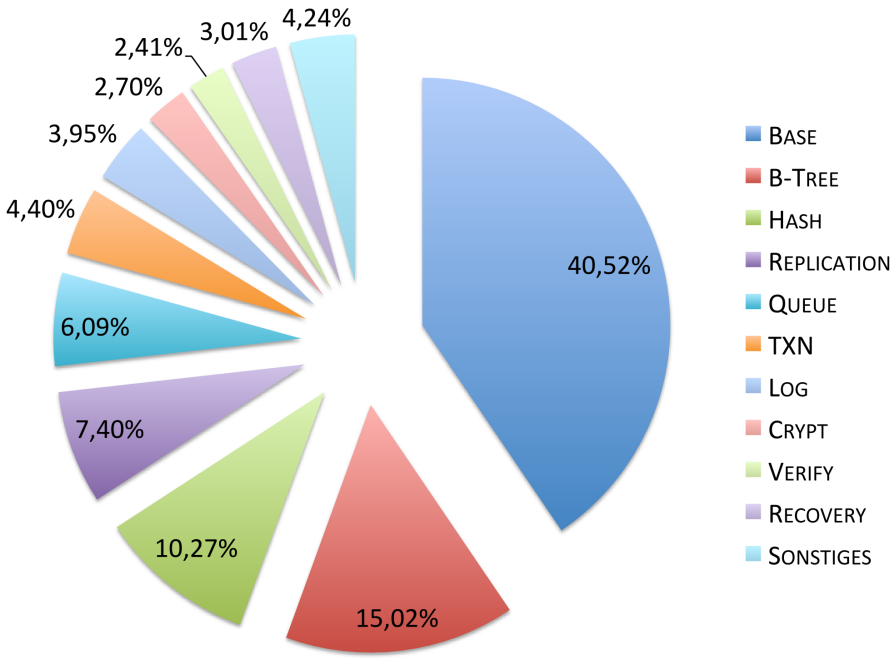

Abbildung 7.14: Anteile der Merkmale am Gesamtquelltext

Analyse des B-Baums

Mit mehr als 12.000 Zeilen Quelltext ist insbesondere der B-Baum recht umfangreich und wurde bezüglich seiner Funktionalität genauer untersucht. Die Basis des B-Baums (*B-Tree Base*) bildet eine 5.708 Zeilen Quelltext umfassende Implementierung, die nicht weiter zerlegt wurde (vgl. hierzu Tabelle 7.3). Im Gegensatz zur Originalversion konnten wir mit Hilfe der Merkmalsorientierten Programmierung auch die Merkmalsinteraktionen des *B-Tree* mit den Merkmalen *Recovery, Log, Compact, Verify, Statistic* und *Upgrade* extrahieren. Knapp 54 Prozent der 12.363 Zeilen Quellcode (vgl. hierzu Abbildung 7.15) der B-Baum-Implementierungen sind Interaktionen mit anderen Merkmalen. Durch Modularisierung dieser Merkmalsinteraktionen in Merkmale werden diese Interaktionen explizit gemacht. Dies erhöht nicht nur die Lesbarkeit des Basisquelltextes des B-Baums, sondern erlaubt auch ein besseres Verständnis hinsichtlich des Zusammenspiels mit anderen Merkmalen.

Teilmerkmal	Funktion	LOC	Anteil
$B-TreeBase$	Basisfunktionalität des B-Baums	5.708	46,17 %
$Recovery/Log$	Wiederherstellungsmanagement	3.186	25,77 %
$Compact$	Datenkomprimierung	1.611	13,03 %
$Verify$	Konsistenzcheck für Datendateien	1.457	11,79 %
$Statistic$	Indexstruktur	319	2,58 %
$Upgrade$	Versionskompatibilität	82	0,66 %

Tabelle 7.3: Umfang der B-Baumfunktionalität inklusive der Merkmalsinteraktionen

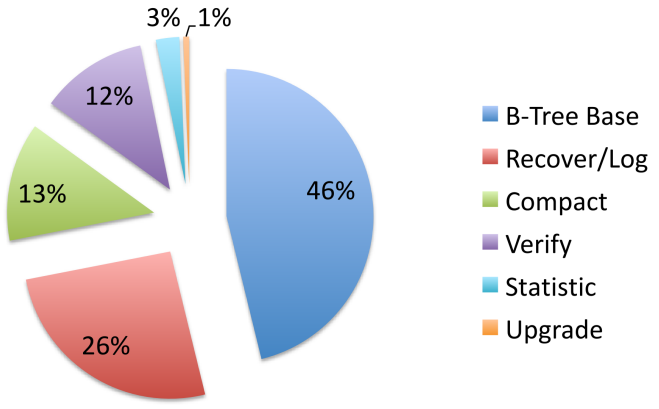

Abbildung 7.15: Anteile der Teilmerkmale am B-Baum

Größe des ausführbaren Programmcodes

Verringerung des ausführbaren Programm-codes

In Tabelle 7.4 zeigen wir die Größen des ausführbaren Programmcodes für verschiedene Konfigurationsvarianten. Für die Varianten 3 bis 8 ergeben sich minimale Unterschiede in der Größe des ausführbaren Programmcodes zu Gunsten unseres FEATUREC++ Ansatzes. Dieser Vorteil ist im Wesentlichen auf eine Besonderheit der Implementierung der C-Version zurückzuführen. Dort werden Funktionszeiger verwendet, die manuell instanziiert werden müssen, um eine Objektorientierte Programmierung nachzuahmen. Dieses Hilfskonstrukt ist hauptverantwortlich für den etwas größeren Programmcode. Ein wesentlicher Unterschied bezüglich der Programmgröße zeigen die beiden minimalen Versionen unseres Ansatzes. Die minimalen Versionen eins und zwei implementieren zum einen eine Version, die nur einen B-Baum besitzt, und zum anderen eine Variante, die zur Speicherorganisation nur eine Queue enthält. In der originalen C-Version von Berkeley DB können diese Varianten so nicht konfiguriert werden. Verglichen zur kleinen C-Version liegt die Reduktion des ausführbaren Codes bei 50 Prozent.

#	Que.	B-Tr.	TXN	Oth.	Repl.	Hash	Cry.	C in KB	FC++ in KB
1	+	-	-	-	-	-	-	N/A	184
2	-	+	-	-	-	-	-	N/A	224
3	-	+	+	-	-	-	-	416	376
4	-	+	+	+	-	-	-	492	452
5	+	+	+	+	-	-	-	528	484
6	+	+	+	+	+	-	-	580	552
7	+	+	+	+	+	+	-	644	620
8	+	+	+	+	+	+	+	664	636

Tabelle 7.4: Programmgröße von verschiedenen Merkmalskombinationen

Performanz

Benchmarktest

Für die Evaluierung der Performanz verwendeten wir den Benchmarktest[8], den Oracle zur Leistungsdemonstration von Berkeley DB anbietet. Dabei handelt es sich um einen lesenden Test, bei dem Anfragen an das System gestellt werden. Abbildung 7.16 zeigt die von einem Intel Core 2 System mit 2,4 GHz und einem Windows XP bearbeiteten Anfragen pro Sekunde der verschiedenen Konfigurationsvarianten von Berkeley DB. Die Konfigurationen entsprechen der in den Tabelle 7.4 vorgestellten Varianten. Je größer der

8 http://www.oracle.com/technology/products/berkeley-db/pdf/berkeley-db-perf.pdf

Wert, desto mehr Anfragen können von dem System pro Sekunde bearbeitet werden und desto besser ist die Performanz des Systems. Da die Konfiguration eines Systems mit

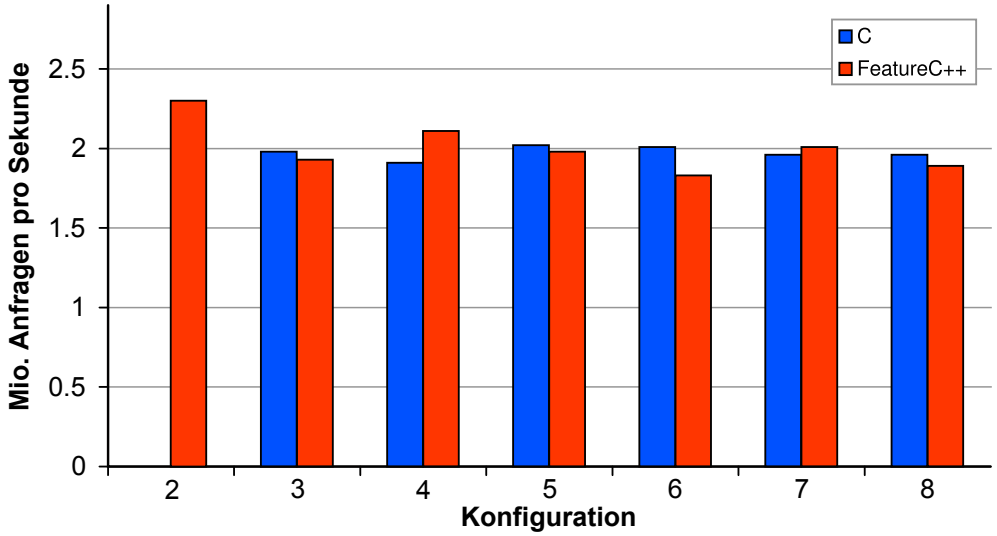

Abbildung 7.16: Performanzvergleich der C-Variante und FEATUREC++ Version von Berkeley DB

einer `Queue`[9] als Zugriffsstruktur für eine Punktanfrage nicht sinnvoll ist, wurden hierfür keine Messungen ausgeführt. Generell lässt sich erkennen, dass in den Konfigurationen drei bis acht keine signifikanten Unterschiede der Varianten nachzuweisen sind. Dies bestätigt im Wesentlichen unsere Grundannahme, dass bei sorgfältiger Wahl der durch FEATUREC++ bereitgestellten Konzepte kein Nachteil in Kauf genommen werden muss. Durch die Möglichkeit der statischen Merkmalskomposition kann auf virtuelle Methoden verzichtet werden. Des Weiteren können wir auf die Ausführung von dynamischen Laufzeitüberprüfungen, ob ein Merkmal konfiguriert ist oder nicht, verzichten. Ein Beispiel hierfür ist das Merkmal *Queue*, das in der Originalversion nur teilweise statisch konfiguriert werden kann. Abbildung 7.17 verdeutlicht dies anhand des Aufrufs der Funktion `__rep_queue_filedone` (Zeile 2). Hier wird auch bei Deaktivierung der `Queue` durch statische Konfiguration zur Laufzeit überprüft, ob es sich um eine `Queue` handelt, das zu einer Verringerung der Ausführungsgeschwindigkeit führt.

Vorteil von FEATUREC++

Der wesentliche Vorteil des FEATUREC++ Ansatzes bezüglich der Performanz zeigt sich bei der „Herauskonfiguration“ von nicht benötigter Funktionalität. In der Originalversion von Berkeley DB ist es nicht möglich, die Transaktionsverwaltung herauszukonfigurieren. Viele Einsatzszenarien im Bereich der eingebetteten Systeme benötigen auch keine Transaktionsverwaltung. In der Konfiguration zwei der Abbildung 7.16 ist

9 Die `Queue` kann im Originalsystem nur in Verbindung mit dem B-Baum konfiguriert werden.

```
1 if  (rfp->type  ==  DB_QUEUE
2      && ((ret = __rep_queue_filedone(dbenv, rep, rfp)) != DB_REP_PAGEDONE))
3 return (ret);
4  ...
```

Abbildung 7.17: Dynamische Prüfung, ob das Merkmal `Queue` in Berkeley DB konfiguriert ist

die Transaktionsverwaltung in unserer FEATUREC++ Variante von Berkeley DB deaktiviert worden. Gegenüber der minimalen C-Version kann durch diese Einstellung ein Geschwindigkeitsvorteil von 16 Prozent ermöglicht werden. Dies zeigt, dass ein wesentlicher Geschwindigkeitsvorteil durch das Weglassen von nicht benötigter Funktionalität erreicht werden kann.

