JAVADAPTOR: Unrestricted Dynamic Updates of Java Applications

Dissertation
zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

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

von Dipl.-Inform. Mario Pukall
geb. am 4. September 1978 in Salzwedel

Gutachter:
Prof. Dr. Gunter Saake
Prof. Dr. Walter Cazzola
Prof. Dr. Uwe Aßmann

Ort und Datum des Promotionskolloquiums Magdeburg, den 22.03.2012

Pukall, Mario:
JavAdaptor: Unrestricted Dynamic Updates of Java Applications
Dissertation, Otto-von-Guericke-Universität
Magdeburg, Germany, 2012.

Abstract

Software is changed frequently during its life cycle. New requirements come and bugs must be fixed. To update a deployed application, it usually must be stopped, patched, and restarted. This update strategy causes different problems. For instance, frequent program restarts to test newly added functionality or debug applications consume time and thus render software development processes ineffective. Another issue is that the application and its provided services are unavailable during the update. This conflicts with highly available applications, because downtimes of those applications are usually costly.

Because of the described problems, we aim at another update strategy – namely dynamic software updates. The idea of dynamic software updates is to update applications during their execution, which prevents time consuming restarts and costly service downtimes.

There is a large body of research on dynamic software updates, but so far, existing approaches are too restrictive. Some of them are inflexible (i.e., they do not support all updates that are possible with program updates based on restarts), whereas others cause significant performance penalties, require specific runtime environments, dictate the program architecture, or offer coarse-grained updates only. With the arrival of virtual machines, which abstract the runtime environment from the operating system and thus offer new starting points for the development of DSU approaches, the situation slightly relaxed and less restrictive approaches came up. However, unrestricted dynamic software updates remain to be provided only by dynamically typed languages (which are less performant than statically typed languages), whereas DSU approaches for statically typed languages are still restrictive.

With this work, we present JAVADAPTOR, the first dynamic software update approach based on statically typed Java, which is highly flexible, performant, platform independent, architecture independent, and applies updates at a fine level of granularity. Conceptually, JAVADAPTOR combines schema changing class reloadings with caller updates through Java HotSwap, containers, and proxies. We detail the concepts and implementation of JAVADAPTOR. In addition, we demonstrate the practicability of JAVADAPTOR with different non-trivial case studies. We further evaluate whether JAVADAPTOR fulfills the above mentioned criteria. That is, we analyze whether JAVADAPTOR hits the targeted high level of update flexibility, is performant, does not cause platform or architecture dependencies, and performs updates at a fine granularity level.

Zusammenfassung

Computerprogramme unterliegen während ihres Lebenszyklus ständigen Änderungen. Häufig müssen neue Anforderungen erfüllt oder Fehler beseitigt werden. Zum Ändern des Programms, muss dieses gestoppt, aktualisiert und neu gestartet werden. Aus dieser Art der Programmaktualisierung ergeben sich vielfältige Probleme. Auf der einen Seite senken zeitaufwendige Neustarts, zum Testen hinzugefügter Funktionen oder zur Fehlerbeseitigung, die Effizienz des Entwicklungsprozesses. Auf der anderen Seite beeinträchtigt diese Form der Aktualisierung die Verfügbarkeit des Programms und der durch das Programm bereitgestellten Dienste, was insbesondere in Bezug auf hochverfügbare Systeme problematisch ist, da deren Ausfall oft hohe Kosten verursacht.

Aufgrund der genannten Probleme, zielt die vorliegende Arbeit auf Lösungsansätze ab, die keinen Programmneustart erfordern und die Verfügbarkeit nicht einschränken. Genauer beschäftigt sich die Arbeit mit Lösungen, die die Aktualisierung von Programmen zur Laufzeit ermöglichen.

Wenngleich zu diesem Thema bereits viele verschiedene Lösungen existieren, so sind diese zumeist mehr oder weniger starken Einschränkungen unterlegen. Einige der Lösungen sind nicht flexibel genug, um alle Programmänderungen, die mittels Neustart möglich sind, zu bewerkstelligen. Andere Ansätze wiederum verlangsamen zumeist deutlich die Programmausführung, verursachen Plattformabhängigkeiten, sind nicht zu allen Programmarchitekturen kompatibel oder erfordern den (ineffizienten) Austausch großer Programmteile. Mit der Einführung virtueller Maschinen änderten sich die Vorraussetzungen und weniger eingeschränkte Lösungen kamen auf. Nichtsdestotrotz bieten bisher nur dynamische Programmiersprachen die Möglichkeit zur uneingeschränkten Laufzeitaktualisierung von Programmen (wobei dynamische Sprachen im Vergleich zu statisch getypten Sprachen eine geringere Performanz aufweisen). Statisch getypte Sprachen unterliegen diesbezüglich weiterhin nicht zu unterschätzenden Einschränkungen.

In dieser Arbeit wird mit JAVADAPTOR ein Ansatz präsentiert, der die statisch getypte Programmiersprache Java um die Möglichkeit erweitert, Programme zur Laufzeit und ohne die genannten Einschränkungen zu aktualisieren. Das vorgestellte Konzept kombiniert das schemaverändernde Nachladen von Klassen mit Referenzaktualisierungen auf der Basis von Java HotSwap, Containern und Proxies. Zentrale Beiträge der Arbeit sind detaillierte Beschreibungen der Konzepte und deren Implementierung, sowie der Nachweis der Praxistauglichkeit der Lösung anhand verschiedener Fallstudien. Weiterhin wird untersucht, ob die präsentierte Lösung Einschränkungen bezüglich Flexibilität, Performanz, Plattform, Programmarchitektur oder Änderungsgranularität unterliegt.

Acknowledgement

Doing good research and finally sum up the research results within a doctoral thesis means a lot of hard work. You have to solve many issues, which is often fun but sometimes pain. During my journey to this thesis, I got a lot of help from many different people.

First of all, I would like to thank my family for their support. Especially, I thank my beloved wife Ivonne and our beloved son Marek. Your love gave me the power to pursue my big goal. I further thank my parents and my sister for believing in me and supporting me and my little family when times were tough.

Moreover, I thank my best friends Michael Soffner, Lothar Schlesier, Stephan Vornholt, Karin Ranft, Jan Erdtmann, and Alexander Seitz. Laughing with you always cheered me up and helped me to relax and recover.

Next, I would like to thank Gunter Saake, who was always in when advice was needed or when my position was at risk.

I also owe a lot to Walter Cazzola, who partly founded me and gave me valuable advice as well. Thank you for your support.

The here presented work consists of many different partial solutions which, put together, result in a dynamic software update approach with high practical and scientific relevance. The resulting tool, i.e., JavAdaptor, would have been not as half as good without the help of my students Alexander Grebhahn and Reimar Schröter. Thank you for intensive discussions and for helping me with the implementation of JavAdaptor.

Furthermore, I thank Christian Kästner, who gave me a lot of invaluable feedback on my research and helped to improve not only my papers, but also this thesis.

Finally, I thank all my colleagues for their support/advice and for treating me as one of theirs. I simply could not ask for better colleagues.

Contents

List of Figures

List of Tables

List of Abbreviations

DSU	Dynamic Software Updates
JVM	Java Virtual Machine
OS	Operating System
JPDA	Java Platform Debugger Architecture
JVMTI	Java Virtual Machine Tool Interface
JDWP	Java Debug Wire Protocol
JDI	Java Debug Interface
JIT	just-in-time

1 Introduction

When thinking about software development and software maintenance there is just one thing which is predictable – alteration. Once a program goes live and works in productive mode its development is not completed. Well-known reasons for program updates are new requirements and incorrect program code. To update a program, it is usually necessary to *stop the program*, *apply a patch* and to *start it again*. Unfortunately, this software update strategy causes different problems.

A major problem is, that frequent program restarts to test newly added functionality and to debug applications consume time and thus render software development processes ineffective. However, applying updates through application restarts is not only a pain regarding software development, but also regarding software execution. What is characteristically for the described update strategy is that the application and its provided services are *unavailable* during the update. The problem is that unavailability conflicts with applications which must be highly available, e.g., security applications, web applications, and banking systems. Downtimes of those applications are usually costly. Furthermore, updates through restarts cause problems regarding end-user desktop applications, because end-users prefer update approaches that do not interrupt their tasks.

One may think that updating a program the described way (i.e., stopping, updating, restarting) is no problem during software development or software execution, because updates could be prepared beforehand and program restarts take not more than few seconds. This may be true for simple applications but it is not for complex systems. Complex systems often need a considerable startup, time until they run with desired performance, or time until they are at the point of execution where they were when shut down, e.g., because caches first need to be filled or program state must be recovered. Particularly, highly available applications are well-known to be stateful and to generate a lot of data during execution. That is, possibilities are high that updating complex applications in the described way results in time consuming restarts, which render development processes ineffective and cause problematic time periods of unavailability.

A strategy to face the unavailability problem is to postpone the update until it is appropriate to take the service provided by the application offline (e.g., at night, on weekend). Even if this strategy might be acceptable in some cases it may be not in others. In terms of highly available applications which provide their services globally (24 hours), nightly updates are no longer reasonable. When it comes to services which have to be available the whole week (e.g., market platforms) it is no longer reasonable to postpone the update until weekend. However, even in cases where update postponing is possible, problems arise. For instance bugs, e.g., relevant to security, which must be fixed immediately conflict the postponing strategy.

Another reasonable approach to face the unavailability problem is to use backup systems. That is, multiple instances of one and the same application are executed at the same time. In order to update the application, one instance is stopped and updated while another instance still provides the application's service. Then the updated application is restarted and continues to provide the service. Even if common practice in industry, this strategy does not come without issues. On the one hand, certain time is required to synchronize data and state between old and updated program instances to produce valid results when the updated application instance continues to provide the service. On the other hand, the strategy increases the requirements to be met in order to execute the application (e.g., in terms of memory consumption, cpu power, network bandwidth, power consumption, etc.).

However, update postponing and backup systems might help to decrease service downtimes under certain conditions. But, these strategies do not prevent developers from the burden of time consuming restarts during development to check the correctness of the recent program changes.

Because of the problems described above, we aim at a different program update strategy – namely *Dynamic Software Updates (DSU)*. The idea of dynamic software updates is to update programs during their execution, which prevents us from time consuming restarts and costly service downtimes.

1.1 Exploratory Focus

Research in the field of dynamic software updates (DSU) has a long tradition and a lot of approaches and solutions have been proposed over the years. Nevertheless, DSU is still an active field of research, because most of the existing DSU approaches are too restrictive. Some of them are *inflexible* (i.e., they do not support all updates that are possible when statically changing the program code) whereas others *require specific runtime environments*, *cause significant performance penalties*, or *dictate the program architecture*. With the arrival of virtual machines, which abstract the runtime environment from the *Operating System (OS)* and thus offer new starting points for DSU approach development, the situation slightly relaxed and less restrictive DSU approaches came up. However, unrestricted DSU remained to be provided only by dynamic languages like Smalltalk, Python, or Ruby (with a typical associated performance loss compared to statically typed languages [FG11]). By contrast, DSU support for statically typed languages is still restrictive. But, there is a growing class of complex (potentially highly available) applications which are written in statically typed languages. Thus, with this work, we address the shortcomings of statically typed languages regarding unrestricted dynamic software updates. In particular, we address *Java* for several reasons. First, it is frequently used to implement complex (potentially highly available) applications. Examples are *Apache Tomcat*,[1] *Java DB*,[2] *JBoss Application Server*,[3] and *HyperSQL*.[4] Second, Java is the most popular programming language according the *TIOBE Programming Community Index*,[5] which allows us to assume that the number of Java applications will further grow in the future. Third, with place number three in Oracle's current request for enhancement (RFE) list,[6] unrestricted DSU is one of the top-most requested features for Java.

1.2 Goals

Our goal is to provide the state-of-the-art regarding dynamic software updates for statically typed programming language Java. In particular, we aim to overcome the restrictions of current DSU approaches. We further aim to demonstrate that our solutions to overcome the restrictions do not conflict each other and can be integrated coexistent in one solution. In a nutshell our goal is:

> *To design, implement, and evaluate an approach for dynamic software updates in Java that is (1)* highly flexible*, (2)* performant*, (3)* platform independent*, (4)* architecture independent*, and (5)* fine-grained*, at the same time.*

After outlining the big picture of this work, we give an explanation of the stated goal and its inherent aspects.

1.2.1 Flexibility

We regard a DSU approach as flexible, if it allows the developer to change the running program in an unanticipated way, i.e., if it permits the application of functions for which the program was not prepared. With respect to Java, the level of flexibility offered by a DSU approach can be determined by answering the following three questions:

1. Can already loaded (executed) classes/interfaces be changed (note that changing not yet loaded classes/interfaces is no problem in Java)?

[1] `http://tomcat.apache.org/index.html`
[2] `http://www.oracle.com/technetwork/java/javadb/overview/index.html`
[3] `http://www.jboss.org/jbossas/`
[4] `http://hsqldb.org/`
[5] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`
[6] `http://bugs.sun.com/top25_rfes.do`

	Construct to be changed	**Related Elements**
Classes	(1) Class Declaration	Modifiers, Generic, Inner Classes, Superclass, Subclasses, Superinterfaces, Class Body, Member Declarations
	(2) Class Members	Fields, Methods
	(3) Field Declarations	Modifiers, Field Initialization, Field Type
	(4) Method Declarations	Modifiers, Signature (Name, Parameters), Return Type, Throws, Method Body
	(5) Constructor Declarations	Modifiers, Signature (Name, Parameter), Throws, Constructor Body
	(6) Blocks	Statements
	(7) Enums	Enum Declaration, Enum Body
Interfaces	(8) Interface Declaration	Modifiers, Generic, Superinterfaces, Subinterfaces, Interface Body, Member Declarations
	(9) Interface Members	Fields, Method Declarations
	(10) Field (Constant) Declarations	Field Initialization, Field Type
	(11) Abstract Method Declarations	Signature (Name, Parameters), Return Type, Throws
	(12) Blocks	Statements
	(13) Annotations	Annotation Type, Annotation Element

Table 1.1: Language constructs of Java 1.6 [GJSB05].

2. Can classes/interfaces be changed at runtime to the same extent as possible when changed statically (i.e., are changes to all elements listed in Table 1.1 possible)?

3. Is the program state kept beyond the update?

DSU approaches for which the aforementioned questions can be answered with *yes* can be considered as *highly flexible*.

We aim at highly flexible DSU approaches for different reasons. First, we believe that it is impossible to prepare an application for all potential upcoming requirements before program start, thus, unanticipated program changes must be supported. Second, only offering modifications of not previously executed program parts while disregarding the executed parts (e.g., already loaded classes/interfaces) restricts the application of unanticipated program changes. Third, we think that DSU approaches must support all changes that are possible with static software development in order to cover all update scenarios occurring at program runtime. Fourthly, in our opinion, dynamic software updates are only valuable if they do not cause program state losses. Consequently, DSU approaches which do not fulfill the mentioned requirements fail in scenarios they do not cover, i.e., they will cause program stops.

1.2.2 Performance

What comes to mind first is that performance corresponds to the speed of program execution. However, in the context of dynamic software updates it additionally corresponds to the time required to perform a runtime update. That is, we consider a DSU approach as *performant* if:

1. it does not introduce significant program execution overhead.
2. the time period required to apply the update is acceptable.

We argue that DSU approaches should not cause significant performance overhead for different reasons. One reason is that if a program gets slow and unresponsive because of dynamic updates, program restarts (instead of dynamic updates) may be the better strategy. In addition, dynamic languages such as Smalltalk, Python, or Ruby already support dynamic software updates. On the contrary, Java performs better in terms of program execution speed compared to dynamic languages (a comparison can be found at [FG11]). Ending up with an updated Java program whose execution speed is worse than the execution speed of the same updated program based on a dynamic language might be a good reason to prefer dynamic languages. Last but not least, dynamic software updates that cause comparable or even longer time periods of application unavailability than updates through program restarts are of no avail. Generally, users virtually always prefer a good performing approach over a comparable but worse performing one (particularly when the program is supposed to be used in production).

1.2.3 Platform Independence

In general, software needs a platform on which it can be executed. Java programs are typically executed on-top of a *Java Virtual Machine (JVM)*. Today, many different JVM implementations exist. Examples are the Oracle (formerly Sun) HotSpot JVM,[7] the Oracle JRockit JVM,[8] and IBM JVM.[9] We consider a dynamic update approach as *platform independent*, if it can be applied to different execution platforms, i.e., different Java virtual machines, while avoiding expensive adjustments to the Java-specific interpretation of platform independence.

In our opinion, DSU approaches should not cause dependencies to specific JVM implementations because platform independence is one of the reasons for the success of Java. For

[7] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html

[8] http://www.oracle.com/technetwork/middleware/jrockit/overview/index.html

[9] http://www.ibm.com/developerworks/java/jdk/

instance, it may be no good idea to force the customer to use the IBM JVM (which is not available for Windows) even though the customer only runs Windows-based machines. Other customers may prefer a specific JVM, e.g., for performance reasons, and may miss out when the DSU approach does not support this specific JVM.

1.2.4 Architecture Independence

Software can be designed in many different ways. Garlan et al. [GBI+10] describe the term software architecture (design) as follows:

> *The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.*

In correspondence to this description, we characterize a dynamic software update approach as *architecture independent* if it:

1. can be combined with every program no matter what software architecture it bases on.
2. has no influence on the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

We pick up *architecture independence*, because in software development there is no such thing like “one architecture fits all scenarios”, i.e., different scenarios may require different architectures. Thus, DSU approaches should not restrict the usage of different architectures, i.e., they should be capable of being integrated into the program’s natural architecture.

1.2.5 Update Granularity

There is a big difference between the granularity of the program changes made and the granularity of the corresponding update that have to be performed in order to apply the changes to the running program. For instance component-based DSU approaches require to update whole components even if only one class has changed. We consider a DSU approach as *fine-grained* if:

1. it requires to update only the program parts that have changed.

Generally, we aim at fine-grained DSU approaches for efficiency reasons. Coarse-grained DSU approaches require to not only map the state of changed program parts but also of unchanged program parts. Those additional state mappings are time consuming and thus increase the time required to perform the update.

However, different criteria may be of different importance to different stakeholders. For instance, users may emphasize *high flexibility*, whereas administrators may attach great importance to *platform independence*. Furthermore, companies may set value on *architecture independence*, because they offer products for different domains which require different software architectures. That is, in order to satisfy all stakeholders, a DSU approach must fulfill all mentioned criteria.

All in all, the goal of this work is to develop a dynamic software update approach which offers the most complete package and is useful in real-world scenarios, i.e., a solution with the best possible practical benefit. In our opinion particularly the chosen criterions and optimal solutions regarding them are crucial to achieve this goal.

1.2.6 Other Criteria?

With the previous section, we explained the goals of this work and justified our criteria selection. But, there are other criteria which are just as important as the criteria central to this work. One example are *consistent program updates* – in a nutshell, the ability to update applications in such way that they most certainly do not produce wrong results after the update. We chose the above mentioned criteria above other criteria such as consistent program updates, because we think that the chosen criteria must be fulfilled **before** we can go on solving other issues. Put simply, if a program cannot be dynamically updated, there is no need to think about how to ensure the program's consistency beyond dynamic updates. However, even if not central part of our considerations, we will discuss possible (partial) solutions for the consistency problem and their applicability in our DSU approach in Chapter 6.

1.3 Contributions

After we detailed our main objectives, we summarize the contributions of the here presented work.

Goals at Design Level. We develop a novel and dynamic software update approach for Java, which:

- allows the developer to flexibly update a program.
- does not significantly decrease the performance of the program.
- works with all major standard Java virtual machines (without modifications).
- does not depend on specific software architectures.
- only updates the program parts scheduled for an update.

Tool Support. We provide tool support for the developed concepts:

- We provide a tool, which is useful in practice.
- We provide a tool, which is easy to use.
- We detail our implementation.

Case Studies. We apply our tool to different non-trivial (real-world) case studies:

- We demonstrate the practicability of our solution.
- We show the usefulness of the tool.

Empirically Goals. We evaluate the fulfillment of our main objectives:

- We demonstrate the flexibility of our tool.
- We measure the performance of our solution on the basis of different case studies and micro benchmarks.
- We apply our approach to all major Java runtime environments demonstrating its platform independence.
- We give reasons for why our solution is architecture independent.
- We show that our tool only updates the program parts that must be updated and thus hits the targeted fine level of update granularity.

1.4 Thesis Structure

In the following chapters, we will reveal the details of our approach for unrestricted dynamic software updates of Java applications. The chapters are structured as follows:

With **Chapter 2**, we provide background information on the field of research, i.e., dynamic software updates. We introduce basic terms and definitions and argue why DSU for statically typed languages are challenging. We further describe the internals of Java's runtime environment (i.e., the Java Virtual Machine).

Within **Chapter 3**, we present the conceptual core of our DSU approach. We describe how JAVADAPTOR integrates with the JVM and detail how our approach combines class reloadings, Java HotSwap, containers, and proxies to hit the targeted high level of update flexibility.

Chapter 4 reveals the implementational details of our DSU approach. We first describe our tool from the users point of view. Then, we go through every part of the tool's workflow and detail its implementation including possible pitfalls.

In **Chapter 5**, we evaluate if our solution meets the thesis goals, i.e., if the approach offers highly flexible, performant, platform independent, architecture independent, and fine-grained dynamic software updates.

Within **Chapter 6**, we summarize ongoing and future work to extend and improve the presented DSU approach and present first results of our efforts.

In **Chapter 7**, we discuss the related work and review it with respect to our evaluation criteria.

With **Chapter 8** we sum up the presented work, list our contributions, and finally conclude the thesis.

2 Dynamic Software Updates

With this chapter, we provide background information necessary to understand the domain of dynamic software updates and its challenges. Furthermore, we impart the fundamentals required to understand our solution to overcome the challenges regarding Java.

We start with an abstract of terms and definitions regarding the field of research. Next, we outline why dynamic software updates for statically typed languages are challenging. Last but not least, we detail the characteristics of the Java Virtual Machine and discuss how it prevents and supports dynamic software updates.

2.1 Terms and Definition

Research in the field of dynamic software updates has a long tradition and a lot of approaches and solutions have been proposed over the years. Gupta et al. [GJB96] define dynamic software updates (i.e., on-line program changes) as followed:

Definition. *An on-line change from program* Π *to* Π*' at time* t *using the state mapping* S*, in process* P *(executing* Π*) is equivalent to the following sequence of steps:*

1. P *is stopped at time* t *in state* s *(say).*
2. *The code of* P *(which, till now, was the program* Π*) is replaced by the program* Π*', its state is mapped by* S *and* P *is then continued (from state* S(s) *and with code* Π*').*

However, even if precisely defined what this field of research is about, there is no standard phrase for it commonly used by the community. That is, Gupta et al. [GJB96], for instance, call it *on-line program change*, whereas researchers such as Gustavsson et al. [GSA04], Ebraert et al. [EVDB05], Di Stefano et al. [SPT04], Malabarba [MPG$^+$00], Oriol [Ori04], Würthinger et al. [WWS10], or Dmitriev [Dmi01a] prefer the term *runtime/dynamic (software) evolution*. Fabry, as one of the pioneers of the research area, was talking about *on the fly module changes* [Fab76]. Other synonyms for dynamic software updates are *dynamic adaptation* [RC02] or *runtime adaptation*, e.g., used by Griffith and Kaiser [GK06] and Morin et al. [MLH$^+$09]. In this thesis, we conform to researchers such as Previtali and Gross [PG06], Orso et al. [ORH02], Hicks and Nettles [HN05], Shen et al. [SSH$^+$05], Zhang and Huang [ZH06], Hayden et al. [HSHF11], or Bialek [Bia06] and use the term *dynamic (software) updates*.

2.2 Dynamic Software Updates vs. Type System

Because most modern dynamic languages widely support dynamic software updates, research efforts of the (recent) past regarding DSU have been focussed mainly on statically typed languages. But, what makes dynamic software updates based on statically typed languages such challenging that most of those languages still do not have the same runtime update capabilities modern dynamic languages have?