7.3.4 Diskussion der Ergebnisse

Diskussion

Zur besseren Vergleichbarkeit und Demonstration der Leistungsfähigkeit von FEATUREC++ wurde in dieser Fallstudie eine refaktorisierte Version von Berkeley DB mit der originalen C-Version von Berkeley DB verglichen. Dabei stand im Vordergrund, dass der Funktionsumfang der Originalversion erhalten bleibt. Aus diesem Grund wurde nicht die komplette Funktionalität von Berkeley DB refaktorisiert. In einer Evaluierung wurde unser Ansatz bezüglich der Variabilität und Anpassbarkeit, Performanz, Programmgröße und Verständlichkeit des Codes mit der Originalversion verglichen. Im Folgenden findet sich eine Zusammenfassung der Ergebnisse:

Variabilität und Anpassbarkeit: Im ersten Schritt dieser Fallstudie wurde gezeigt, dass der bisherige Variabilitätsmechanismus mittels Präprozessoranweisungen durch den Kompositionsansatz von FEATUREC++ ersetzt werden kann. Die Originalversion von Berkely DB kann in 11 Variationspunkten angepasst werden. Dadurch sind etwa 1.000 verschiedene Varianten von Berkeley DB möglich. Die von uns präsentierte Version auf Basis von FEATUREC++ von Berkeley DB ermöglicht Variabilität in 24 Merkmalen und schafft somit mehr als 400.000 mögliche Varianten. Dies führt zu einer feingranularen Anpassungsmöglichkeit an den Anwendungskontext. Einige der neu hinzugenommenen Merkmale sind zusätzlich optional und können somit zur Verschlankung in bestimmten Anwendungskontexten beitragen.

Performanz: Ein Hauptargument für den Einsatz von C sind Performanzargumente. C++ bietet gerade in Bezug auf Variabilität mit virtuellen Methoden Konzepte an,

welche die Performanz des Systems negativ beeinflussen. Unsere Fallstudie konnte zeigen, dass bei einem wohlüberlegten Einsatz[10] von einem erweiterten objektorientierten Konzept auf Basis von FEATUREC++ die Performanz mindestens gleich zu der C-Version ist. Darüber hinaus konnte evaluiert werden, dass es in der per Präprozessor konfigurierten Version von Berkeley DB an einigen Stellen im Code dynamische Prüfungen gibt, die ergründen, ob Merkmale an- beziehungsweise ausgeschaltet sind. Durch unsere merkmalsorientierte Umsetzung mit FEATUREC++ konnten diese Überprüfungen statisch umgesetzt werden. Konfiguriert man eine Variante ohne eine Transaktionsverwaltung, führte dies zu einer Performanzverbesserung um bis zu 16 Prozent. Grundlage für die Performanzmessungen war ein Benchmarktest des Herstellers Oracle.

Programmgröße: Eine wichtige Eigenschaft von Anwendungen im Bereich der eingebetteten Systeme ist die Größe des ausführbaren Programmcodes. In unserer Fallstudie konnten wir zeigen, dass der Ansatz der Merkmalsorientierten Programmierung mittels FEATUREC++ keine negativen Auswirkungen auf die Programmgröße bei vergleichbarem Funktionsumfang besitzt. So hat beispielsweise ein voll ausgestattetes Berkeley DB-System in der C-Variante 664 KB Programmgröße und das gleiche System auf Basis von FEATUREC++ 636 KB Programmgröße. Darüber hinaus war es uns möglich, weitere Varianten von Berkeley DB zu erzeugen, die einen noch geringeren Funktionsumfang als die minimale C-Version anbieten. Während die Originalversion durch das „Herauskonfigurieren“ von Funktionalität auf ca. 62 Prozent der vollumfänglichen Lösung von Berkeley DB zu reduzieren ist, kann das refaktorisierte Berkeley DB-System auf 35 Prozent reduziert werden. Diese stark reduzierte Variante liefert aus Anwendungssicht im Benchmarktest die selben Ergebnisse.

Verständlichkeit des Codes: Ein Problem der Originalversion in Berkeley DB ist der massive Einsatz von Präprozessoranweisungen. Dies verschlechtert die Lesbarkeit und Verständlichkeit des Codes. Eine Folge davon ist, dass eine Erweiterung des Codes nur sehr schwierig zur realisieren ist. Des Weiteren ist es in der originalen Version auch nur mit sehr großem Aufwand möglich, weitere Merkmale optional zu gestalten. Dies würde den Quelltext in der Lesbarkeit weiter stark einschränken, da weitere Präprozessoranweisungen eingefügt werden müssten. Generell hat die Lesbarkeit des Codes durch die Überführung in den merkmalsorientierten Quelltext stark zugenommen und eröffnet die Möglichkeit für weitere Dekompo-

10 Wir verzichteten auf den Einsatz von virtuellen Methoden.

sitionen von Berkeley DB und damit feingranularer Konfigurierbarkeit. Dies ist vor allem durch den Wegfall von Präprozessoranweisungen und durch die bessere Trennung von Belangen realisiert worden. Des Weiteren konnten teilweise sehr lange Methoden in verständlichere Einheiten aufgetrennt werden.

Mit unserer dritten Fallstudie konnten wir zeigen, dass FEATUREC++ keine negativen Auswirkungen auf die Performanz und die Größe des Programmcodes hat.

7.4 Fazit der Fallstudien

In den vorangegangenen drei Fallstudien haben wir aus verschiedenen Blickwinkeln die Entwicklung des variablen Nanodatenmanagements untersucht. Im zweiten Kapitel der Arbeit zeigten wir, dass variables Nanodatenmanagement eine starke und feingranulare Anpassbarkeit an den Anwendungskontext unter Beachtung der streng limitierten Ressourcen von eingebetteten Systemen verlangt. Aufgabe unserer drei Fallstudien war es zu zeigen, inwieweit der von uns präsentierte Ansatz von FEATUREC++, der auf die Bedürfnisse des variablen Nanodatenmanagements für eingebettete Systeme zugeschnitten wurde, in dieser Domäne auch anwendbar ist. Für eine Analyse unserer Ergebnisse wollen wir die in der explorativen Fallstudie vorgestellten Problemfelder bezüglich unserer drei Fallstudien diskutieren:

Granularität: Die Fähigkeiten, grobgranulare Strukturen von Variationspunkten einer Produktlinie des variablen Nanodatenmanagements mit Hilfe der Merkmalsorientierten Programmierung umzusetzen, zeigten wir bereits in unserer explorativen Fallstudie. Diese Eigenschaft der Merkmalsorientierten Programmierung gilt auch uneingeschränkt in den von uns durchgeführten Fallstudien auf Basis von FEATUREC++. Hierzu lässt sich explizit auf die problemlose Überführung der vergleichsweise groben Variationsstruktur von Berkeley DB hinweisen. Als weniger untersucht galt die Eignung für sehr feingranulare Merkmalsstrukturen. Insbesondere mit unserer ROBBYDBMS-Fallstudie konnten wir zeigen, dass wir sehr feingranulare Merkmale realisieren können. Einige unserer Merkmale erweitern den Funktionsumfang des mit weniger als 100 Byte ausführbaren Programmcodes. Dennoch gibt es auch Grenzen für die Merkmalsorientierte Programmierung. Die Grenze der Granularität ist dann erreicht, wenn wir beispielsweise in Konstrukten wie Schleifen auf Statementebene mit unseren Erweiterungen in den Kontrollfluss eingreifen müssen. Hier können auch unsere *Aspectual Mixin Layer* nicht mehr eingreifen, da diese nur auf Methodensignaturen einsetzbar sind. Für die-

se wenigen Ausnahmen, die besonders für Synchronisationsbelange notwendig sind, haben wir das Konzept der *Hook*-Methoden verwendet, die explizit diese Variationspunkte invasiv kennzeichnen. Wir sind überzeugt, dass für diese wenigen Ausnahmen die Nachteile insbesondere bezüglich der nicht vollständigen Trennung von Belangen akzeptabel sind.

Overhead: Eine Vielzahl von vergleichbaren Lösungen zu FEATUREC++ im Software-Engineering-Umfeld wie beispielsweise *Adaptive Plug-and-Play Components* [ML98], *Aspectual Components* [MO04a], *Aspectual Collaborations* [LO03] und *CaesarJ* [MO04b, AGMO06] berücksichtigen nicht den *Overhead* bezüglich der Laufzeit und der Programmgröße. Unsere statischen Konfigurationsmöglichkeiten in FEATUREC++ auf der Grundlage der *Jampack*-Komposition erzeugen keinen signifikanten *Overhead* bezüglich der Laufzeit und der Programmgröße, wie wir mit unserer Berkeley DB-Studie auf Basis eines Benchmarks präsentieren konnten. Darüberhinaus konnten wir sogar zeigen, dass durch das Herauskonfigurieren von Funktionaliät, die in Berkeley DB bisher nicht komplett statisch ausgeschalten werden konnte, ein deutlicher Geschwindigkeitsvorteil bei gleicher Funktionalität erreicht werden kann.

Wiederverwendbarkeit: Insbesondere die saubere Trennung von Belangen, die FEATUREC++ bietet, ermöglicht eine leichtere Wiederverwendung von Funktionalität in verschiedenen Varianten. Das Wiederverwenden von Funktionalität innerhalb einer Produktlinie wird als geplante Wiederverwendung bezeichnet. Darüber hinaus konnten wir zeigen, dass die im FAMEDBMS-Projekt nicht anwendungsspezifisch entworfene Funktionalität auch in weiteren Szenarien eingesetzt werden kann (RobbyDBMS, *Celluar DBMS* [uR11] und *AutoDaMa* [TSP+11]). Dies verdeutlicht, dass auch ungeplant eine bessere Wiederverwendung erfolgen kann.

Querschneidende Belange: Unsere Analyse im Abschnitt 4.4.3 zeigt, dass die Merkmalsorientierte Programmierung Probleme mit homogenen querschneidenden Belangen hat. Unsere drei Fallstudien ergaben, dass nur sehr wenige homogene Erweiterungen erforderlich waren. In diesem Fall konnten wir im Bereich der Transaktionsverwaltung von FAMEDBMS *Aspectual Mixin Layer* einsetzen. Während TESANOVIC et. al [TSH04] bei ihrer Studie Berkeley DB replizierten Quellcode durch homogene Kapselung in Aspekten vermieden haben, konnten wir diese Art der querschneidenden Belange durch Konzepte der Objektorientierten und Merkmalsorientierten Programmierung vermeiden. Optionale Merkmalsin-

teraktionen in FAMEDBMS konnten durch optionales Weben im Gegensatz zum „Lifter"-Ansatz kohärent gekapselt werden.

Automatisierbarkeit der Konfiguration: Die von uns vorgestellten Ansätze des variablen Nanodatenmanagements auf Basis der Merkmalsorientierten Programmierung ermöglichen eine vollständig automatisierte Konfiguration von Varianten. Alle vorgestellten Fallstudien benötigen keine zusätzlichen Konfigurationsanweisungen im Quelltext.

Praktikabilität: Der Einsatz von FEATUREIDE stellt eine deutliche Verbesserung gegenüber der von der AHEAD-Tool-Suite bereitgestellten Unterstützung dar. Produktlinien wie etwa FAMEDBMS mit fast 200 Merkmalen können mit Hilfe von FEATUREIDE in ihrer Entwicklung beherrscht werden.