As Vandewoude et al. describe in [VEBD05], the main reason are the constraints the type system of statically typed languages imposes. In statically typed languages, types are assigned to variables in early stages of the program build process (e.g., at compile time), whereas types in dynamic languages are bound at runtime and thus can be changed even if the program is already executed. Listing 2.1 illustrates the situation. In dynamically typed Smalltalk, variables could be declared independent from any type (see Line 2), whereas statically typed languages such as Java require to specify the name as well as the type of a variable (see Line 5). That is, variable declarations such as depicted in Line 6 of Listing 2.1 are not possible.

Listing 2.1: Variable declaration in Smalltalk and Java.

```
1 //variable declaration in Smalltalk
2 |varX varY| ✔
3
4 //variable declaration in Java
5 TypeA varX; ✔
6 varY; ✘
```

Another fact that hinders DSU in statically typed languages is static type checking, i.e., the process to verify whether the operands of an operator are type compatible or not. As a result, it is impossible to arbitrarily alter operand and operator types in statically typed languages (see Line 9 of Listing 2.2).

Listing 2.2: Type compatibility in Smalltalk and Java.

```
1 //type compatibility in Smalltalk
2 |varX|
3 varX := TypeA new. ✔
4 varX := TypeB new. ✔
5
6 //type compatibility in Java
7 TypeA varX;
8 varX = new TypeA(); ✔
9 varX = new TypeB(); ✘
```

Group	**Dimension**	**Static**		**Both**		**Dynamic**	
		Weak	Strong	Weak	Strong	Weak	Strong
Temporal (when)	Time of change						
	Offline changes	++	++	++	++	+	+
	Online changes	- -	- -	+/-	-	++	+
	Change history	++	+	++	+	+	+/-
	Change frequency	-	- -	-	- -	++	+
Object of change (where)	Anticipation	-	-	+/-	+/-	+	+
	Granularity						
	Coarse-grained	++	++	++	++	+	+
	Fine-grained	-	-	-	-	++	++
	Impact	+	++	+/-	+	- -	-
	Change propagation	+/-	+/-	+/-	+/-	++	+
System properties (what)	Availability	-	-	+/-	+/-	+	+
	Openness	+/-	-	+	+/-	++	+
	Safety	+/-	+	+/-	+	- -	-
Change support (how)	Automation degree	-	-	+/-	+/-	+/-	+/-
	Formality degree	+	++	+	++	-	+/-
	Change type	+	+	+	+	+/-	+/-

Table 2.1: Type system impact on dynamic software updates [VEBD05].

Furthermore, in statically typed languages, type conversions are rather restricted. They must be either explicitly declared (through type casts) or must comply with language features such as polymorphism (if any supported). That is, type conversions such as depicted in Line 9 of Listing 2.2 would be only valid if both types are members of the same inheritance hierarchy.

In addition to the mentioned constraints imposed by static typing, chances to establish DSU support rely on how good a language (its type system) prevents type errors. Weakly typed languages (i.e., languages with poor type error prevention) offer possibilities to circumvent the type system (i.e., to run unsafe code) which opens the door to establish DSU support, while strongly typed languages (i.e., languages with good type error prevention) do not.

Buckley et al. proposed in [BMZ$^+$05] a taxonomy regarding different aspects meaningful to software updates, i.e., a list of questions and subsequent questions about the *when*, the *where*, the *what*, and the *how* of software updates. Vandewoude et al. [VEBD05] put this taxonomy and summarized how the type system affects the taxonomy's aspects (see Table 2.1). As shown in Table 2.1, the best possible basis for dynamic software updates is offered by weakly and dynamically typed languages. On the contrary, chances are low to provide strongly and statically typed languages with substantial dynamic software update capabilities. In terms of Java, as a strongly and statically typed language with few features known from dynamic languages such as limited support for late type binding (i.e., polymorphism) or basic reflection capabilities, chances are only slightly better (compared to strongly and statically typed languages) to establish wide DSU support (see gray colored Column 6 of Table 2.1). A priori, Vandewoude et al. [VEBD05] rate the chances to apply dynamic update capabilities to Java as very low (see dark-gray colored Row 5 Column 6 of Table 2.1). However, even if a DSU approach can be established, they only give coarse-grained runtime updates (e.g., based on components) good chances to be realized (see dark-gray colored Row 10 Column 6 of Table 2.1) but not those we aim at in this work, i.e., fine-grained dynamic software updates.

2.3 The Java Virtual Machine

In order to understand what is provided or possible in Java and what challenges remain regarding dynamic software updates, it is necessary to understand the standard design of Java's runtime environment – the Java Virtual Machine (JVM). In this section, we summarize all JVM specific information relevant to our update approach. We first describe the basic components and the class loading concept of a JVM, which are specified in *The Java Virtual Machine Specification* [LY99] and thus are standard for all certified JVM implementations. Afterwards, we detail the internals of Java HotSwap and describe the Java Platform Debugger Architecture (JPDA), which, even if not part of the specification, are standard for all major certified JVMs.

2.3.1 Architecture

Modern JVMs are highly complex programs which are composed of many different components. Figure 2.1 shows the most important parts of a JVM. It is a matter of common knowledge that Java programs consist of *classes*. In order to execute a Java program, the program specific classes have to be loaded into the JVM, which is done by the *class loader subsystem*. After a class is loaded into the JVM, the program code (i.e., the bytecode) it contains may be scheduled for execution. The bytecode-execution job is done by the *execution engine*. In order to execute a Java program, memory to store information, such as bytecodes, intermediate results of computation, local variables, etc., is needed. This memory

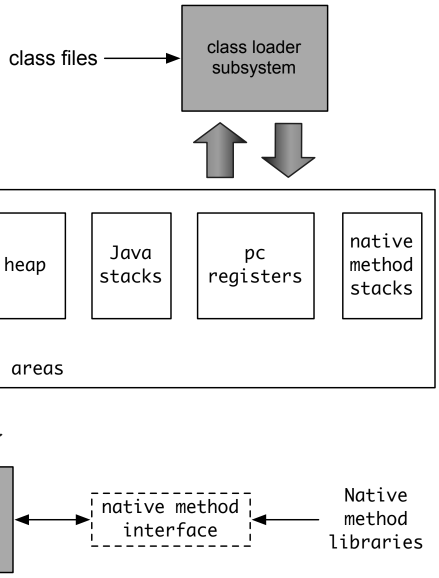


Figure 2.1: The internal architecture of the Java virtual machine [Ven00].

area is called *runtime data area*, which basically consists of the method area, the heap, the Java stacks, pc registers, and the native method stacks.

The *method area* is shared among all JVM threads and stores all class (type) specific data of a loaded class, e.g., the runtime constant pool, field and method data, and the code for methods and constructors [LY99]. The *heap* is the memory area which stores the runtime data of all class instances and arrays. Like the method area, the heap is shared by all JVM threads. The next parts of the runtime data area are the *Java stacks*. Each Java stack belongs to one specific JVM thread and stores information relevant to method invocations/executions, e.g., intermediate calculations, local variables, parameters, or return values. Dedicated to each stack is a *pc register* which contains the address of the currently executed bytecode instruction. Different from the Java stacks and the pc registers, which are responsible for executing bytecodes, *native method stacks* get involved if the Java program requires to execute native methods. Like with the Java stacks, each JVM thread gets its own native method stack. To manage the execution of native methods a *native method interface* is provided.

For further information about the standard JVM architecture (i.e., the components and their tasks) see the work of Venners [Ven00] and Lindholm and Yellin [LY99].

2.3.2 Class Loading

As mentioned above, to run a Java program, the JVM must at first load the program's classes, which is done by the class loader subsystem. In order to avoid long program startup times,

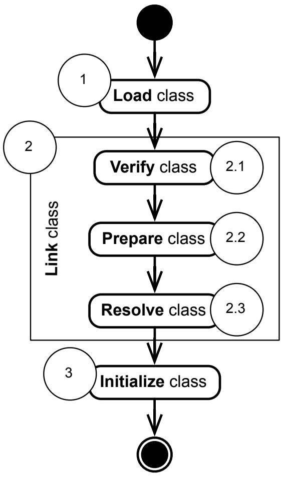


Figure 2.2: Class loading.

the JVM never loads classes other than the classes it right now needs to execute the program, i.e., the JVM loads the required classes lazily. If the JVM must load a class, it performs the class loading steps depicted in Figure 2.2. But, before the JVM processes the class loading steps described in the following, it will check whether the class is already loaded or not. In case the class is already loaded, the class loading process will be aborted.

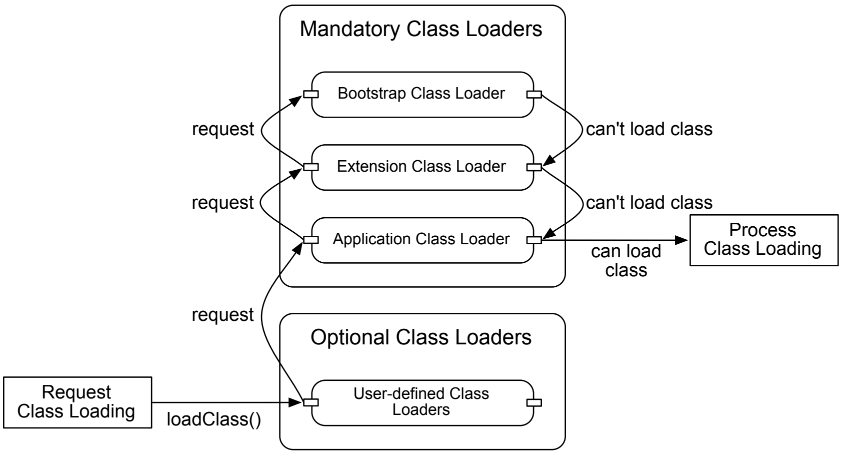


Figure 2.3: Class loader hierarchy.

If the class was not previously loaded, the JVM reads in the binary representation of the class, i.e., the class file (see Action 1 of Figure 2.2). This is either done by the *bootstrap* class loader (loads system classes), the *extension* class loader (loads classes of the extension library), the *application* class loader (loads classes from classpath), or *user-defined* class loaders (load classes from user-defined locations). As shown in Figure 2.3, the class loaders are hierarchically ordered with the bootstrap class loader acting as root. To load the class file, the class loader that received the request for class loading delegates this request to its predecessors (see Figure 2.3). The first class loader that really tries to load the class file, is the bootstrap class loader. If the bootstrap class loader is not able to load the class file, it will ask its successor to load the class and so forth. The next class loader in the hierarchy that is able to load the class file (e.g., the application class loader, see Figure 2.3) processes the class loading and creates the class object for the loaded class. We point out that the class will be finally bounded to the class loader and none of the other class loaders is allowed to load or reload this class afterwards.

With the next step, the loaded class will be linked (see Figure 2.2, Action 2). The class linking process consists of three substeps. At first, the loaded class will we verified (Action 2.1). That is, the JVM checks whether the loaded class file is structurally valid or not. The preparation phase (Action 2.2) triggers the creation of static fields and their initialization with standard default values (note that the class's static initializers will be executed in later class loading steps). The class resolution step (Action 2.3 of Figure 2.2) concludes the linking process. Here, the symbolic references in the runtime constant pool of the class object will be resolved to concrete values (e.g., a symbolic class reference will be transformed to a reference to the concrete class, which triggers to load the referenced class as well if not already done).

To complete the class loading process, the successfully linked class must be initialized (see Figure 2.2, Action 3). That is, the static initializers of the class are executed. If this is done, the class is fully applied to the running program and thus ready to be executed.

Further information on the class loading concept of Java are given by Venners [Ven00], Lindholm and Yellin [LY99], and Gosling et al. [GJSB05].

2.4 Dynamic Software Updates and the JVM

Having described the basic architecture and the class loading concept of the JVM, it is time to have a look into how dynamic software updates are supported or restricted by the JVM.

Changing a program during its execution in the JVM requires to modify the data within the heap, the method area, and on the stacks. For instance, program updates, which include method replacements, field removals, or inheritance hierarchy changes, require to extensively change the data of a class. In general, they require to modify the class schema. Unfortunately, the JVM does not permit class schema changes, because class schema changes may let the data on the stack, on the heap, and the class data stored in the method area become inconsistent, while the JVM does not provide functions to synchronize them.

Moreover, the class loading concept described above makes it difficult to circumvent the restrictions of the JVM regarding class schema changing dynamic software updates. One problem is that we could not simply reload already loaded classes in order to update the class schema, because an already loaded class cannot be reloaded by any of the systems class loaders.

Nevertheless, there are two ways to reload (update) a class with a changed schema. First, we could unload the old class version before we load its up-to-date counterpart. However, a class could be only unloaded if the owning class loader can be garbage collected. Unfortunately, a class loader can only be garbage collected if all classes (even the unchanged ones) loaded by this class loader are dereferenced, which is equivalent to a (partial) application stop. This fact disqualifies class unloadings for DSU purposes. Alternatively, we could reload an already loaded class with a customized class loader [LB98]. But, in order to efficiently handle dynamic software updates on the basis of customized class loaders, components are required, which have major disadvantages compared to single class updates. Amongst others, component-based dynamic software updates require to partially stop the program and result in extensive state mappings. For all these reasons, customized class loaders are not in our scope as well.

2.4.1 Java HotSwap

Despite the insufficient native dynamic software update support of the JVM, there is one feature that provides some simple runtime update capabilities – called *Java HotSwap*. It allows the developer to dynamically replace method bodies (which partly covers points 4 – 6 of Table 1.1 presented in Section 1), but restricts class schema changes and thus highly flexible dynamic software updates. Even if HotSwap is not a standard feature, it is implemented by all major certified Java virtual machines commonly used in production, i.e., the HotSpot JVM,[1] the JRockit JVM,[2] and IBM's JVM.[3] In the following we will describe the basic mechanism of Java HotSwap. Because Java HotSwap was originally developed by Dmitriev [Dmi01a], we describe the mechanism on the basis of his implementation for the HotSpot JVM.[4] Note that HotSwap may be implemented in a different way in other JVMs.

Central to Java HotSwap is the class object, which the JVM creates during class loading. Figure 2.4 shows the basic elements of the class object. The most important part of the class object is object `InstanceClass`, which is the entry point for requests for all instances of the class. Another part of the class object is the `constantPoolOop` object. It conforms to the constant pool structure and maps indexes to the corresponding constants. Because accesses to elements such as fields or methods solely through object `constantPoolOop` would be slow, which is due to the fact that those accesses require to obtain additional indexes, the access

[1] `http://www.oracle.com/technetwork/middleware/jrockit/overview/index.html`

[2] `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html`

[3] `http://www.ibm.com/developerworks/java/jdk/`

[4] Java HotSwap officially debuted in version 1.4.2 of the HotSpot JVM.

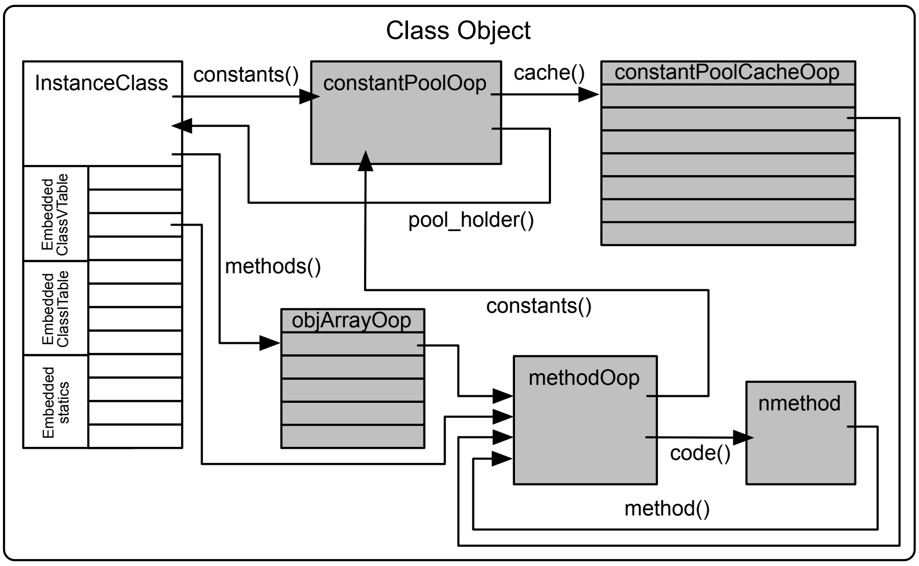


Figure 2.4: Internal class object representation [Dmi01a].

information is cached in object `constantPoolCacheOop`. The next important part of the class object is object `methodOop`, which stores the class's bytecode. The corresponding binary code produced by the *just-in-time (JIT)* compiler is stored in object `nmethod`. Not less important for Java HotSwap are the virtual method table (`ClassVTable`), the interface method table (`ClassITable`), and the representations of the static fields (`statics`), which are used to invoke the methods and static fields of the class.

In order to redefine an already loaded class, Java HotSwap proceeds as follows. First of all, the new version of the class to be redefined is loaded into the JVM and the class object for this class is created. Next, the class must successfully pass the verification phase. If this is done, it is checked if new and old class share the same class schema, i.e., if both classes have the same methods, fields, interfaces, super types, etc. If the class schemas differ, the JVM rejects the update request. In case the changes within the new class version do only affect the method bodies but not the class schema, the JVM starts the class transformation process.

With the first step of the class transformation process, all compiled methods of the class scheduled for redefinition and all compiled code that relates to the class are deoptimised, i.e., the JVM quits the execution of binaries and resumes to execute the corresponding bytecode. To ensure that every call to a compiled method, will be redirected to the corresponding bytecode-based method, traps will be installed inside the compiled methods (see Marker 1 in Figure 2.5). Next, the indexes into the constant pool (i.e., object `constantPoolOop`) are updated, because the layouts of the constant pools of the old and new class object

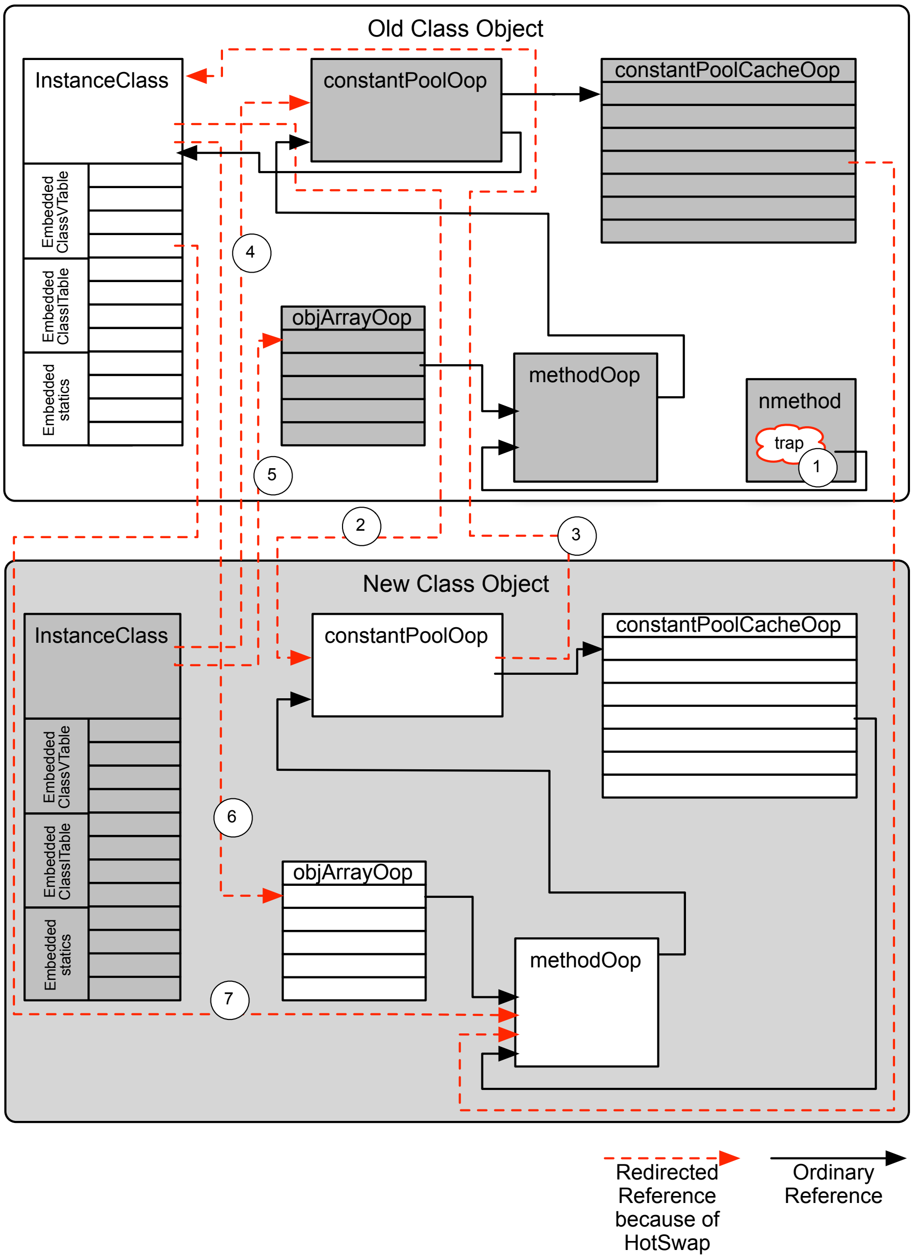


Figure 2.5: Java HotSwap mechanism [Dmi01a].

may differ, which may let the indexes point to wrong constants. After this is done, the pointer from old InstanceClass to old constantPoolOop will be redirected to the new constantPoolOop, while the new constantPoolOop object now points to the old InstanceClass object (see Figure 2.5, Markers 2 and 3). Furthermore, the HotSwap mechanism lets the new InstanceClass object point to old constantPoolOop and objArrayOop (see Markers 4 and 5). This is done, to keep the latter two objects from being garbage collected. In the next step, the methods of the old class object will be replaced by the methods of the new class object. Therefore, the HotSwap mechanism lets the old InstanceClass refer to the new method pointer array (i.e., objArrayOop, see Marker 6). After method replacement, all old methods are marked as obsolete, which prevents the recompilation of those methods. In the last step, the virtual and interface method tables (i.e., ClassVTable and ClassITable) of the old InstanceClass object are re-initialized, which is necessary to let the method tables point to the new methods, i.e., to new methodOop (see Figure 2.5, Marker 7). This step must be processed for possible subclasses of the redefined class as well. To conclude the class redefinition process, the HotSwap mechanism iterates over all Java classes that belong not to the core and searches the classes constant pool caches (i.e., object constantPoolCacheOop) for pointers to old methods and if found, updates those pointers so that they point to the corresponding new methods.

Ones the class is redefined, the new (updated) methods can be executed. That is, all new method calls will go to the up-to-date method and not to the outdated method. But, what happens with old methods active on the stack? In the current Java HotSwap implementation, old methods active on the stack will remain on the stack until their execution has finished. That is, old and new method versions may coexist in the JVM. We will discuss in Chapter 6 what problems may arise from this fact and how those problems could be solved.

All in all, Java HotSwap is a valuable feature supporting dynamic software updates. It does not enable highly flexible dynamic updates, but it is a good starting point for the development of DSU approaches aiming at high update flexibility.

For deeper insights into the above sketched Java HotSwap mechanism, see Dmitriev's original descriptions [Dmi01a, Dmi01b].

2.4.2 Java Platform Debugger Architecture

As mentioned above, there is limited support for DSU natively provided by all major JVMs – namely Java HotSwap – which allows the developer to redefine method bodies, i.e., to apply class-schema-retaining program updates. The HotSwap feature is part of the *Java Platform Debugger Architecture (JPDA)* [Ora11b], which constitutes the framework to debug Java applications. The JPDA is standard for all major certified JVMs. As depicted in Figure 2.6, the JPDA consists of two interfaces and one protocol.