Umsetzung von ungeplanten Änderungen: Den Nachweis, das unser Ansatz eine gewisse Robustheit in Hinsicht auf ungeplante Änderung hat, konnten wir in keiner der drei Fallstudien erbringen. Dies kann in Wesentlichen auf zwei Gründe zurückgeführt werden. Zum einen sind alle drei Fallstudien im akademischen Umfeld entstanden, in dem es keine plötzlichen kundenspezifischen Änderungswünsche gibt, die in einer schnellen und preiswerten Art und Weise realisiert werden müssen. Unsere Änderungen konnten immer wohlüberlegt hinzugefügt werden. Zum anderen hat sich im Gegensatz zu unserer explorativen Studie im Laufe der Zeit ein gewisser Erfahrungsschatz angesammelt, der in der Domäne dazu führte, dass die gewählte Architektur durch die Dekompositionsform schon inhärent besser erweiterbar war. Somit konnten die Fehler, die uns in der explorativen Fallstudie unterlaufen sind, in unseren drei Fallstudien in dieser Art vermieden werden. Dennoch sind wir davon überzeugt, dass FEATUREC++ mit *Aspectual Mixin Layers* besser auf ungeplante Änderungen reagieren kann.

Mit FEATUREC++ und FEATUREIDE haben wir Werkzeuge vorgestellt, die eine Entwicklung des variablen Nanodatenmanagements in Form von Produktlinien ermöglichen. Hierzu haben wir den Produktlinienansatz der Merkmalsorientierten Programmierung in C++ erweitert und mit einer effizienten Werkzeugunterstützung für den Bereich der eingebetteten Systeme vereint. Wir konnten zeigen, dass unser Forschungsansatz auf dem Gebiet des Software-Engineering feingranular konfigurierbares und damit an die speziellen Anforderungen von eingebetteten Systemen erweiterbares Datenmanagement ermöglicht, ohne dabei die Ressourcenschonung zu vernachlässigen.

7.5 Zusammenfassung

In den vorangegangenen Abschnitten wurden die Ergebnisse der Evaluierung von drei Fallstudien aus der Domäne des Datenmanagements für eingebettete Systeme präsentiert. Ziel dieser Fallstudien ist es, die Anwendbarkeit des vorgestellten Sprachansatzes FEATUREC++ in dieser Domäne aus verschiedenen Perspektiven zu untersuchen.

Fallstudie FAMEDBMS

Die erste Fallstudie beschäftigt sich mit dem auf FEATUREC++ neu entwickelten FAMEDBMS. Der Fokus dieser Entwicklung liegt auf einer wissenschaftlich getriebenen Domänenanalyse in der Domäne der eingebetteten Systeme. Hierfür wurden mehr als 180 Variationspunkte einer Datenbankproduktlinie für den Bereich der eingebetteten Systeme analysiert und mit Hilfe von FEATUREC++ im Rahmen einer merkmalsorientierten Implementierung umgesetzt. Dabei konnte vollständig auf den Einsatz von Präprozessorstatements zur Konfiguration von Variationen der Produktlinie verzichtet werden. Obwohl FAMEDBMS auf eingebetteten Systemen lauffähig ist, stellt FAMEDBMS nur eine Referenzimplementierung dar. Auf die Maßschneiderung bezüglich konkreter Anwendungsfälle wurde verzichtet. Mit der Integration einer Transaktionsverwaltung und einer Anfrageoptimierung wurden zwei bisher schwer zu kapselnde Funktionalitäten variabel gestaltet. Bis auf einige wenige Ausnahmen waren die Erweiterungen auch in diesem Umfeld heterogener Natur. Da sowohl die Anfrageoptimierung als auch die Transaktionsverwaltung mit vielen optionalen Merkmalen kann hier die Methode des optionalen Webens eingesetzt werden.

Fallstudie ROBBYDBMS

In Fallstudie zwei diskutieren wir die Produktlinie ROBBYDBMS, die Datenmanagementfunktionalität für autonome, mobile Roboter auf der Basis der AVR-Hardware-Produktlinie von Atmel bereitstellt. Die Basis und die grundlegende Architektur für unsere Produktlinie bildet FAMEDBMS. Auf Grund der sehr restriktiven Hardware-Anforderungen, die den Programmspeicherplatz für die Anwendung, Parameterdaten und unsere Datenmanagementfunktionalität auf 32 bis 128 KB einschränken, ist der sparsame Umgang mit Ressourcen enorm wichtig. Die 42 Variationspunkte von ROBBYDBMS ermöglichen eine feingranulare Anpassung an den Anwendungskontext. Unsere minimalen Varianten mit abstrahiertem Lese- und Schreibzugriff über eine API benötigen weniger als 1 KB Programmspeicher. Die vollumfängliche Konfiguration benötigt nicht mehr als 19 KB Programmspeicher und umfasst unter anderem neben mehrdimensionalen Indexstrukturen, einer dynamischen Speicherverwaltung, einer einfachen Transaktionsverwaltung auch eine *Recovery*-Komponente. Die vergleichsweise feingranularen Erweiterungen, unter Berücksichtigung von Ressourcenbeschränkungen, führten zu den konzeptbedingten Grenzen von FEATUREC++, das sowohl im

merkmalsorientierten Kern als auch in den aspektorientierten Erweiterungen keine Möglichkeiten für Erweiterungen unterhalb der Methodenebene vorsieht. Die von uns skizzierte Lösung durch *Hook*-Methoden bietet durch die explizite Kennzeichnung eines Einsprungpunktes in der zu erweiternden Klasse eine mögliche Lösung, um auch auf Anweisungsebene in den Kontrollfluss des zu erweiternden Quelltextfragments einzugreifen. Durch den Einsatz von FEATUREC++ in Kombination mit *Hook*-Methoden kann vollständig auf Präprozessoranweisungen zur ressourcenschonenden, feingranularen Erweiterung von Merkmalen verzichtet werden.

Fallstudie Berkeley DB

Die dritte Fallstudie Berkeley DB zeigt anhand des in der Praxis etablierten Berkeley DB-Systems, dass durch den Einsatz der Merkmalsorientierten Programmierung von FEATUREC++ keine Performanznachteile gegenüber des klassischen Ansatzes der statischen Konfiguration vom C-Quelltext durch den CPP in Kauf genommen werden müssen. Hierzu wurde zunächst eine teilweise automatisierte Refaktorisierung des C-Quelltextes der Originalversion in objektorientierten Quelltext vorgenommen. Auf Grundlage dieser Basis wurde für einen Teil der Funktionalität eine merkmalsorientierte Refaktorisierung des Quelltextes realisiert. Die Variabilität von Berkeley DB konnte von etwa 1.000 möglichen Varianten auf mehr als 400.000 mögliche Konfigurationen gesteigert werden. Auf Basis dieser neuen Variabilität konnten wir in unserem merkmalsorientierten Ansatz von Berkeley DB die Größe des ausführbaren Programmcodes der minimalen Variante von 62 auf 35 Prozent der vollumfänglichen Version senken. Darüber hinaus konnten wir zeigen, dass durch die vollständige Optionalisierung des Merkmals Transaktionsverwaltung in einem Benchmarktest des Herstellers Geschwindigkeitsvorteile von 16 Prozent gegenüber der Originalversion erreicht werden können, in der diese Funktionalität nicht variabel gestaltet ist.

KAPITEL 8

Zusammenfassung und Ausblick

Problemfeld

Die Verbreitung von eingebetteten Systemen, die ausgewählte maßgeschneiderte Datenmanagementfunktionalität benötigen, hat in den letzten Jahren stark zugenommen. Auf Grund der eingeschränkten Ressourcen brauchen diese Systeme stark auf den Anwendungskontext zugeschnittenes Datenmanagement, das ohne großen *Overhead* in die Anwendung des eingebetteten Systems integriert werden kann. Wir bezeichnen dieses als *Nanodatenmanagement*. Bisherige Lösungen sind entweder für einen sehr kleinen Bereich konzipiert oder in ihrer Variabilität nicht ausreichend feingranular anpassbar.

Ziele der Dissertation

Ziel dieser Arbeit ist die Anpassung vorhandener Software-Engineering-Methoden an die Bedürfnisse des variablen Nanodatenmanagements. Hierzu zeigen wir, dass Variabilität in Form von feingranularen Merkmalen in einer Produktlinie unter Berücksichtigung der speziellen Ressourcenrestriktionen organisiert werden kann. Im Fokus der Arbeit steht dabei die Anpassung des Konzepts der Merkmalsorientierten Programmierung an die Bedürfnisse des variablen Nanodatenmanagements. Diese Dissertation präsentiert hierzu unsere Forschungsarbeiten der letzten Jahre in chronologischer Reihenfolge, beginnend mit einer explorativen Studie, um unsere Beweggründe für einige unserer Entscheidungen im zeitlichen Verlauf besser darzustellen.

8.1 Zusammenfassung der Arbeit

Kapitel 2

Im zweiten Kapitel zeigten wir einen Überblick zum Thema Datenmanagement für eingebettete Systeme und führten zunächst in die wichtigsten Eigenschaften von eingebetteten Systemen ein. Anschließend präsentierten wir anhand von Beispielszenarien, unter anderem aus der Automobilherstellung, Anforderungen an das Datenmanagement für eingebettete Systeme. Geschlossen wird das Kapitel mit einem Literaturüberblick zu

variablen und anpassbaren Datenmanagementsystemen und dem aktuellen Datenmanagement für ressourcenbeschränkte Umgebungen.

Kapitel 3 Im dritten Kapitel führten wir in die Grundbegriffe der Variabilitätsgestaltung und Wiederverwendung von Softwareeinheiten in der Softwareentwicklung ein. Fokussiert wurde dabei auf Definitionen, die für die Arbeit notwendig sind.

Kapitel 4 Mit Hilfe einer explorativen Studie evaluierten wir im vierten Kapitel den Einsatz der Merkmalsorientierten Programmierung mit Hilfe der AHEAD-Tool-Suite zur Umsetzung eines Speichermanagers am Beispiel einer idealisierten Fallstudie aus dem Bereich der Sensornetzwerke. Prinzipiell eignete sich diese Umsetzungsmethode zur Variabilitätsgestaltung durch die Merkmalsorientierte Programmierung für die feingranulare Gestaltung von Merkmalen unseres Speichermanagers. Unsere Analyse ergab, dass es dennoch einige Herausforderungen zu überwinden gibt. Dies sind unter anderem die Merkmalskohäsion von optionalen Merkmalen, nicht-hierarchiekonforme Änderungen, homogene Erweiterungen, komplexe, dynamische Erweiterungen und Probleme bei der Praktikabilität.

Kapitel 5 Im Kapitel fünf präsentierten wir FEATUREC++, eine Umsetzung der Merkmalsorientierten Programmierung für C++ in Anlehnung an die AHEAD-Tool-Suite, das speziell auf unsere Bedürfnisse im Bereich des eingebetteten Datenmanagements zugeschnitten wurde. Nach einer Analyse möglicher Umsetzungstechniken auf Basis von existierenden Ansätzen präsentierten wir eine merkmalsorientierte Umsetzung. Diese Erweiterungen ermöglichen einen besseren Umgang mit homogenen querschneidenden Erweiterungen, nicht-hierarchiekonformen Erweiterungen aus ungeplanten Änderungen und eine Kapselung von Merkmalsinteraktionen bezüglich optionaler Merkmale.

Kapitel 6 Nicht alle Probleme der Merkmalsorientierten Programmierung konnten auf Sprachebene gelöst werden. Im sechsten Kapitel stellten wir deshalb FEATUREIDE auf Basis von Eclipse als eine integrierte Entwicklungsumgebung zur Unterstützung der Merkmalsorientierten Softwareentwicklung vor. FEATUREIDE ermöglicht einen praktikablen Umgang mit unseren Datenmanagementproduktlinien.