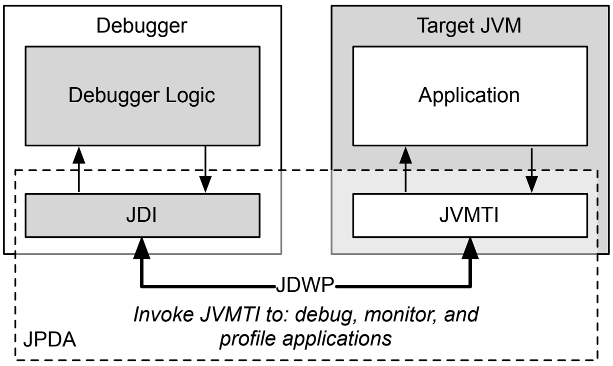


Figure 2.6: Java Platform Debugger Architecture.

The central part of the JPDA is the *Java Virtual Machine Tool Interface (JVMTI)* [Ora11c] which is integral part of the JVM. It extends the *Java Native Interface* [Lia99] and provides functions that allow the developer, e.g., to inspect the currently executed code, to request loaded classes and all their existing instances, to pause and resume individual program threads, or to trigger class redefinitions based on Java HotSwap. That is, it provides functions to inspect and control applications during their runtime. Those functions can be invoked from outside the JVM executing the application to be debugged.

The communication between debugger and targeted JVM bases on the *Java Debug Wire Protocol (JDWP)*. The exchange of JDWP messages between debugger and target JVM is either based on *Socket Transports* (OS independent) or *Shared Memory Transports* (which are only available for Windows). Even if the JDWP is optional and might be not available in some JVM implementations (in which case a proprietary protocol must be implemented in order to invoke the JVMTI), all major JVMs implement it. That is, a once written debugger works with all JVMs which implement the JDWP.

Basically, JVMTI and JDWP are sufficient to implement debuggers for Java applications. A debugger has only to implement the JDWP and thus would be able to debug Java applications. However, the JPDA additionally provides the *Java Debug Interface (JDI)*,[5] which implements the JDWP and provides an easy to use interface to let debugger and target JVM communicate with each other.

2.5 Summary

Within this chapter, we imparted crucial knowledge on the field of research, i.e., dynamic software updates. We introduced basic terms and definitions and argued why dynamic software updates on the basis of statically typed languages are challenging. Since we are

[5] `http://download.oracle.com/javase/6/docs/jdk/api/jpda/jdi/index.html`

aiming at DSU solutions for statically typed Java, we further detailed how Java, respectively its runtime environment (i.e., the JVM), enables or prevents dynamic software updates. To sum up, DSU in Java is severely limited. Due to the limitations, approaches are required, which provide Java with highly flexible DSU.

3 Concepts of JAVADAPTOR

This chapter shares material with the SPE'2011 paper "JavAdaptor – Flexible Runtime Updates of Java Applications" [PKC^{+}12].

As already stated, the primary goal of this work is to provide Java with *highly flexible*, *performant*, *platform independent*, *architecture independent*, and *fine-grained* dynamic software updates. In this chapter, we present the design and basic concepts of our approach fulfilling these criteria – namely JAVADAPTOR.

At first, we present a small program building the base of our explanations. Next, we introduce the basic architecture of JAVADAPTOR. Afterwards, we discuss the mechanisms and concepts of JAVADAPTOR which allow us to dynamically update running programs. In this regard, we proceed in a bottom-up manner starting with descriptions of concepts for simple program updates followed up by descriptions of advanced concepts for complex program updates.

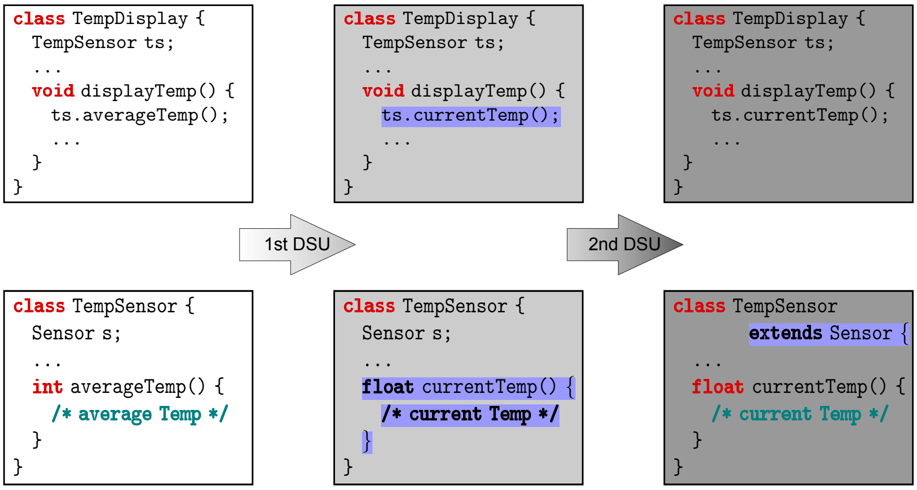
3.1 A Running Example: Weather Station

The following sections and chapters are filled with detailed descriptions on our DSU approach, i.e., JAVADAPTOR. To minimize confusions because of the amount of details, we decided to describe JAVADAPTOR on the basis of a running example.

As the central theme of our descriptions, we chose the small weather station program depicted in Figure 3.1. The weather station consists of two classes. One class (`TempSensor`) measures the air temperature while the other class (`TempDisplay`) is responsible for displaying the measured temperature.

Consider a maintenance task: the actual measuring algorithm (average temperature) must be replaced by another measuring algorithm (current temperature). Because the service provided by the weather station must be non-stop available, stopping the program in order to apply the necessary changes is no option; we want to change it at runtime. The application of the new functionality requires to change different parts of the program. First, we must replace method `averageTemp` of class `TempSensor` by method `currentTemp`, which requires to change the class schema. Second, we must redefine method `displayTemp` of class `TempDisplay` in order to execute the new measuring algorithm.

Short time after updating the measuring algorithm, it was also decided to let `TempSensor` inherit from class `Sensor` in order to add new functions to `TempSensor` while avoiding to implement them again. Therefore, we have to apply statement `extends Sensor` to class `TempSensor`. Additionally, we must remove member `s` from class `TempSensor`, because superclass `Sensor` let it become useless.



```
class TempDisplay {
  TempSensor ts;
  ...
  void displayTemp() {
    ts.averageTemp();
    ...
  }
}
```

```
class TempDisplay {
  TempSensor ts;
  ...
  void displayTemp() {
    ts.currentTemp();
    ...
  }
}
```

```
class TempDisplay {
  TempSensor ts;
  ...
  void displayTemp() {
    ts.currentTemp();
    ...
  }
}
```

```
class TempSensor {
  Sensor s;
  ...
  int averageTemp() {
    /* average Temp */
  }
}
```

```
class TempSensor {
  Sensor s;
  ...
  float currentTemp() {
    /* current Temp */
  }
}
```

```
class TempSensor
        extends Sensor {
  ...
  float currentTemp() {
    /* current Temp */
  }
}
```

Figure 3.1: Weather station. The depicted example spans updates which replace methods, remove fields, and change inheritance hierarchies. That is, the updates require class schema changes.

Even if the required program changes seem to be simple, they affect many different parts of the program (i.e., see points 1-6, Table 1.1 in Section 1). The updates particularly require to change the schema of already loaded classes, which is not supported by the Java runtime. Therefore, we search for a new mechanism in Java that allows us to change every part of a program at runtime without anticipating the changes.

3.2 Architectural Design of JAVADAPTOR

As aforementioned, there is severely limited support for DSU natively provided by all major JVMs – namely Java HotSwap. Because HotSwap is an integral part of JAVADAPTOR, the architectural design of JAVADAPTOR must be geared to the natural working environment of HotSwap, i.e., the Java Platform Debugger Architecture we already described in Section 2.4.2.

Because the JPDA is for debugging, profiling, or monitoring and not for dynamic software update purposes, we have to modify it to our needs. The lower part of Figure 3.2 shows how we changed the JPDA to host our DSU approach. We simply replace the debugger by

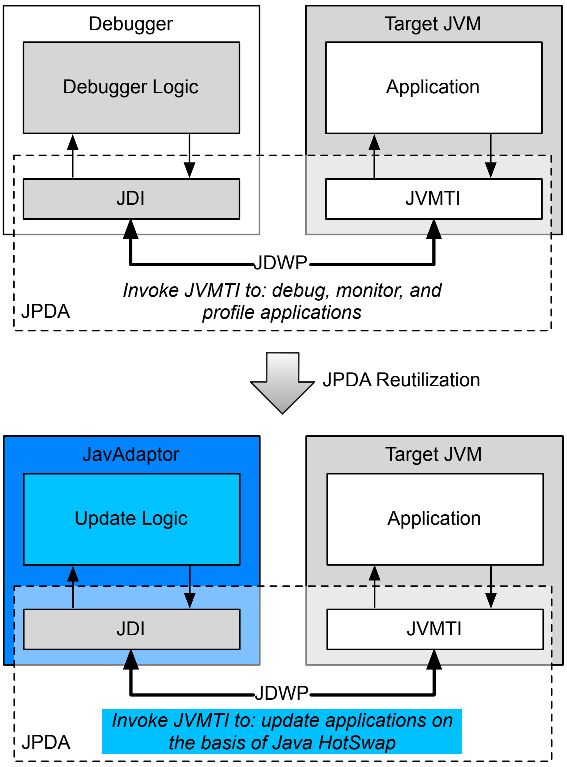


Figure 3.2: The architectural design of JAVADAPTOR.

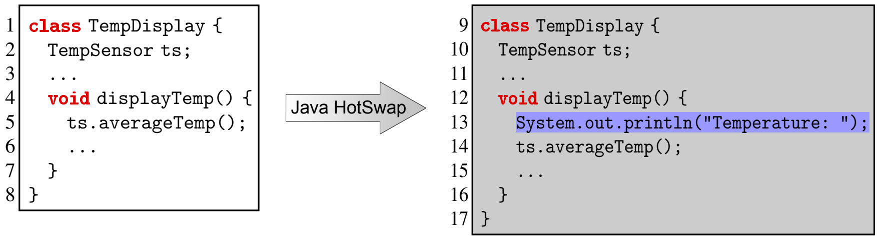
JAVADAPTOR, which contains update logic instead of debugging logic and uses the JPDA for dynamic software update purposes and not for debugging, profiling, or monitoring purposes.[1]

3.3 Updates without Affecting the Class Schema

After describing the architecture of JAVADAPTOR, we go on explaining the concepts of our approach. The simplest kind of update we must support with JAVADAPTOR are updates which have no influence on the class schema, i.e., method body redefinitions such as depicted in Figure 3.3.

However, there is no trick at providing this kind of update with JAVADAPTOR, because class-schema-retaining runtime updates (i.e., method body redefinitions) are already supported by the JVM through feature Java HotSwap. That is, within JAVADAPTOR we simply make use of Java HotSwap in order to apply class-schema-retaining dynamic software updates.

[1] JAVADAPTOR can be of course combined with an existing debugger to provide debugging as well as DSU functionality.



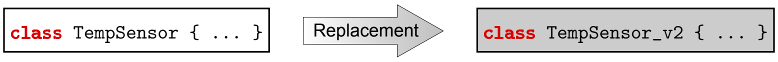
```
class TempDisplay {
  TempSensor ts;
  ...
  void displayTemp() {
    ts.averageTemp();
    ...
  }
}
```

```
class TempDisplay {
  TempSensor ts;
  ...
  void displayTemp() {
    System.out.println("Temperature: ");
    ts.averageTemp();
    ...
  }
}
```

Figure 3.3: Updates solely using Java HotSwap.

3.4 Class Reloading

In Section 2.3, we discussed why the JVM does not permit class schema changing dynamic software updates. We further identified the mechanisms within the JVM which ensure that the schema of a loaded class cannot be changed, i.e., the strict class loading that irreversibly bounds a class to a specific class loader. However, we found that customized class loaders [LB98] could be used to flexibly update a running application, i.e., they could be used to reload classes with changed class schemas. But, in order to passably handle dynamic software updates on the basis of customized class loaders components are required, which have major disadvantages compared to single class updates (e.g., they are invasive, they partially stop and restart programs, they require extensive state mappings, etc.) and thus are not in our scope. Design patterns such as proxies and wrappers/decorators are no alternative to customized class loaders because they do not even achieve their level of flexibility, which amongst others is because those patterns do not allow to remove methods and complicate inheritance changing program updates.



```
class TempSensor { ... }
```

```
class TempSensor_v2 { ... }
```

Figure 3.4: Class renaming.

Because of the disadvantages of the above mentioned approaches, we use another strategy to update the schema of an already loaded class – namely *class renaming*. As exemplified in Figure 3.4, the key idea is that, while we cannot load a new class version with the same name, we rename the new version and load it under a fresh name. Since the resulting class name is not registered in any class loader, the updated class can be loaded by the same class loader that also loaded the original class.

3.5 Reference Updates

While class renamings allow us to load a new version of an already loaded class even if the class schema has changed, the mechanism only triggers the loading of the updated class, but not its deployment, i.e., its usage within the application (up to now the old class version is still deployed). To let the new class version become part of program execution, the references to the original class have to be changed to point to the new class version. The primary goal regarding the reference update step is to update the references **without** changing the schemas of the referring classes, because this would cause additional class replacements and at the worst require to essentially replace all classes of the system and thus let our DSU approach become inefficient. For the sake of clarity, in the following, we will name the classes which hold references to classes to be reloaded (updated) *caller classes* and the classes subject to updates *callee classes*. In addition, the terms *caller side* and *callee side* cover the class itself as well as all its instances.

3.5.1 Caller-Side Detection

Before a newly loaded class version can be deployed, we have to identify all program parts referring the old class version. In this regard, we first parse all class files of the system and look for outdated references within the bytecode. However, parsing the bytecode is not enough because we can only gain information about static references within the bytecode, but no such information as the number of objects, the relationships between specific objects, the values of fields, etc. (i.e., the current program state). That is, in a second step, we additionally identify all instances from type of the old class and ask the JVM for the caller side of them (i.e., for the classes and instances that refer to the outdated instances).

3.5.2 State Mapping

One of the central goals we aim at with JAVADAPTOR is to preserve the program state beyond updates, which (as we described in Section 2.3) is fully supported by Java HotSwap (i.e., updates solely using Java HotSwap do not cause state losses). But, what could cause program state losses are our class reloadings. Since the JVM considers the new class version just as another class to be loaded without any relation to the old class version, we have to manually map all state from old class to new class. First, we begin with the mapping of the class specific state. That is, we map the values currently assigned to the class fields of the outdated class to the corresponding class fields of the new class, which of course not only includes the declared fields but the inherited fields, too. Then, we continue to map the instance specific state. That is, we create for each outdated instance, identified during caller-side detection, an instance from type of the up-to-date class version and map the state field by field (again including the inherited fields) from old to new instance.

One-to-one Mappings. The simplest state mappings supported by JavAdaptor are one-to-one mappings. JavAdaptor considers state mappings as one-to-one mappings if each field of the old class has its counterpart (i.e., a field with same name, type, and modifiers) in the new class and the number of fields is the same in both classes. In case of one-to-one mappings, JavAdaptor automatically maps the state field by field to the newly created instances. However, there may be updates which require more complex state mappings than one-to-one mappings. In order to even cover those updates, we extended our mapping mechanism.

Removed Fields. To support mappings where the update removes fields, we have to modify the basic one-to-one mapping mechanism only slightly. Fields existing in both classes (i.e., fields with same name, type, and modifiers) are mapped by JavAdaptor as described above. Fields that are only present in the old class are simply ignored.

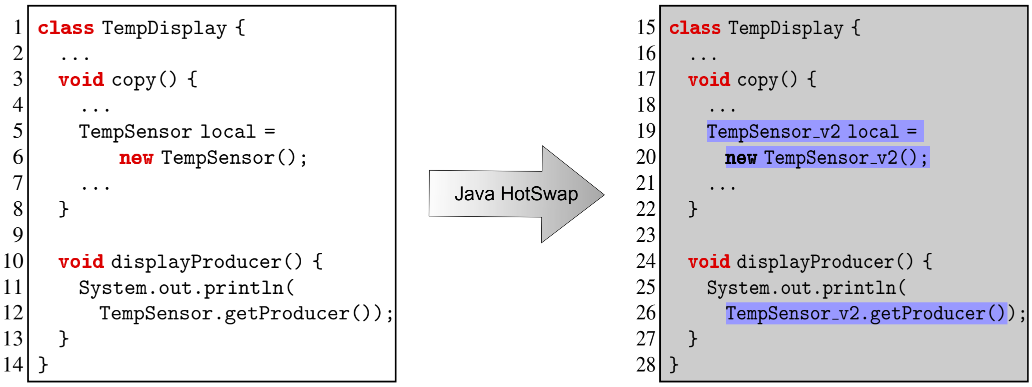
Added Fields. What we have to support in addition, are updates that add fields to the new class. In this case user input is required. The user has to decide whether JavAdaptor should ignore the fields, gives them default values, or initializes them as specified by the user. In case the user wants to specify how the fields are initialized, she has to write an initialization method that JavAdaptor then executes.

Moved Fields. Moved fields can be considered as a combination of removed and added fields. Whereas, moved fields are ignored during the state mapping between the former field owner and its updated counterpart, they must be taken into account during the mapping involving the new field owner. What field of the one class maps to what field of the other class must be specified by the user.

Changed Fields. Another scenario that requires user input, are updates that change fields. We consider a field as changed if either its name, type, or modifier has changed. Like moved fields, changed fields can be considered as a combination of removed and added fields. In case the name has changed, the user has to specify which fields in the old and new class correspond to each other. Because changes to the `static` modifier shift fields between instance and class level (note that each instance has its own field value, whereas at class level only one value exists), the user has to specify how the fields should be initialized. Type changes require user-specified mapping functions.

3.5.3 Class References and References to Local Variables

Having mapped the state and gained the static and runtime information regarding the references to outdated classes, we process the reference updates. First, we update the class references and references to local variables.



```
 1 class TempDisplay {
 2   ...
 3   void copy() {
 4     ...
 5     TempSensor local =
 6         new TempSensor();
 7     ...
 8   }
 9
10   void displayProducer() {
11     System.out.println(
12       TempSensor.getProducer());
13   }
14 }
```

```
15 class TempDisplay {
16   ...
17   void copy() {
18     ...
19     TempSensor_v2 local =
20       new TempSensor_v2();
21     ...
22   }
23
24   void displayProducer() {
25     System.out.println(
26       TempSensor_v2.getProducer());
27   }
28 }
```

Figure 3.5: Caller-side updates regarding class references and references to local variables.

When it comes to local variables, such as variable `local` in method `copy` of class `TempDisplay` (Figure 3.5, Lines 5 – 6), only method body redefinitions via Java HotSwap are required to refer to the new class version. State mappings are not necessary because with each new method execution the local variables are newly created and local variables from previous method executions are garbage collected. Thus, after redefining a method, such as depicted in Figure 3.5 (Lines 19 – 20), the local variables created during method execution will be of type of the updated class (here of class `TempSensor_v2`).

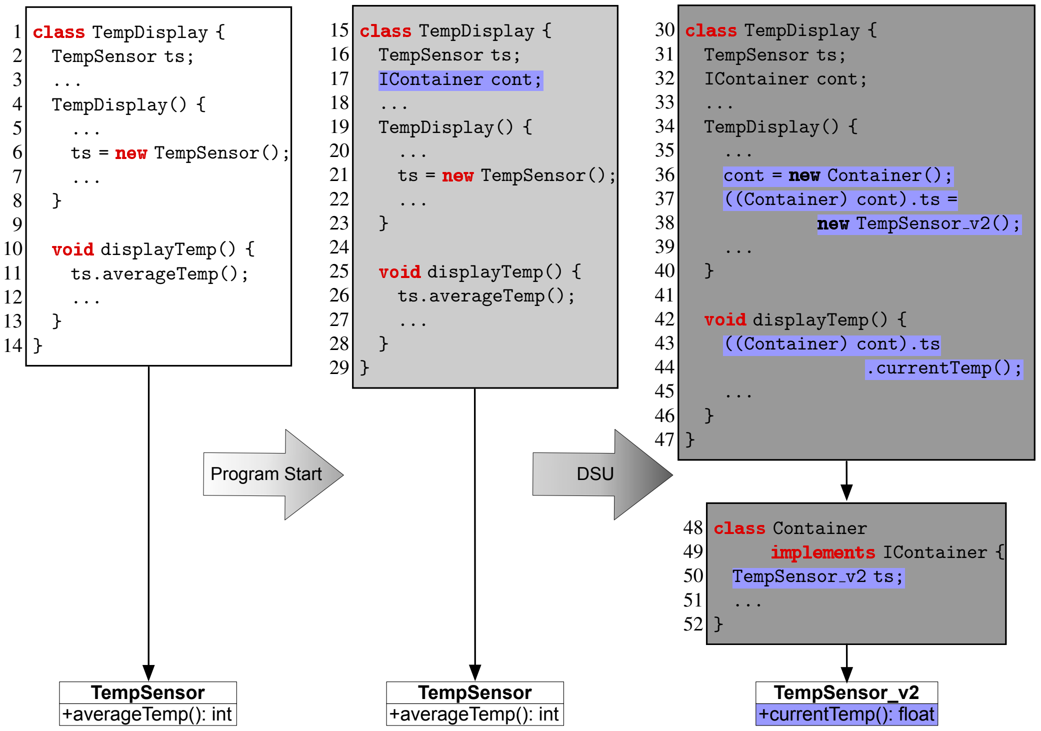
Similar to reference updates regarding local variables, we also update class references (such as depicted in Figure 3.5, Line 12) through method body redefinitions (see Figure 3.5, Line 26). Because we have previously mapped the class specific state from the outdated to the updated class, no additional update step is required.

Finally, we could update the references without touching the schema of the caller class, i.e., without replacing the caller class as well.

3.5.4 References to Long-Living Objects

Different from references to local variables, references to long-lived objects (such as class or instance field references) are vital beyond method executions, i.e., they are inherent parts of the caller side. Thus, caller-side updates because of references to long-lived objects of type of the callee (updated class) must be handled in a different way. We already described the first step to handle those updates (i.e., the state mapping) in Section 3.5.2. In this section, we go on describing how JAVADAPTOR updates the references in order to deploy the updated class and its instances.

Containers. Suppose, we updated a class through class reloading (such as for instance class `TempSensor`, depicted in Figure 3.6). What we now have to do is to update the field references of type of the reloaded class (here of class `TempSensor`) at caller side (in our example of class `TempDisplay` and its instances). Unfortunately, up-to-date version (here version `TempSensor_v2`) and outdated version (here version `TempSensor`) of the replaced class are not type compatible, thus, objects of the up-to-date class version cannot be assigned to fields of type of the outdated class version (such as required to update field `ts` of example caller class `TempDisplay`).



```
 1 class TempDisplay {
 2   TempSensor ts;
 3   ...
 4   TempDisplay() {
 5     ...
 6     ts = new TempSensor();
 7     ...
 8   }
 9
10   void displayTemp() {
11     ts.averageTemp();
12     ...
13   }
14 }
```

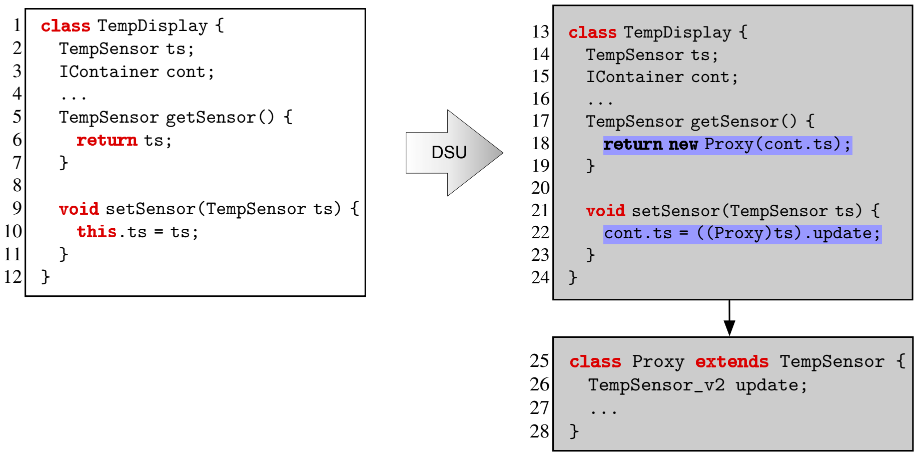
```
15 class TempDisplay {
16   TempSensor ts;
17   IContainer cont;
18   ...
19   TempDisplay() {
20     ...
21     ts = new TempSensor();
22     ...
23   }
24
25   void displayTemp() {
26     ts.averageTemp();
27     ...
28   }
29 }
```

```
30 class TempDisplay {
31   TempSensor ts;
32   IContainer cont;
33   ...
34   TempDisplay() {
35     ...
36     cont = new Container();
37     ((Container) cont).ts =
38               new TempSensor_v2();
39     ...
40   }
41
42   void displayTemp() {
43     ((Container) cont).ts
44                   .currentTemp();
45     ...
46   }
47 }
```

```
48 class Container
49        implements IContainer {
50   TempSensor_v2 ts;
51   ...
52 }
```

Figure 3.6: Reference updates through containers.