Kapitel 7 Im Kapitel sieben präsentierten wir drei verschiedene Fallstudien, die unseren Sprachansatz FEATUREC++ im Bereich des hochkonfigurierbaren Datenmanagements für eingebettete Systeme evaluieren. In der ersten Fallstudie FAMEDBMS präsentierten wir die Implementierung eines Speichermanagers, der um eine variable Transaktionsverwaltung und einen Anfrageoptimierer erweitert werden konnte. Bei der Implementierung von FAMEDBMS sind sowohl in der Transaktionsverwaltung als auch in der Anfrageoptimie-

rung keine homogenen Erweiterungen aufgetreten. Im Gegensatz zur zweiten Studie ROBBYDBMS ist die FAMEDBMS-Implementierung als eine Referenzimplementierung ausgelegt. Bei der Implementierung sind keine homogenen Erweiterungen aufgetreten. Mit ROBBYDBMS setzten wir auf Basis der FAMEDBMS-Implementierung einen Speichermanager für tief eingebettete Systeme im Bereich der mobilen autonomen Roboter mit einer extremen Ressourceneinschränkung um. 42 Variationspunkte ermöglichen eine feingranulare Anpassung an den Anwendungskontext. Die minimalen Varianten mit abstrahiertem Lese- und Schreibzugriff über eine API benötigen weniger als 1 KB Programmspeicher. Eine vollkonfigurierte Variante von ROBBYDBMS braucht maximal 19 KB Programmspeicher. Durch die vergleichsweise feingranularen Erweiterungen zeigte diese Studie auch die konzeptbedingten Grenzen von FEATUREC++, die unterhalb der Methodenebene keine Erweiterungen ermöglicht. Durch den Einsatz von FEATUREC++ in Kombination mit Hook-Methoden kann vollständig auf Präprozessoranweisungen zu ressourcenschonenden, feingranularen Erweiterungen von Merkmalen verzichtet werden. In der dritten Studie wurde Berkeley DB auf Basis von FEATUREC++ merkmalsorientiert refaktorisiert. In einem vergleichenden Benchmark wurde dann die originale Version von Berkeley DB auf der Basis von C mit unserer refaktorisierten verglichen, die eine bessere Variabilität bezüglich der Performanz aufweist.

8.2 Wissenschaftlicher Beitrag der Arbeit

Der wissenschaftliche Beitrag der Arbeit kann in zwei Bereiche aufgeteilt werden. Zum einen evaluieren wir Produktlinientechniken für variables Nanodatenmanagement in eingebetteten Systemen in unterschiedlichen Ausprägungen und zum anderen passen wir bestehende Konzepte des Software-Engineering für die Domäne des variablen Nanodatenmanagements für eingebettete Systeme an.

Produktlinientechniken für variables Nanodatenmanagement in eingebetteten Systemen

Der von uns vorgestellte Ansatz geht über die Ansätze von anpassbaren und maßgeschneiderten DBMS aus den achtziger und neunziger Jahren insofern hinaus, dass wir eine feingranulare Konfiguration von Datenmanagementfunktionalität ermöglichen. Dies führt zu einer exakteren Anpassung an die Anwendungsszenarien der eingebetteten Systeme. Mit dieser Dissertation leisten wir einen Beitrag zum Einsatz von Datenmanagementfunktionalität in ressourcenbeschränkten Umgebungen und geben Lösungsmöglichkeiten für Fragestellungen, die in den Diskussionen von CHAUDURI

und WEIKUM [CW00], OLSON [Ols00] und STONEBRAKER sowie CETINTEMEL [Sc05] aufgeworfen wurden. Wir nutzen hierzu, nach unseren Kenntnissen als erste, Produktlinientechniken, wie etwa die merkmalsorientierte Domänenanalyse und modellieren systematisch mögliche Variationspunkte von unserem Nanodatenmanagement. Diese explizite Darstellung von Variationspunkten unserer Referenzimplementierung FAMEDBMS erlaubt uns und anderen, nun die Diskussion über Varianten und alternative Implementierungen von Datenmanagementfunktionalität vergleichbar zu führen. Mit dem von uns gewählten Ansatz, Merkmale schrittweise um neue Funktionalität zu erweitern und diese Erweiterungen in einem Merkmal zu kapseln, können wir die Komplexität eines Merkmals exakter bestimmen, da alle Erweiterungen, die zur Umsetzung dieses Merkmals benötigt werden, kohärent in dieser Erweiterung zusammengehalten werden. Dies erhöht das Verständnis. Mit ROBBYDBMS konnten wir zeigen, dass sehr feingranulare Zerlegungen von Datenmanagementfunktionalität mit wenigen Kilobyte möglich sind. Die Berkeley DB-Studie zeigt, dass unsere Konzepte, die wir in FEATUREC++ vereint haben, zu Variabilität ohne zusätzliche Performanznachteile oder zu einer Vergrößerung des Programmcodes führen.

Merkmalsorientierte Programmierung in C++ für eingebettete Systeme

Konträr zu anderen Lösungen (wie beispielsweise PICO DBMS, COMET DBMS, Berkeley DB, TinyDB, Celluar DBMS, Prevayler), die den Funktionsumfang oder neue Algorithmen und Verfahren ihrer (variablen) Datenmanagementlösungen für eingebettete Systeme in den Vordergrund stellen, fokussiert unsere Arbeit stärker auf softwaretechnische Fragestellungen. Produktlinien, Variabilität und die klare Trennung von Belangen sind im Software-Engineering seit Jahren Gegenstand der Forschung. Viele der neu angebotenen Konzepte fokussieren nicht auf Performanz, *Footprint* oder hardware-nahe Programmierung. Einige wenige Ansätze wie etwa AspectC++ werden von der Betriebssystemforschung weiterentwickelt. Auch die von uns entwickelte Lösung FEATUREC++ berücksichtigt die Performanz und *Footprint* im Kontext des variablen Nanodatenmanagements für eingebettete Systeme. In dieser Arbeit leisten wir mit FEATUREC++ einen Beitrag zur Anpassung von hierarchischen, kollaborationenbasierten Kompositionsmechanismen, um die im Bereich des variablen Nanodatenmanagements notwendigen feingranularen Variationspunkte einer Produktlinie umzusetzen. Wir zeigen durch unsere Integration von Aspekt- und Merkmalsorientierter Programmierung in FEATUREC++ Lösungen, wie auf Basis der systemnahen Programmiersprache C++ Probleme von homogenen, komplexen und nicht-hierarchiekonformen Erweiterungen behandelt werden können. Des Weiteren präsentieren wir eine Lösung für das Problem von optionalen

Merkmalen in hierarchischen Dekompositionen, die im Gegensatz zu existierenden Lösungen wie etwa PREHOFERS „Lifter“ alle Quelltextfragmente unseres Merkmals kohärent in einem Merkmal kapseln können. Für die Handhabung großer feingranularer Produktlinien mit unserem merkmalsorientierten Ansatz FEATUREC++ ist eine Werkzeugunterstützung, wie wir sie mit FEATUREIDE präsentierten, unumgänglich.

8.3 Ausblick auf Folgearbeiten

In dem letzten Abschnitt dieser Dissertation wollen wir interessante Themen für weitere Forschungsarbeiten aufzeigen, die sich während der Erstellung dieser Arbeit ergaben.

Automatisches Ableiten von Konfigurationen

Produktlinien mit mehreren 100 Merkmalen benötigen bei der Auswahl von Merkmalen ein hohes Maß an Fachkenntnissen über die Eigenschaften dieser Merkmale. Wir glauben, dass der Entwickler einer eingebetteten Anwendung dabei unterstützt werden kann, indem Eigenschaften der Anwendung automatisiert erhoben werden. Hierzu müssen die Funktionen der Anfrage-API beziehungsweise der Funktionsumfang des verwendeten SQLs automatisiert ausgewertet werden. Des Weiteren müsste untersucht werden, inwieweit bestimmte Interaktionsmuster Rückschlüsse auf den benötigten Funktionsumfang zulassen. Zudem kann eine Analyse der in der Anwendung verwendeten Bibliotheken und Betriebssystemaufrufe Rückschlüsse auf den Funktionsumfang der benötigten Datenmanagementfunktionalität ermöglichen. Dass aus solchen Informationen Konfigurationsentscheidungen gewonnen werden können, zeigt in Ansätzen der ACCORD in COMET DBMS [TNHN03].

Berücksichtigung von nicht-funktionalen Eigenschaften

In dieser Arbeit haben wir nur die funktionalen Eigenschaften des Datenmanagements in eingebetteten Systemen untersucht. Ein wesentlicher Faktor für variable Strukturen sind aber auch nicht-funktionale Eigenschaften wie etwa Sicherheit, Robustheit, Effizienz, Performanz, Energieeffizenz oder Echtzeitfähigkeit. Diese zusätzlichen Parameter können eine automatische Konfiguration unterstützen. Arbeiten von SIEGMUND et al. [SRK+12] zeigen, wie automatisierte Aussagen zu bestimmten nicht-funktionalen Eigenschaften einer Konfiguration auf Basis von statischen Abschätzungen vorhergesagt werden können.

Variable Anfrageoptimierung

CHAUDURI und WEIKUM [CW00] als auch STONEBRAKER und CETINTEMEL [Sc05] diskutieren das Ende von *General-Purpose*-Systemen. Insbesondere Selbst-Tuning-Mechanismen benötigen eine saubere Trennung von Belangen und feingranulare Strukturen. Unsere Fallstudie von FAMEDBMS zeigt, dass die Anfrageoptimierung bei einer sehr feingranularen Zerlegung eine Herausforderung ist. Das Entwerfen einer Anfrageoptimierung ist von Grund auf für eine umfangreiche Untersuchung nicht zu empfehlen. Stattdessen sollte der Anfrageoptimierer eines bestehenden Systems merkmalsorientiert zerlegt werden, um angelehnt an unsere Fallstudie Berkeley DB mit Benchmarks die Effekte einzelner Konfigurationen zu messen. Eine erste Analyse von Merkmalen der Anfrageoptimierung bestehender Systeme präsentierten wir in SOFFNER et al. [SSR+12]. Wir glauben, dass der Einsatz von domänenspezifisch angepassten Anfrageoptimierern signifikante Vorteile bei der Anfrageoptimierung bringt.

Programmverständnis

Ein wesentliches Thema, das wir in unseren Diskussionen stets ausgeblendet haben, ist der Nachweis, dass eine saubere Trennung von Belangen und eine kohärente Kapselung von Funktionalität eines Merkmals durch die Merkmalsorientierte Programmierung zu einem besseren Programmverständnis führt. In einigen Studien haben wir hierzu vergleichend zu C-Präprozessoransätzen erste Daten gesammelt [FKA+12]. Insgesamt stehen diese Untersuchungen erst am Anfang. Einen neuen Ansatz für einen Nachweis kann die funktionelle Magnetresonanztomographie (fMRT) bieten, wie wir in SIEGMUND et al. [SBA+13] beschreiben. Dort wollen wir mittels bildgebender Verfahren herausfinden, welche Hirnareale für das Programmverständnis benötigt werden. Dies soll Rückschlüsse zur optimalen Gestaltung der Sprach- und Werkzeugunterstützung zur Handhabbarkeit auf Implementierungsebene in großen Produktlinien ermöglichen.

Werkzeugunterstützung und Synergien zu Präprozessoransätzen

Die Werkzeugunterstützung, die wir in FEATUREIDE bündeln, wird weltweit für die Umsetzung von Produktlinien verwendet. Des Weiteren werden nicht nur Sprachansätze der Merkmalsorientierten Programmierung durch FEATUREIDE unterstützt, sondern auch Präprozessoransätze. Im Bereich der C und C++ Umsetzungen dominieren derzeitig Präprozessoransätze. Um einen praktikablen Übergang zu FEATUREC++ zu ermöglichen, benötigen wir Werkzeuge zur automatischen oder zumindest semiauto-

matischen Überführung von Präprozessoransätzen. Hierzu könnte eine Integration der Präprozessorunterstützung CIDE [Käs07b, KAK08] einen Lösungsansatz bieten.

Literaturverzeichnis

[ABB+04] Arasu, A.; Babcock, B.; Babu, S.; Cieslewicz, J.; Datar, M.; Ito, K.; Motwani, R.; Srivastava, U.; Widom, J.: STREAM: The Stanford Data Stream Management System. Technical Report Nr. 2004-20, Stanford InfoLab, 2004.

[AC04] Antkiewicz, M.; Czarnecki, K.: FeaturePlugin: Feature Modeling Plug-In for Eclipse. In *ETX*, S. 67–72, 2004.