To solve the type-incompatibility problem while avoiding to change the caller class schema, we use containers whose usage is exemplified in Figure 3.6. Before program start, JAVADAPTOR prepares the program for the container approach, i.e., it adds field `cont` (Line 17) to each class in the program. The container field does not affect program execution as long as no callee of the caller class has to be replaced. To replace callees referenced by the caller class, the program has to be changed as depicted in the right part of Figure 3.6. First, JAVADAPTOR creates a container class (see Figure 3.6, Lines 48 – 52) used to store instances of the new callee class (here of class `TempSensor_v2`). Second, our tool assigns the up-to-date counterpart of an outdated object (such as referenced by field `ts` in Figure 3.6) to an instance of the container. The container instance containing the up-to-date object is then assigned to field `cont` within the caller class (here class `TempDisplay`). Third, the tool redirects all accesses of the old callee instance to the updated callee instance located in the container (see Figure 3.6, Lines 36 – 38 and Lines 43 – 44), i.e., the tool redefines all method bodies in which the old callee instance is accessed and swaps the resulting method bodies via Java HotSwap (for more details see [Sch10]). Note that we, for clarity reasons, will remove the necessary downcasts to the specific container type (as shown in Lines 37 and 43 of Figure 3.6) from the following code examples.



```
 1 class TempDisplay {
 2   TempSensor ts;
 3   IContainer cont;
 4   ...
 5   TempSensor getSensor() {
 6     return ts;
 7   }
 8
 9   void setSensor(TempSensor ts) {
10     this.ts = ts;
11   }
12 }
```

```
13 class TempDisplay {
14   TempSensor ts;
15   IContainer cont;
16   ...
17   TempSensor getSensor() {
18     return new Proxy(cont.ts);
19   }
20
21   void setSensor(TempSensor ts) {
22     cont.ts = ((Proxy)ts).update;
23   }
24 }
```

```
25 class Proxy extends TempSensor {
26   TempSensor_v2 update;
27   ...
28 }
```

Figure 3.7: Schema-preservation through proxies.

Proxies. The basic container approach described above is sufficient in many cases. However, it fails when the caller class to be updated contains methods whose parameters or returned objects are of type of the old callee class (such as shown in Figure 3.7, Line 5 and 9). One workaround would be to replace the caller class as well. But, this strategy may result in additional class replacements, which at the worst require to essentially replace all classes of the system and thus let our DSU approach become inefficient. In order to avoid cascading class replacements, we extend our approach by proxies (see Figure 3.7). Caller updates work in the same manner as described above. Only difference is, that, in addition to the container class a proxy class is generated.

The idea of proxies is to guide objects of an updated callee class through the caller methods that require or return objects of type of the old callee class. The usage of proxies is exemplified on the basis of method `getSensor` of class `TempDisplay` which returns an instance of callee class `TempSensor` (Line 6). After replacing callee class `TempSensor` by class TempSensor_v2, method getSensor has to return an instance of the new callee class, which is not possible because TempSensor and TempSensor_v2 are not type compatible. To achieve type compatibility, we wrap the instance of TempSensor_v2 with an instance of class Proxy (Line 18). Since the proxy extends class TempSensor, it can be returned by method getSensor. In order to use the returned object wrapped by the proxy at receiver side (i.e., within the class that called method getSensor) the object is unwrapped. That is, the proxy is only used to guide instances of the new callee class through type-incompatible methods. The receiver will finally work with the new callee object and not with the proxy object. How to propagate instances of the updated callee class back to the caller (more precisely to the container) is exemplarily shown in Figure 3.7 (Line 22). Before method setSensor is called, its parameter (i.e, an instance of TempSensor_v2) is wrapped by a proxy. In order to unwrap and use the received instance of class TempSensor_v2, proxy ts must be cast to type Proxy.

3.6 Concurrent Updates of Multiple Classes

So far, we described the mechanisms and concepts of JavAdaptor on the basis of the very simple weather station example given in Section 3.1. This example only consists of one single class update and the corresponding caller-side update. However, JavAdaptor must not only allow the developer to update a single class but multiple classes in one step, which is essential to flexibly update complex real-world applications. On the one hand, this is because updates of real-world applications normally span many different classes. On the other hand, concurrent updates of multiple classes is essential for inheritance hierarchy updates, because superclass updates implicitly require to update and reload corresponding subclasses, too.

Figure 3.8 sketches how JavAdaptor handles concurrent updates of multiple classes. At first, JavAdaptor reloads all classes with changed schemas (as described in Section 3.4). Afterwards, it identifies all classes (callers) with references to the classes to be reloaded (see Section 3.5.1). This information is gained in one atomic step for efficiency reasons. That is, having an overview about all changes required to update the running program allows us to create possible containers and proxies in one single step. In addition, we only have to touch each class one-time in order to modify its bytecode. However, in the next two steps, JavAdaptor creates the new callee instances and maps the state (as we described in Section 3.5.2). If this is done, JavAdaptor updates all references conform to the workflow described in Sections 3.5.3 and 3.5.4. Since we already gained information about all dependencies between callers and callees, this can be efficiently done in one atomic step, too. In the last update step, we update all modified and hotswapable classes at once using Java HotSwap. This includes not only all callers of reloaded classes, but also classes which are explicitly changed by the developer.

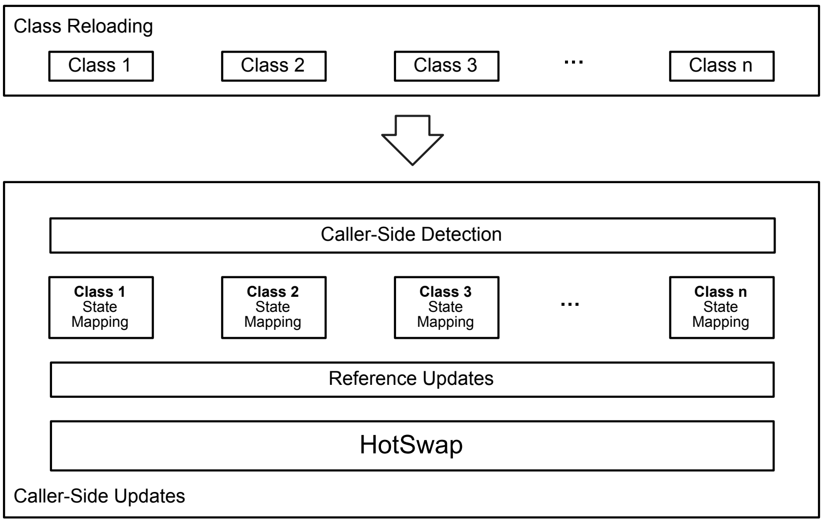


Figure 3.8: Concurrent multiple class updates.

3.7 Special Cases

After we described how JAVADAPTOR handles concurrent updates of multiple classes, one can possibly imagine that JAVADAPTOR is capable to support a wide range of complex update scenarios. In order to further confirm the capabilities of JAVADAPTOR, we explain how the tool supports special cases such as dynamic updates of inheritance hierarchies and nested classes. As we will reveal in the following, support for these scenarios can be entirely traced back to the already described concepts.

3.7.1 Inheritance Hierarchies

Inheritance is one of the basic principles of the object-oriented paradigm and thus must be considered by a DSU approach aiming at flexible dynamic software updates. Therefore, we go on describing how JAVADAPTOR supports program updates which change inheritance hierarchies (for deeper insights into our solution see [Gre10a]).

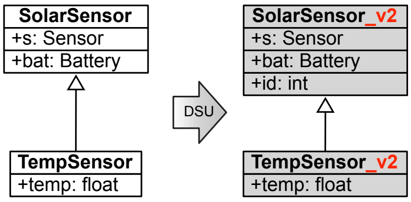


Figure 3.9: Implicit inheritance hierarchy updates.

Implicit Updates. Let us remind that the basic concept to support class schema changes with JavAdaptor is to load the updated class under a new name into the JVM, which works as described above in case the reloaded class is no superclass. That is, as a result of our class reloading approach, reloading superclasses such as class `SolarSensor` depicted in Figure 3.9 requires to reload the classes downwards the inheritance hierarchy (here class `TempSensor`) as well. This is due to the fact, that the subclasses have to extend the new superclass version in order to get access to the changes made upwards the inheritance hierarchy.

Explicit Changes. Explicit changes to the inheritance hierarchy are triggered through changes to the optional parts (in brackets) of the following language constructs:

$$\texttt{\textbf{class} X [\textbf{extends} Y] [\textbf{implements} IZ, ...]} \tag{3.1}$$

$$\texttt{\textbf{abstract class} X [\textbf{extends} Y] [\textbf{implements} IZ, ...]} \tag{3.2}$$

$$\texttt{\textbf{interface} IX [\textbf{extends} IY, IZ, ...]} \tag{3.3}$$

The goal of course is to support any kind of changes to the language constructs declared above. Furthermore, eligible solutions must be conform to the already introduced concepts. Fortunately, explicit inheritance hierarchy changes can be handled the same way as every other class schema affecting update. We just reload the class in order to change its inheritance hierarchy and implicitly update possible subclasses (see Figure 3.10).

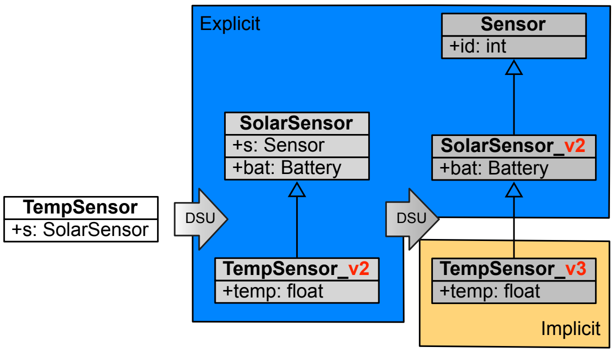
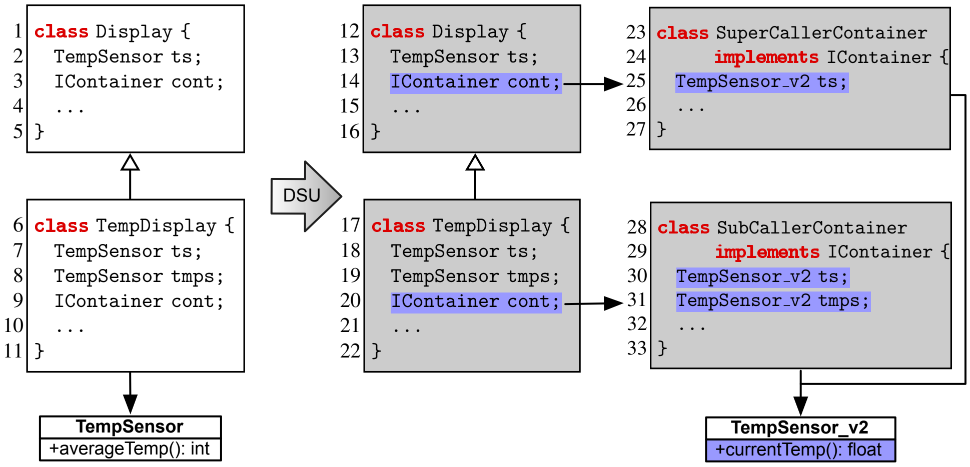


Figure 3.10: Explicit inheritance hierarchy changes.