[AGMO06] Aracic, I.; Gasiunas, V.; Mezini, M.; Ostermann, K.: An Overview of CaesarJ, Band 3880. S. 135–173, 2006.

[AK09] Apel, S.; Kästner, C.: An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, Band 8, Nr. 5, S. 49–84, 2009.

[AKL09] Apel, S.; Kästner, C.; Lengauer, C.: FEATUREHOUSE: Language-Independent, Automated Software Composition. In *ICSE*, S. 221–231, 2009.

[ALM06] Angenvoort, J.; Loewenau, J. P.; Mezger, K.: Real Time Map Information with an Standardized Interface For Advanced In-Vehicle Applications. In *13th World Congress on ITS, London,*, S. 365–394, 2006.

[ALRS05] Apel, S.; Leich, T.; Rosenmüller, M.; Saake, G.: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *GPCE*, S. 125–140, 2005.

[ALS06] Apel, S.; Leich, T.; Saake, G.: Aspectual Mixin Layers: Aspects and Features in Concert. In *ICSE*, S. 122–131. ACM Press, 2006.

[ALS08] Apel, S.; Leich, T.; Saake, G.: Aspectual Feature Modules. *IEEE Transactions on Software Engineering*, Band 34, Nr. 2, S. 162–180, 2008.

[Ape07] Apel, S.: *The role of features and aspects in software development: similarities, differences, and synergetic potential*. Dissertation, 2007.

[AWSC02] Akyildiz, I. F.; Weilian, S.; Sankarasubramaniam, Y.; Cayirci, E. E.: A survey on sensor networks. *IEEE Communications Magazine*, Band 40, Nr. 8, S. 102–114, 2002.

[BAC+81] Blasgen, M. W.; Astrahan, M. M.; Chamberlin, D. D.; Gray, J.; III, W. F. K.; Lindsay, B. G.; Lorie, R. A.; Mehl, J. W. et al.: System r: An architectural overview. *IBM Systems Journal*, Band 20, Nr. 1, S. 41–62, 1981.

[Bar99] Barr, M.: Programming Embedded Systems in C and C++: Thinking Inside the Box. S. 174, 1999.

[Bat04] Batory, D.: *A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite (ATS)*. Department of Computer Sciences University of Texas at Austin, Austin, Texas, 78712 U.S.A., 2004.

[Bat05] Batory, D. S.: Feature Models, Grammars, and Propositional Formulas. In *SPLC*, S. 7–20, 2005.

[Bat86] Batory, D. S.: GENESIS: A Project to Develop an Extensible Database Management System. In *Proc. Int'l Workshop on Object-Oriented Database Sys.*, S. 207. Pacific Grove, CA, 1986.

[Bau03] Bauer, H. (Hrsg.): *Ottomotor-Management: Motronic Systeme: Systemübersichten, Elektronische Steuerung und Regelung, Elektronische Diagnose, Steuergeräteentwicklung*. 2003.

[BBD+01] Bercken, J. V.; Blohsfeld, B.; Dittrich, J.-P.; Krämer, J.; Schäfer, T.; Schneider, M.; Seeger, B.: XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, S. 39–48. Morgan Kaufmann, 2001.

[BBD+04] Babcock, B.; Babu, S.; Datar, M.; Motwani, R.; Thomas, D.: Operator Scheduling in Data Stream Systems. *The VLDB Journal*, Band 13, Nr. 4, S. 333–353, 2004.

[BBG+88] Batory, D.; Barnett, J.; Garza, J.; Smith, K.; Tsukuda, K.; Twichell, B.; Wise, T.: GENESIS: An Extensible Database Management System. *TSE*, Band 14, Nr. 11, S. 1711–1730, 1988.

[BBPV00] Bobineau, C.; Bouganim, L.; Pucheral, P.; Valduriez, P.: PicoDMBS: Scaling Down Database Techniques for the Smartcard. In El Abbadi, A.; Brodie, M. L.; Chakravarthy, S.; Dayal, U.; Kamel, N.; Schlageter, G.; Whang, K.-Y. (Hrsg.): *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10–14, 2000, Cairo, Egypt*, S. 11–20. Morgan Kaufmann Publishers, 2000.

[BC90] Bracha, G.; Cook, W.: Mixin-based inheritance. In *Conference on Object Oriented Programming Systems Languages and Applications*, S. 303 – 311. ACM Press, 1990.

[BCC+05] Bloomfield, R.; Cazin, J.; Craigen, D.; Juristo, N.; Kesseler, E.; Voas, J.: Validation, Verification and Certification of Embedded Systems . Technical Report, RTO-TR-IST-027 AC/323(IST-027)TP/3, NATO Research and Technology Organisation, 2005.

[Beu03] Beuche, D.: *Composition and Construction of Embedded Software Families*. Dissertation, University of Magdeburg, Germany, 2003.

[BGP+99] Beuche, D.; Guerrouat, A.; Papajewski, H.; Schröder-Preikschat, W.; Spinczyk, O.; Spinczyk, U.: The PURE family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *In 2nd IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC 99)*, S. 45–53, 1999.

[Bil92] Biliris, A.: An Efficient Database Storage Structure for Large Dynamic Objects. In Golshani, F. (Hrsg.): *ICDE*, S. 301–308. IEEE Computer Society, 1992.

[BLHM02] Batory, D.; Lopez-Herrejon, R. E.; Martin, J.-P.: Generating Product-Lines of Product-Families. In *17 th IEEE International Conference on Automated Software Engineering (ASE'02)*, 2002.

[BO92] Batory, D.; O'Malley, S.: The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Band 1, S. 355 – 398, 1992.

[Boe81] Boehm, B.: *Software Engineering Economics*. Prentice Hall, 1981.

[BP95] Biliris, A.; Panagos, E.: A High Performance Configurable Storage Manager. In Yu, P. S.; Chen, A. L. P. (Hrsg.): *ICDE*, S. 35–43. IEEE Computer Society, 1995.

[BP96] Biliris, A.; Panagos, E.: The BeSS Object Storage Manager: Architecture Overview. *SIGMOD Record*, Band 25, Nr. 3, S. 53–58, 1996.

[BPC+07] Baronti, P.; Pillai, P.; Chook, V. W. C.; Chessa, S.; Gotta, A.; Hu, Y. F.: Wireless sensor networks: A survey on the state of the art and the 802.15.4 and ZigBee standards. *Comput. Commun.*, Band 30, Nr. 7, S. 1655–1695, 2007.

[BPSP03] Beuche, D.; Papajewski, H.; Schröder-Preikschat, W.: Variability Management with Feature Models. In *Proceedings of the Workshop on Software Variability Management*, S. p. 72–83, 2003.

[Bro75] Brooks, R.: A Model Of Human Cognitive Behavior in Writing Code For Computer Programs. In *IJCAI*, S. 878–884, 1975.

[BS06] Braess, H.-H.; Seiffert, U.: *Vieweg Handbuch Kraftfahrzeugtechnik*. Vieweg + Teubner, 2006.

[BSR03] Batory, D. S.; Sarvela, J. N.; Rauschmayer, A.: Scaling Step-Wise Refinement. In *ICSE*, S. 187–197, 2003.

[BSST93] Batory, D. S.; Singhal, V.; Sirkin, M.; Thomas, J.: Scalable Software Libraries. In *SIGSOFT FSE*, S. 191–199, 1993.

[BT97] Batory, D. S.; Thomas, J.: P2: A Lightweight DBMS Generator. *J. Intell. Inf. Syst.*, Band 9, Nr. 2, S. 107–123, 1997.

[BW01] Babu, S.; Widom, J.: Continuous Queries Over Data Streams. *SIGMOD Rec.*, Band 30, Nr. 3, S. 109–120, 2001.

[CC04] Colyer, A.; Clement, A.: Large-scale AOSD for middleware. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD, S. 56–65. ACM, New York, NY, USA, 2004.

[CcC+02] Carney, D.; Çetintemel, U.; Cherniack, M.; Convey, C.; Lee, S.; Seidman, G.; Stonebraker, M.; Tatbul, N.; Zdonik, S.: Monitoring Streams: A new Class of Data Management Applications. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, S. 215–226. VLDB Endowment, 2002.

[CD87] Carey, M. J.; DeWitt, D. J.: An overview of the EXODUS project. *Database Engineering*, Band 10, Nr. 2, S. 107–114, 1987.

[CDKK85] Chou, H.-T.; Dewitt, D. J.; Katz, R. H.; Klug, A. C.: Design and Implementation of the Wisconsin Storage System. *Softw. Pract. Exper.*, Band 15, Nr. 10, S. 943–962, 1985.

[CDRS86] Carey, M. J.; DeWitt, D. J.; Richardson, J. E.; Shekita, E. J.: Object and File Management in the EXODUS Extensible Database System. In Kambayashi, Y.; Chu, W.; Gardarin, G.; Ohsuga, S. (Hrsg.): *Twelfth international conference on very large data bases, proceedings (VLDB '86)*, S. 91–100. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1986.

[CE00] Czarnecki, K.; Eisenecker, U. W.: *Generative Programming Methods, Tools and Applications*. Addison-Wesley, 2000.

[Cha12] Chandrakasan, A.: Research - What is a Sensor Network?, 2012. http://www-mtl.mit.edu/researchgroups/icsystems/uamps/-research/sensornet.shtml .

[Chi95] Chiba, S.: A Metaobject Protocol for C++. *ACM SIGPLAN Notices*, Band 30, Nr. 10, S. 285–299, 1995.

[CK96] Choi, K.-R.; Kim, K.-C.: T*-Tree: A Main Memory Database Index Structure for Real Time Applications. *Real-Time Computing Systems and Applications, International Workshop*, Band 0, S. 81, 1996.

[CL01] Cardone, R.; Lin, C.: Comparing Frameworks and Layered Refinement. In *ICSE*, S. 285–294, 2001.

[CLN02] Clements, P.; Linda Northrop, L.: *Software Product Lines: Practices and Patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001, 2002.

[CM93] Chiba, S.; Masuda, T.: Designing an Extensible Distributed Language with a Meta-Level Architecture. In Nierstrasz, O. (Hrsg.): *ECOOP'93—Object-Oriented Programming, 7th European Conference*, Lecture Notes in Computer Science, Band 707, S. 482–501. Springer, Kaiserslautern, Germany, 1993.

[Com00] Company, B. C.: Future of Embedded Systems Technology, 2000. BCC Press release on market study RG-229.

[CRM98] Casparsson, L.; Rajnak, K. T. A.; Malmberg, P.: Volcano - A Revolution in On-Board Communication. Technical re-

port, http://www.cs.bu.edu/pub/ieee-rts/articles/Casparsson-Volcano, 1998.

[CW00] Chaudhuri, S.; Weikum, G.: Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *VLDB*, S. 1–10, 2000.

[DG00] Dittrich, K. R.; Geppert, A.: *Component Database Systems*. Morgan Kaufmann, 2000.

[DH96] Driesen, K.; Hölzle, U.: The Direct Cost of Virtual Function Calls in C++. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, S. 306–323. ACM, 1996.

[Dij72] Dijkstra, E. W.: The Humble Programmer. *Commun. ACM*, Band 15, Nr. 10, S. 859–866, 1972. Turing Award Lecture.

[Dij76] Dijkstra, E.: *A Discipline of Programming.* Prentice-Hall, 1976.

[Dre07a] Dressler, F.: Bio-inspired Network-centric Operation and Control for Sensor/Actuator Networks. *Springer Transactions on Computational Systems Biology (TCSB)*, Band VIII, Nr. LNCS 4780, S. 1–13, 2007.

[Dre07b] Dressler, F.: *Self-Organization in Sensor and Actor Networks*. John Wiley & Sons, 2007.

[Dre08] Dressler, F.: Bio-Inspired Networking - Self-organizing Networked Embedded Systems. In Würtz, R. P. (Hrsg.): *Organic Computing*, S. 285–302. Springer, 2008.

[EBC00] Eisenecker, U. W.; Blinn, F.; Czarnecki, K.: A Solution to the Constructor Problem of Mixin-Based Programming. In *In First Workshop on C++ Template Programming*, 2000.

[EGHK99] Estrin, D.; Govindan, R.; Heidemann, J. S.; Kumar, S.: Next Century Challenges: Scalable Coordination in Sensor Networks. In *MOBICOM*, S. 263–270, 1999.

[FBB93] Ford, S.; Blakeley, J. A.; Bannon, T. J.: Open OODB: a modular object-oriented DBMS. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, S. 552–553. ACM, New York, NY, USA, 1993.

[FKA+12] Feigenspan, J.; Kästner, C.; Apel, S.; Liebig, J.; Schulze, M.; Dachselt, R.; Papendieck, M.; Leich, T. et al.: Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empirical Software Engineering*, 2012.

[GD87] Graefe, G.; DeWitt, D. J.: The EXODUS optimizer generator. In Dayal, U.; Traiger, I. (Hrsg.): *Proceedings of Association for Computing Machinery Special Interest Group on Management of Data 1987 annual conference, San Francisco, May 27–29, 1987*, S. 160–172. ACM Press, New York, NY 10036, USA, 1987.

[GGP+03] Ganesan, D.; Greenstein, B.; Perelyubskiy, D.; Estrin, D.; Heidemann, J.: An Evaluation of Multi-resolution Storage for Sensor Networks. In *Proceedings of the ACM SenSys Conference*, S. 89–102. ACM, Los Angeles, California, USA, 2003.

[GHJV96] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1996.

[Git08] Gittler, M.: Merkmalsorientierte Refaktorisierung von Berkeley DB. Diplomarbeit, Universität Magdeburg, Germany, 2008.

[GM93] Graefe, G.; McKenna, W.: The Volcano Optimizer Generator: Extensibility and Efficient Seach. In *Proceedings of the 9th IEEE Internation Conference on Data Engineering*. IEEE Computer Society, 1993.

[GÖ03] Golab, L.; Özsu, M. T.: Issues in Data Stream Management. *SIGMOD Record*, Band 32, Nr. 2, S. 5–14, 2003.

[Gra95] Graefe, G.: The Cascades Framework for Query Optimization. *Data Engineering Bulletin*, Band 18, 1995.

[GS11] Gropengießer, F.; Sattler, K.-U.: Transactions a la carte - Implementation and Performance Evaluation of Transactional Support on Top of Amazon S3. In *IPDPS Workshops*, S. 1082–1091, 2011.

[GSD97] Geppert, A.; Scherrer, S.; Dittrich, K. R.: KIDS: Construction of Database Management Systems based on Reuse. Technischer Bericht, 1997.

[HFLP89] Haas, L. M.; Freytag, J. C.; Lohman, G. M.; Pirahesh, H.: Extensible query processing in Starburst. In Clifford, J.; Lindsay, B. G.; Maier, D. (Hrsg.): *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, S. 377–388. Portland, Oregon, 1989.

[Hir02] Hirschfeld, R.: AspectS – Aspect-Oriented Programming with Squeak. In *Proceedings of Net.ObjectDays (NODe)*, S. 219–235, 2002.

[HKM+05] Helmerich, A.; Koch, N.; Mandel, L.; Munich, F. G.; Braun, P.; Dornbusch, P.; Gruler, A.; Keil, P.; Leisibach, R.; Romberg, J.; Schätz, B.; Wild, T.; Wimmel, G.: Study of Worldwide Trends and R&D Programmes in Embedded Systems in View of Maximising the Impact of a Technology Platform in the Area. Technical Report, European Commission, Brussels, Belgium, 2005.

[HMTN+08] Hänninen, K.; Mäki-Turja, J.; Nolin, M.; Lindberg, M.; Lundbäck, J.; Lundbäck, K.-L.: The Rubus Component Model for Resource Constrained Real-Time Systems. In *SIES*, S. 177–183, 2008.

[HNP95] Hellerstein, J. M.; Naughton, J. F.; Pfeffer, A.: Generalized Search Trees for Database Systems. In Dayal, U.; Gray, P. M. D.; Nishio, S. (Hrsg.): *VLDB*, S. 562–573. Morgan Kaufmann, 1995.

[HR85] Härder, T.; Reuter, A.: Architektur von Datenbanksystemen für Non-Standard-Anwendungen. In *BTW*, S. 253–286, 1985.

[HRR98] Henzinger, M.; Raghavan, P.; Rajagopalan, S.: Computing on Data Streams. Technischer Bericht Nr. SRC-TN-1998-011, Hewlett Packard Laboratories, 1998.

[HS90] Haas, L., Chang, W., Lohman, G., McPherson, M., Wilms, P., Lapis, G., Lindsay, B., Pirahesh, H., Carey, M.; Shekita, E.: Starburst Mid-Flight: As The Dust Clears. *tkde*, Band 2, Nr. 1, S. 143–160, 1990.

[HZ87] Hornick, M. F.; Zdonik, S. B.: A Shared, Segmented Memory System for an Object-Oriented Database. *ACM Transactions on Information Systems*, Band 5, Nr. 1, S. 70–95, 1987.

[IEE00] IEEE: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. The Institute of Electrical and Electronics Engineers, Inc., 2000.

[IGE00] Intanagonwiwat, C.; Govindan, R.; Estrin, D.: Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *MOBICOM*, S. 56–67, 2000.

[IHS+96] Ishikawa, Y.; Hori, A.; Sato, M.; Matsuda, M.; Nolte, J.; Tezuka, H.; Konaka, H.; Maeda, M.; Kubota, K.: Design and Implementation of Metalevel Architecture in C++ – MPC++ Approach –. In *Reflection '96*, S. 141–154. 1996.

[JBZZ03] Jarzabek, S.; Bassett, P.; Zhang, H.; Zhang, W.: XVCL: XML-based Variant Configuration Language. In *ICSE*, S. 810–811, 2003.

[JF88] Johnson, R. E.; Foote, B.: Designing Reusable Classes. *Journal of Object-Oriented Programming*, Band 1, Nr. 2, S. 22–35, 1988.

[JOW+02] Juang, P.; Oki, H.; Wang, Y.; Martonosi, M.; Peh, L.-S.; Rubenstein, D.: Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. 2002.

[KA09] Kästner, C.; Apel, S.: Virtual Separation of Concerns – A Second Chance for Preprocessors. *Journal of Object Technology (JOT)*, Band 8, Nr. 6, S. 59–78, 2009.

[KAK08] Kästner, C.; Apel, S.; Kuhlemann, M.: Granularity in Software Product Lines. In *ICSE*, S. 311–320, 2008.

[Kan03] Kandé, M. M.: *A Concern-Oriented Approach to Software Archtitecure*. Dissertation, Technische Universität Berlin, 2003.

[Käs07a] Kästner, C.: Aspect-Oriented Refactoring of Berkeley DB. Diplomarbeit, Universität Magdeburg, Germany, 2007.

[Käs07b] Kästner, C.: CIDE: Decomposing Legacy Applications into Features. In *SPLC*, S. 149–150, 2007.

[Käs10] Kästner, C.: *Virtual Separation of Concerns: Toward Preprocessors 2.0*. Dissertation, Universität Magdeburg, 2010.

[KATS11] Kästner, C.; Apel, S.; Thüm, T.; Saake, G.: Type Checking Annotation-Based Product Lines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2011.

[KCH+90] Kang, K. C.; Cohen, S. G.; Hess, J. A.; Novak, W. E.; Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technischer Bericht Nr. CMU/SEI-90-TR-21 ESD-90-TR-222, Software Engineering Institute Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, USA, 1990.

[KD99] Kabra, N.; DeWitt, D. J.: OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization. *VLDB Journal*, Band 8, Nr. 1, S. 55–78, 1999.

[KGR+] Kästner, C.; Giarrusso, P. G.; Rendel, T.; Erdweg, S.; Ostermann, K.; Berger, T.: Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[KKHL10] Kenner, A.; Kästner, C.; Haase, S.; Leich, T.: TypeChef: Toward Type Checking #ifdef Variability in C. In *Proceedings of the Second Workshop on Feature-Oriented Software Development (FOSD)*, S. 25–32. ACM Press, New York, NY, USA, 2010.

[KKP99] Kahn, J. M.; Katz, R. H.; Pister, K. S. J.: Next century challenges: Mobile Networking for "Smart Dust". In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, S. 271–278. ACM, New York, NY, USA, 1999.

[KLM+97] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C. V.; Loingtier, J.-M.; Irwin, J.: Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, S. 220–242. Springer-Verlag LNCS 1241, Finland, 1997.

[KTS+09] Kästner, C.; Thüm, T.; Saake, G.; Feigenspan, J.; Leich, T.; Wielgorz, F.; Apel, S.: FeatureIDE: Tool Framework for Feature-Oriented Software Development. In *Proceedings of the 31th International Conference on Software Engineering (ICSE)*, S. 611–614. IEEE Computer Society, Los Alamitos, CA, 2009.

[Kun11] Kunze, S.: Untersuchung und Nutzung mehrdimensionaler Indexstrukturen in einem Datenbankmanagementsystem für tief eingebettete Systeme. Bachelorarbeit, Universität Magdeburg, Germany, 2011.

[LAL+10] Liebig, J.; Apel, S.; Lengauer, C.; Kästner, C.; Schulze, M.: An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, S. 105–114. ACM Press, New York, NY, USA, 2010.

[LALL09] Liebig, J.; Apel, S.; Lengauer, C.; Leich, T.: RobbyDBMS – A Case Study on Hardware/Software Product Line Engineering. In *Proceedings of the First Workshop on Feature-Oriented Software Development (FOSD)*, S. 60–65. ACM Press, 2009.

[LAMS05] Leich, T.; Apel, S.; Marnitz, L.; Saake, G.: Tool Support for Feature-Oriented Software Development - FeatureIDE: An Eclipse-Based Approach -. In *Proceedings of OOPSLA Eclipse Technology eXchange (ETX) Workshop*. San Diego, USA, 2005.

[LARS05] Leich, T.; Apel, S.; Rosenmüller, M.; Saake, G.: Handling Optional Features in Software Product Lines. In *Proceedings of OOPSLA Workshop on Managing Variabilities consistently in Design and Code*. San Diego, USA, 2005.

[LAS05] Leich, T.; Apel, S.; Saake, G.: Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In *Proceedings of the 9th East-European Conference on Advances in Databases and Information Systems (ADBIS'05)*, September 12-15. Lecture Notes in Computer Science (LNCS), Springer, 2005.

[LBL06] Liu, J.; Batory, D. S.; Lengauer, C.: Feature Oriented Refactoring of Legacy Applications. In *ICSE*, S. 112–121, 2006.

[LBN05] Liu, J.; Batory, D. S.; Nedunuri, S.: Modeling Interactions in Feature Oriented Software Designs. In *FIW*, S. 178–197, 2005.

[LC86] Lehman, T. J.; Carey, M. J.: A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, S. 294–303. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986.

[Lee08] Lee, E. A.: Cyber physical systems: Design challenges. Technischer Bericht Nr. UCB/EECS-2008-8, EECS Department, University of California, Berkeley, 2008.

[LHBC05] Lopez-Herrejon, R. E.; Batory, D.; Cook, W.: Evaluating Support for Features in Advanced Modularization Technologies. In *In ECOOP*, S. 169–194. Springer, 2005.

[LHBL06] Lopez-Herrejon, R. E.; Batory, D. S.; Lengauer, C.: A Disciplined Approach to Aspect Composition. In *PEPM*, S. 68–77, 2006.

[Lie08] Liebig, J.: Untersuchung der Anwendung erweiterter Programmierparadigmen für die Programmierung eingebetteter Systeme. Diplomarbeit, Universität Magdeburg, Germany, 2008.

[LK98] Lopes, C. V.; Kiczales, G.: Recent Developments in AspectJ. In *ECOOP'98 Workshop Reader (Aspect-Oriented Programming Workshop)*. Springer-Verlag LNCS 1543, 1998.

[LKA11] Liebig, J.; Kästner, C.; Apel, S.: Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD)*, S. 191–202. ACM Press, 2011.

[LKL02] Lee, K.; Kang, K. C.; Lee, J.: Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In Gacek, C. (Hrsg.): *Software Reuse: Methods, Techniques, and Tools: Proceedings of the Seventh Reuse Conference (ICSR7)*, S. 62–77, 2002.

[LLM99] Lieberherr, K.; Lorenz, D.; Mezini, M.: Programming with Aspectual Components. Technischer Bericht Nr. NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, 1999.

[LLO03] Lieberherr, K. J.; Lorenz, D. H.; Ovlinger, J.: Aspectual Collaborations: Combining Modules and Aspects. *Computer Journal*, Band 46, Nr. 5, S. 542–565, 2003.

[LNT00] Lu, H.; Ng, Y. Y.; Tian, Z.: T-Tree or B-Tree: Main Memory Database Index Structure Revisited. In *ADC '00: Proceedings of the Australasian Database Conference*, S. 65. IEEE Computer Society, Washington, DC, USA, 2000.

[LO03] Lieberherr, D. H. L. K.; Ovlinger, J.: Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, Band 46, Nr. 5, S. 542–565, 2003.

[LS88] Lynch, C. A.; Stonebraker, M.: Extended User-Defined Indexing with Application to Textual Databases. In Bancilhon, F.; DeWitt, D. J. (Hrsg.): *VLDB*, S. 306–317. Morgan Kaufmann, 1988.

[LWS+08] Lynch, J. P.; Wang, Y.; Sundararajan, A.; Law, K. H.; Kiremidjian, A. S.: Wireless Sensing for Structural Health Monitoring of Civil, 2008.

[Mar03] Marwedel, P.: *Embedded System Design*. Kluwer, 2003.

[Mar05] Marnitz, L.: Werkzeugunterstützung für die Merkmalorientierte Softwareentwicklung. Diplomarbeit, Universität Magdeburg, Germany, 2005.

[Mar06] Martonosi, M.: Embedded Systems in the Wild: ZebraNet Software, Hardware, and Deployment Experiences. *ACM SIGPLAN Notices*, Band 41, Nr. 7, S. 1–1, 2006.

[Mar07] Marwedel, P.: *Eingebettete Systeme*. Springer, 2007.

[MBC09] Mendonça, M.; Branco, M.; Cowan, D. D.: S.P.L.O.T.: Software Product Lines Online Tools. In *OOPSLA Companion*, S. 761–762, 2009.

[MBHT96] McKenna, W. J.; Burger, L.; Hoang, C.; Truong, M.: EROC: A Toolkit for Building NEATO Query Optimizers. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, S. 111–121. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[McI68] McIlroy, M. D.: Mass-Produced Software Components. In Naur, P.; Randell, B. (Hrsg.): *Software Engineering Concepts and Techniques*, S. 88–98. North Atlantic Treaty Organisation (NATO) Conference on SoftwareEngineering, Garmisch-Partenkirchen, 1968.

[MCP+02] Mainwaring, A.; Culler, D.; Polastre, J.; Szewczyk, R.; Anderson, J.: Wireless Sensor Networks for Habitat Monitoring. In *WSNA '02: Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, S. 88–97. ACM, New York, NY, USA, 2002.

[MFH01] McDirmid, S.; Flatt, M.; Hsieh, W. C.: Jiazzi: New-Age Components for Old-Fashioned Java. *ACM SIGPLAN Notices*, Band 36, Nr. 11, S. 211–222, 2001.

[MFHH02] Madden, S.; Franklin, M. J.; Hellerstein, J. M.; Hong, W.: TAG: A Tiny Aggregation Service for Ad-Hoc Sensor Networks. In *OSDI*, 2002.

[MFHH03] Madden, S. R.; Franklin, M. J.; Hellerstein, J. M.; Hong, W.: The Design of an Acquisitional Query Processor for Sensor Networks. In *Proceedings of SIGMOD'03*, 2003.

[MFHH05a] Madden, S. R.; Franklin, M. J.; Hellerstein, J. M.; Hong, W.: TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transactions Database Systems. (TODS)*, Band 30, Nr. 1, S. 122–173, 2005.

[MFHH05b] Madden, S. R.; Franklin, M. J.; Hellerstein, J. M.; Hong, W.: Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, Band 30, Nr. 1, S. 122–173, 2005.

[MH02] Madden, S.; Hellerstein, J. M.: Distributing Queries over Ow-Power Wireless Sensor Networks. In Franklin, M. J.; Moon, B.; Ailamaki, A. (Hrsg.): *SIGMOD Conference*, S. 622. ACM, 2002.

[ML98] Mezini, M.; Lieberherr, K.: Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proceedings of the 13th ACM SIGPLAN*, OOPSLA, S. 97–116. ACM, New York, NY, USA, 1998.

[MO04a] Mezini, M.; Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, SIGSOFT/FSE, S. 127–136. ACM, 2004.

[MO04b] Mezini, M.; Ostermann, K.: Variability Management with FeatureOriented Programming and Aspects. *ACM SIGSOFT*, 2004.

[MP87] McPherson, J.; Pirahesh, H.: An Overview of Extensibility in Starburst. *CS Res. Rep.*, Nr. RJ 5599 (56909), 1987.

[MSvdMW04] Müller-Schloer, C.; Malsburg, C. v. d.; Würtz, R. P.: Organic Computing. *Informatik Spektrum*, Band 27, Nr. 4, S. 332–336, 2004.

[ND96] Nittel, S.; Dittrich, K. R.: A Storage Server for the Efficient Support of Complex Objects. In *POS*, S. 205–221, 1996.

[Nei80] Neighbors, J. M.: *Software Construction Using Components*. Dissertation, Department of Information and Computer Science, University of California, 1980.

[NNT+04] Nystrom, D.; Nolin, M.; Tešanovic, A.; Norstrom, C.; Hansson, J.: Pessimistic Concurrency Control and Versioning to Support Database Pointers in Real-Time Databases. *ECRTS*, S. 261–270, 2004.

[NTN+02] Nyström, D.; Tešanovic, A.; Norström, C.; Hansson, J.; Bånkestad, N.-E.: Data Management Issues in Vehicle Control Systems: A Case Study . In *ECRTS*, S. 249–256. IEEE Computer Society, 2002.

[NTN+04] Nyström, D.; Tešanovic, A.; Nolin, M.; Norström, C.; Hansson, J.: COMET: A Component-Based Real-Time Database for Automotive Systems.

In *Proceedings of the Workshop on Software Engineering for Automotive Systems at 26th International Conference on Software engineering (ICSE'04)*. IEEE Computer Society Press, Edinburgh, Scotland, 2004.

[NTNH03] Nyström, D.; Tešanovic, A.; Norström, C.; Hansson, J.: Database Pointers: A Predictable Way of Manipulating Hot Data in Hard Real-Time Systems. In Chen, J.; Hong, S. (Hrsg.): *RTCSA*, Lecture Notes in Computer Science, Band 2968, S. 454–465. Springer, 2003.

[Ols00] Olson, M. A.: Selecting and Implementing an Embedded Database System. *IEEE Computer*, Band 33, Nr. 9, S. 27–34, 2000.

[Ost01] Ostermann, K.: Implementing Reusable Collaborations with Delegation Layers. In *Proceedings OOPSLA'01*, 2001.

[Ost03] Ostermann, K.: *Modules for Hierarchical and Crosscutting Models*. Dissertation, TU Darmstadt, Darmstadt, 2003.

[OT00] Ossher, H.; Tarr, P.: Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In *In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2000.

[Par76] Parnas, D. L.: On the Design and Development of Program Families. *IEEE Transactions On Software Engineering*, Band 1, S. 1–9, 1976.

[Par78] Parnas, D. L.: Designing Software for Ease of Extension and Contraction. In *3rd international conference on Software engineering*, S. 264 – 277, 1978.

[Par79] Parnas, D. L.: Designing Software for Ease of Extension and Contraction. *Transaction on Software Engineering*, Band IEEE: SE-5, Nr. 2, 1979.

[PB02] Post, W.; Bauer, H.: *Konventionelle und elektronische Bremssysteme: Physikalische Grundlagen des Bremsens. Komponenten und Systeme. Antiblockiersystem ABS. Elektrohydraulische Bremse SBC*. Robert Bosch GmbH Stuttgart and Abteilung Produktmarketing Diagnostics & Test Equipment and Unternehmensbereich Automotive Aftermarket, 2002.

[PBL05] Pohl, K.; Böckle, G.; Linden, F. J. v. d.: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[PBS93] Price, B.; Baecker, R.; Smal, I.: A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, Band 4, Nr. 3, S. 211–266, 1993.

[PBVB01] Pucheral, P.; Bouganim, L.; Valduriez, P.; Bobineau, C.: PicoDBMS: Scaling Down Database Techniques for the Smartcard. *VLDB Journal*, Band 10, Nr. 2–3, 2001.

[PLKR07] Pukall, M.; Leich, T.; Kuhlemann, M.; Rosenmüller, M.: Highly Configurable Transaction Management for Embedded Systems. In *AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, S. 1–8, 2007.

[Pre97] Prehofer, C.: Feature Oriented Programming. A Fresh Look at Objects. In Aksit, M.; Matsuoka, S. (Hrsg.): *ECOOP '97 - Object-Oriented Programming*, S. 419–443. Springer-Verlag, 1997.

[PRSF00] Prassler, E.; Ritter, A.; Schaeffer, C.; Fiorini, P.: A Short History of Cleaning Robots. *Auton. Robots*, Band 9, Nr. 3, S. 211–226, 2000.

[PSS87] Paul, H. B.; Schek, H. J.; Scholl, M. H.: Architecture and Implementation of the Darmstadt Database Kernel System. *SIGMOD Rec.*, Band 16, Nr. 3, S. 196–207, 1987.

[Puk06] Pukall, M.: FAME-DBMS: Entwurf ausgewählter Aspekte der Transaktionsverwaltung. Diplomarbeit, Universität Magdeburg, 2006.

[RALS09] Rosenmüller, M.; Apel, S.; Leich, T.; Saake, G.: Tailor-made Data Management for Embedded Systems: A Case Study on Berkeley DB. *Data and Knowledge Engineering (DKE)*, Band 68, Nr. 12, S. 1493–1512, 2009.

[RCDS87] Richardson, J. E.; Carey, M. J.; DeWitt, D. J.; Schuh, D. T.: Persistence in EXODUS. In *Proc. A Workshop on Persistent Object Sys.: Their Design, Implementation, and Use*. Appin, Scotland, 1987.

[RCS93] Richardson, J. E.; Carey, M. J.; Schuh, D. T.: The Design of the E Programming Language. *ACM Transactions on Programming Languages and Systems*, Band 15, S. 494–534, 1993.

[Rei06] Reif, K.: *Automobilelektronik - Eine Einführung für Ingenieure*. Vieweg + Teubner, 2006.

[RKF98] Rossetti, M. D.; Kumar, A.; Felder, R. A.: Mobile Robot Simulation of Clinical Laboratory Deliveries. In *Proceedings of the 30th conference on Winter simulation*, WSC '98, S. 1415–1422. IEEE Computer Society Press, Los Alamitos, CA, USA, 1998.

[RKS+09] Rosenmüller, M.; Kästner, C.; Siegmund, N.; Sunkle, S.; Apel, S.; Leich, T.; Saake, G.: SQL à la Carte – Toward Tailor-made Data Management. In *Datenbanksysteme in Business, Technologie und Web (BTW)*, S. 117–136, 2009.

[RLA07] Rosenmüller, M.; Leich, T.; Apel, S.: Konfigurierbarkeit für ressourceneffiziente Datenhaltung in eingebetteten Systemen am Beispiel von Berkeley DB. In *Datenbanksysteme in Business Technologie und Web - Workshop*, S. 329–341. Verlag Mainz, 2007.

[RLAS07] Rosenmüller, M.; Leich, T.; Apel, S.; Saake, G.: Von Mini- über Micro- bis zu Nano-DBMS: Datenhaltung in eingebetteten Systemen. *Datenbank Spektrum*, Band 7, Nr. 20, S. 33–43, 2007.

[Ros05] Rosenmüller, M.: Merkmalsorientierte Programmierung in C++. Diplomarbeit, Universität Magdeburg, Germany, 2005.

[Ros11] Rosenmüller, M.: *Towards Flexible Feature Composition: Static and Dynamic Binding in Software Product Lines*. Dissertation, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 2011.

[RSS+08] Rosenmüller, M.; Siegmund, N.; Schirmeier, H.; Sincero, J.; Apel, S.; Leich, T.; Spinczyk, O.; Saake, G.: FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems. In *EDBT'08 Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, S. 1–6. Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 2008.

[RV96] Rozenberg, G.; Vaandrager, F.: *Lectures on Embedded Systems: European Educational Forum School on Embedded*. Springer, 1996.

[RWL95] Reenskaug, T.; Wold, P.; Lehne, O. A.: *Working with Objects: The OORam Software Engineering Method*. Prentice-Hall, 1995.

[SAAF73] Senko, M. E.; Altman, E. B.; Astrahan, M. M.; Fehder, P. L.: Data Structures and Accessing in Data Base Systems. *IBM Systems Journal*, Band 12, Nr. 1, 1973.

[SB00] Smaragdakis, Y.; Batory, D. S.: Mixin-Based Programming in C++. In *GCSE*, S. 163–177, 2000.

[SB93] Singhal, V.; Batory, D. S.: P++: A Language for Software System Generators. Technischer Bericht, Austin, TX, USA, 1993.

[SB98] Smaragdakis, Y.; Batory, D.: Implementing Layered Designs with Mixin Layers. In Jul, E. (Hrsg.): *Proceedings of ECOOP'98, LNCS 1445*, S. 550–570. Springer-Verlag Heidelberg Berlin, 1998.

[SBA+13] Siegmund, J.; Brechmann, A.; Apel, S.; Kästner, C.; Liebig, J.; Leich, T.; Saake, G.: Toward Measuring Program Comprehension with Functional Magnetic Resonance Imaging. In *International Symposium on the Foundations of Software Engineering*, 2013. New ideas paper. Submitted 29.6.2012, Accepted 14.8.2012.

[SBB+10] Schaefer, I.; Bettini, L.; Bono, V.; Damiani, F.; Tanzarella, N.: Delta-Oriented Programming of Software Product Lines. In *SPLC*, S. 77–91, 2010.

[Sc05] Stonebraker, M.; Çetintemel, U.: One Size Fits All: An Idea Whose Time Has Come and Gone (Abstract). In *ICDE*, S. 2–11, 2005.

[SC92] Spencer, H.; Collyer, G.: #ifdef Considered Harmful, or Portability Experience with C News. In *Proceedings of the Summer USENIX Technical Conference*, S. 185–197. USENIX Association, 1992.

[Sch05] Scholz, P.: *Softwareentwicklung eingebetteter Systeme: Grundlagen, Modellierung, Qualitätssicherung*. Springer, 2005.

[Sch12] Schröter, I.: Modulare Integration von konfigurierbaren Transaktionsmechanismen in Cloud-basierte Datenspeichersysteme. Diplomarbeit, Universität Magdeburg, Germany, 2012.

[Sel07] Seltzer, M. I.: Berkeley DB: A Retrospective. *IEEE Data Eng. Bull.*, Band 30, Nr. 3, S. 21–28, 2007.

[SGSP02] Spinczyk, O.; Gal, A.; Schröder-Preikschat, W.: AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*. IEEE Computer Society, Sydney, Australia, 2002.

[SKW92] Singhal, V.; Kakkad, S. V.; Wilson, P. R.: Texas: An Efficient, Portable Persistent Store. In *POS*, S. 11–33, 1992.

[SMI+08] Schulze, M.; Mäkinen, T.; Irion, J.; Flament, M.; Kessel, T.: Preventive and Active Safety Applications Contribute to the Road Safety Goals on European Roads (PREVENT) IP D15: Final Report . http://www.prevent-ip.org/download/deliverables/ip_level/, EU - Report Number: PR-04000-IPD-080222-v16, 2008.

[SMSR02] Stanley M. Sutton, J.; Rouvellou, I.: Modeling of Software Concerns in Cosmos. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, S. 127 – 133, 2002.

[SPSW90] Schek, H.-J.; Paul, H.-B.; Scholl, M. H.; Weikum, G.: The DASDBS project: Objectes, experiences, and future projects. *IEEE Transactions on Knowledge and Data Engineering*, Band 2, Nr. 1, 1990.

[SR03] Sutton, S. M.; Rouvellou, I.: Applicability of Categorization Theory to Multidimensional Separation of Concerns. Technischer Bericht, College of Computer and Information Science, Northeastern University, 360 Huntington Avenue, Boston, USA, 2003.

[SRK+08] Siegmund, N.; Rosenmüller, M.; Kuhlemann, M.; Kästner, C.; Saake, G.: Measuring Non-Functional Properties in Software Product Line for Product Derivation. In *APSEC*, S. 187–194, 2008.

[SRK+12] Siegmund, N.; Rosenmüller, M.; Kuhlemann, M.; Kästner, C.; Apel, S.; Saake, G.: SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines. *Software Quality Journal*, Band 20, Nr. 3, S. 487–517, 2012.

[SRS+09] Saake, G.; Rosenmüller, M.; Siegmund, N.; Kästner, C.; Leich, T.: Downsizing Data Management for Embedded Systems. *Egyptian Computer Science Journal (ECS)*, Band 31, Nr. 1, S. 1–13, 2009.

[SSH11] Saake, G.; Sattler, K.-U.; Heuer, A.: *Datenbanken — Implementierungstechniken*. MITP-Verlag, Bonn, 3. Auflage, 2011.

[SSR+12] Soffner, M.; Siegmund, N.; Rosenmüller, M.; Feigenspan, J.; Leich, T.; Saake, G.: A Variability Model for Query Optimizers. In *International Baltic Conference on Databases and Information Systems*. IOS Press, 2012.

[Ste06] Steimann, F.: The Paradoxical Success of Aspect-Oriented Programming. In *OOPSLA*, S. 481–497, 2006.

[SvGB01] Svahnberg, M.; Gurp, J. v.; Bosch, J.: On the Notion of Variability in Software Product Lines. In *Proceedings Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*, 2001.

[SvGB05] Svahnberg, M.; Gurp, J. v.; Bosch, J.: A Taxonomy of Variability Realization Techniques: Research Articles. *Softw. Pract. Exper.*, Band 35, Nr. 8, S. 705–754, 2005.

[SZ06] Schäuffele, J.; Zurawka, T.: *Automotive Software-Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen*. Friedrich Vieweg & Sohn Verlag, 2006.

[SZR91] Skarra, A. H.; Zdonik, S. B.; Reiss, S. P.: ObServer: An Object Server for an Object-Oriented Database System. In *On Object-Oriented Database System*, S. 275–290. 1991.

[Szy98] Szyperski, C.: *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[Ten00] Tennenhouse, D.: Proactive Computing. *Communications of the ACM*, Band 43, Nr. 5, 2000.

[TKB+12] Thüm, T.; Kästner, C.; Benduhn, F.; Meinicke, J.; Saake, G.; Leich, T.: FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 2012.

[TM03] Tubaishat, M.; Madria, S.: Sensor Networks: An Overview. *Potentials, IEEE*, Band 22, Nr. 2, S. 20–23, 2003.

[TNH+03] Tešanovic, A.; Nyström, D.; Hansson, J.; Norström, C.; Uhlin, P.: Aspect-Level WCET Analyzer: A Tool for Automated WCET Analysis of a Real-Time Software Composed Using Aspects and Components. In *3rd International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*, 2003.

[TNHN03] Tešanovic, A.; Nyström, D.; Hansson, J.; Norström, C.: Towards Aspectual Component-Based Development of Real-Time Systems. In *RTCSA*, S. 558–577, 2003.

[TO00] Tarr, P.; Ossher, H.: Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In Kluwer (Hrsg.): *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2000.

[TOH+99] Tarr, P.; Ossher, H.; Harrison, W.; Sutton, S. M.; Jr.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE*, S. 107–119, 1999.

[TSH04] Tešanović, A.; Sheng, K.; Hansson, J.: Application-Tailored Database Systems: A Case of Aspects in an Embedded Database. In *Proceedings of the 8th International Database Engineering and Applications Symposium (IDEAS'04)*. IEEE Computer Society Press, Coimbra, Portugal, 2004.

[TSP+11] Thüm, T.; Schulze, S.; Pukall, M.; Saake, G.; Günther, S.: AutoDaMa: On the Feasibility of Secure and Tailor-made Automotive Data Management. Technischer Bericht Nr. FIN-008-2011, School of Computer Science, University of Magdeburg, 2011.

[TSP+12] Thüm, T.; Schulze, S.; Pukall, M.; Saake, G.; Günther, S.: Secure and Customizeable Data Management for Automotive Systems - A Feasibility Study. *ISRN Software Engineering*, Band 2012, S. 1–7, 2012.

[Tur02] Turley, J.: The Two Percent Solution. *Embedded Systems Programming*, 2002.

[ULS10] Urban, M.; Lohmann, D.; Spinczyk, O.: The Aspect-Oriented Design of the PUMA C/C++ Parser Framework. In *AOSD*, S. 217–221, 2010.

[uR11] Rahman, S. S. u.: *Cellular DBMS: Customizable and autonomous data management using a RISC-style architecture*. Dissertation, University of Magdeburg, Germany, 2011.

[Van97] VanHilst, M.: Role Oriented Programming for Software Evolution. Ph. D. Thesis, University of Washington, 1997.

[VN96] VanHilst, M.; Notkin, D.: Using Role Components to Implement Collaboration-Based Designs. In *OOPSLA '96 CA, USA*. ACM Press, 1996.

[Wei91] Weikum, G.: Principles and Realization Strategies of Multilevel Transaction Management. *ACM Trans. Database Syst.*, Band 16, Nr. 1, S. 132–180, 1991.

[Wei93] Weiser, M.: Hot Topics: Ubiquitous Computing. *IEEE Computer*, Band 26, Nr. 10, 1993.

[Wir71] Wirth, N.: Program Development by Stepwise Development. *Communications of the ACM*, Band 14, Nr. 4, S. 221–227, 1971.

[WKD04] Wand, M.; Kiczales, G.; Dutchyn, C.: A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. *ACM Trans. Program. Lang. Syst.*, Band 26, Nr. 5, S. 890–910, 2004.

[WLLP01] Warneke, B.; Last, M.; Liebowitz, B.; Pister, K. S. J.: Smart Dust: Communicating with a Cubic-Millimeter Computer. *Computer*, Band 34, Nr. 1, S. 44–51, 2001.

[WM00] Willink, E. D.; Muchnick, V. B.: Object-Oriented Preprocessor fit for C++. *IEE Proceedings - Software*, Band 147, Nr. 2, 2000.

[WMG04] Woo, A.; Madden, S.; Govindan, R.: Networking support for Query Processing in Sensor Networks. *Commun. ACM*, Band 47, Nr. 6, S. 47–52, 2004.

[YG02] Yao, Y.; Gehrke, J.: The Cougar Approach to In-Network Query Processing in Sensor Networks. *SIGMOD Rec.*, Band 31, Nr. 3, S. 9–18, 2002.

[YMG08] Yick, J.; Mukherjee, B.; Ghosal, D.: Wireless Sensor Network Survey. *Comput. Netw.*, Band 52, Nr. 12, S. 2292–2330, 2008.

[YSL91] Yaseen, R.; Su, S. Y. W.; Lam, H.: An Extensible Kernel Object Management System. In *OOPSLA*, S. 247–263. ACM, New York, NY, USA, 1991.