Inheritance and State Mapping. As we described above, the reloading of classes with modified inheritance hierarchies conforms to the basic concepts of JavAdaptor, which is true for the state mappings as well. That is, one-to-one mappings and mappings with removed fields are automated by JavAdaptor. Newly added fields must be initialized based on user input. Moving and changing fields requires user input as well.



```
class Display {
  TempSensor ts;
  IContainer cont;
  ...
}
```

```
class TempDisplay {
  TempSensor ts;
  TempSensor tmps;
  IContainer cont;
  ...
}
```

```
class Display {
  TempSensor ts;
  IContainer cont;
  ...
}
```

```
class TempDisplay {
  TempSensor ts;
  TempSensor tmps;
  IContainer cont;
  ...
}
```

```
class SuperCallerContainer
    implements IContainer {
  TempSensor_v2 ts;
  ...
}
```

```
class SubCallerContainer
    implements IContainer {
  TempSensor_v2 ts;
  TempSensor_v2 tmps;
  ...
}
```

Figure 3.11: Inheritance hierarchies and reference updates.

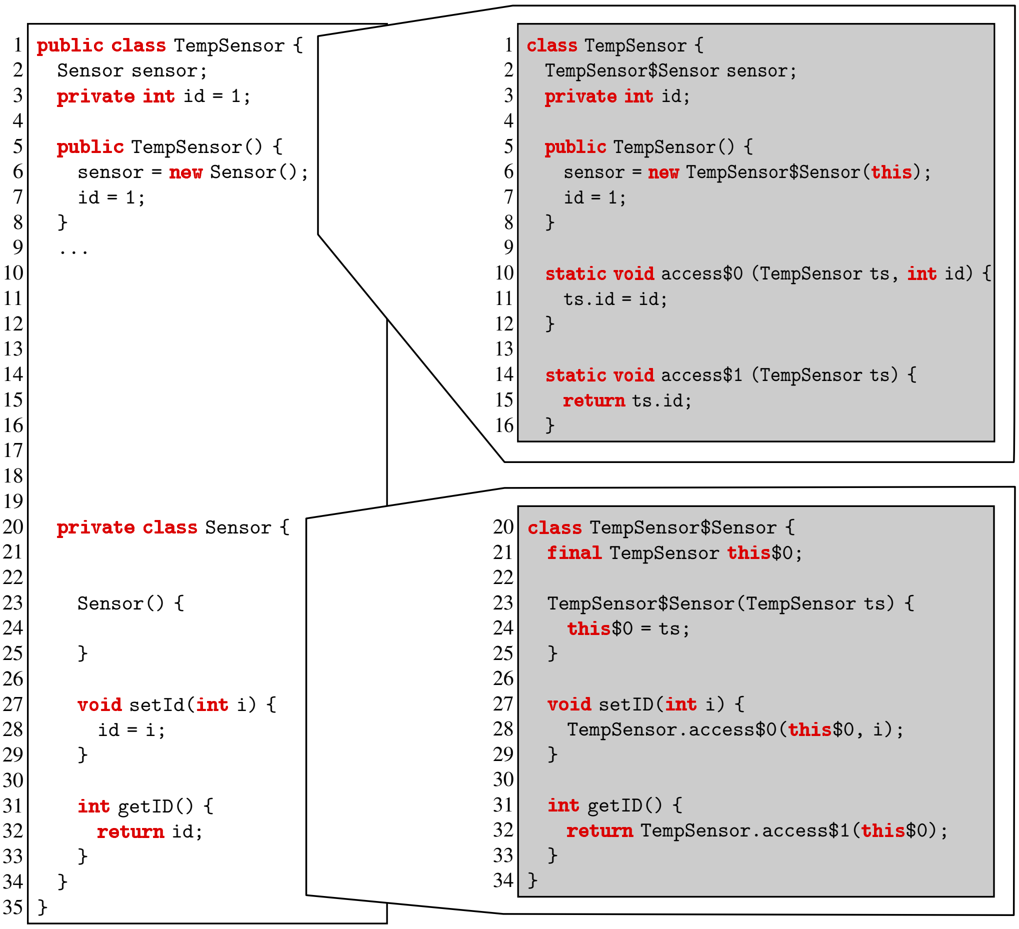
Inheritance and Reference Updates. However, so far we only discussed how we handle inheritance hierarchy modifications at callee side (i.e., of reloaded classes and their instances). What we have to consider in addition is how we must update the references to the class to be reloaded when the caller itself is bound to inheritance relationships. Figure 3.11 exemplifies such a scenario. Here, class `TempDisplay` inherits from class `Display` and thus owns three fields of class `TempSensor` scheduled for an update. In order to update the references, both classes get their own container class whereby each container only manages the fields its corresponding caller class declares, which is crucial, because putting all fields in one container would render field overriding (such as denoted by fields `ts` of our example) impossible.

3.7.2 Nested Classes

Object-oriented programming is all about objects and how they relate to each other. The Java language originally provides many constructs to describe complex relationships between different classes and objects. One well-known construct to describe dependencies between classes and their objects are *nested classes*. Gossling et al. [GJSB05] define nested classes as follows:

> *A* nested class *is any class whose declaration occurs within the body of another class or interface. A* top-level class *is a class that is not a nested class.*

According to this definition, a nested class is a class which depends on its declaring top-level class. The left part of Figure 3.12 shows how nested classes (here member class `Sensor`) are declared. In order to create and use an instance of nested class `Sensor`, the top-level class `TempSensor` must be vital, i.e., the top-level class must be loaded and an instance



```
 1 public class TempSensor {
 2   Sensor sensor;
 3   private int id = 1;
 4
 5   public TempSensor() {
 6     sensor = new Sensor();
 7     id = 1;
 8   }
 9   ...
10
11
12
13
14
15
16
17
18
19
20   private class Sensor {
21
22
23     Sensor() {
24
25     }
26
27     void setId(int i) {
28       id = i;
29     }
30
31     int getID() {
32       return id;
33     }
34   }
35 }
```

```
 1 class TempSensor {
 2   TempSensor$Sensor sensor;
 3   private int id;
 4
 5   public TempSensor() {
 6     sensor = new TempSensor$Sensor(this);
 7     id = 1;
 8   }
 9
10   static void access$0 (TempSensor ts, int id) {
11     ts.id = id;
12   }
13
14   static void access$1 (TempSensor ts) {
15     return ts.id;
16   }
```

```
20 class TempSensor$Sensor {
21   final TempSensor this$0;
22
23   TempSensor$Sensor(TempSensor ts) {
24     this$0 = ts;
25   }
26
27   void setID(int i) {
28     TempSensor.access$0(this$0, i);
29   }
30
31   int getID() {
32     return TempSensor.access$1(this$0);
33   }
34 }
```

Figure 3.12: Dynamic updates of nested classes.

of it must exist. In order to support dynamic software updates affecting nested classes, it is crucial to get to know how top-level class and depending nested class are represented at the bytecode level. As depicted at the right part of Figure 3.12, top-level class (here class `TempSensor`) and nested class (i.e., class `Sensor`) are translated into two different class files. The nested class is bound to the top-level class through `this` references (e.g., right part of Figure 3.12, Line 21), which is nothing different than a special kind of *composition*. One feature unique to nested classes is that they can access even private elements of the top-level class from within the nested class, which is realized through common getter and setter methods (such as method `access$0` depicted at the right part of Figure 3.12, Lines 10 – 12) just generated by the compiler.

Generally, top-level and nested class act as usual caller and callee on another. Due to this fact, we can apply updates affecting nested classes in the same manner as described in Section 3.4 and 3.5. That is, we basically reload the class whose schema has changed (no matter if top-level or nested class) and update the references at caller side (detailed descriptions of the approach can be found in [Sch10]).

To sum up, JAVADAPTOR allows us to flexibly change applications during their runtime. The update granularity can vary from minor changes (i.e., of single classes) to system-wide changes (i.e., of multiple classes). In addition, JAVADAPTOR will only update the changed classes and the corresponding caller classes. All other classes (except from subclasses of reloaded classes) remain untouched, which minimizes the influence of the update on the running program.

3.8 Summary

Within this chapter, we presented the conceptual core of JAVADAPTOR. We detailed how we hook JAVADAPTOR into the JPDA and thus integrate it with the Java runtime. In addition, we described how we change class schemas through class reloadings and update the referring program parts using Java HotSwap, containers, and proxies to hit the targeted high level of update flexibility. We further detailed how we keep the program state through state mappings. Last but not least, we exemplified on the basis of inheritance hierarchy changes and nested class updates that the concepts are generally applicable and cover even special update scenarios.

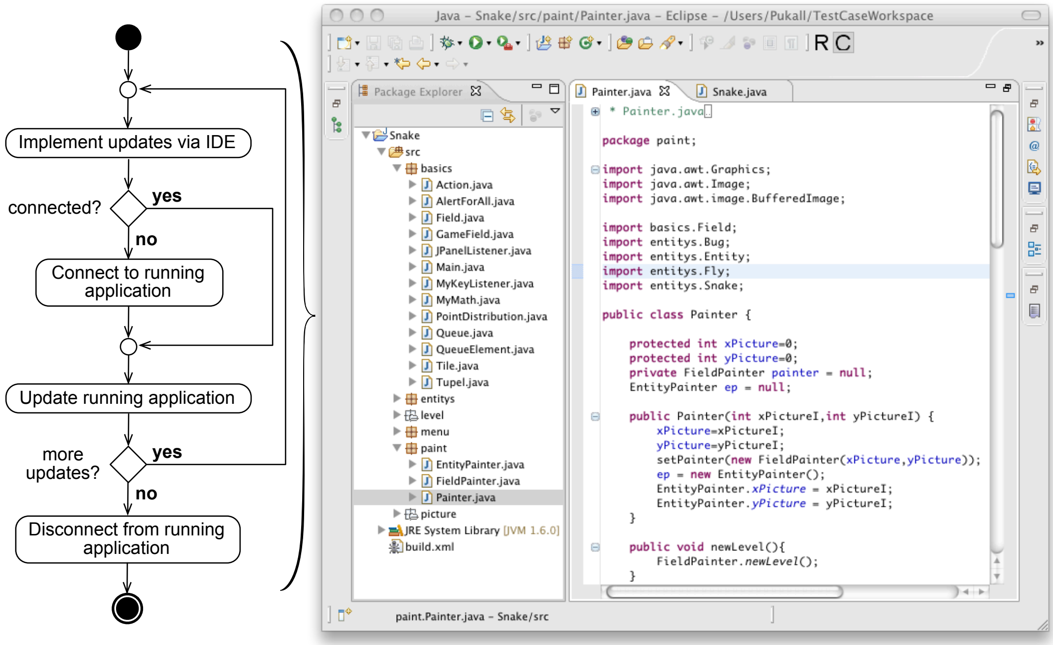
4 Implementation of JAVADAPTOR

This chapter shares material with the SPE'2011 paper "JavAdaptor – Flexible Runtime Updates of Java Applications" [PKC+12] and the ICSE'2011 paper "JavAdaptor: Unrestricted Dynamic Software Updates for Java" [PGS+11].

JAVADAPTOR accomplishes flexible dynamic software updates using class replacements (i.e., class reloadings) and updates of the references pointing to an outdated class. Through combining Java HotSwap, containers, and proxies those reference updates can be applied without replacing the referring classes (i.e., the caller side) as well. However, even if the concepts of JAVADAPTOR are comprehensible, their implementation is challenging for different reasons. One reason is the polymorphism of updates. Some updates require special treatment at implementation level (e.g., inheritance hierarchy changes require not only to update the changed class but all its subclasses as well). In order to demonstrate the general applicability of the concepts of JAVADAPTOR and in order to provide a tool which is usable in practice, our goal is to support as many kinds of updates as possible.

In this chapter, we detail the implementation of JAVADAPTOR. We do this for two major reasons. First, we want to get people, interested in our solution, the chance to reproduce the implementation. It may be a good starting point for their own research and may prevent them from making the same (time consuming) mistakes we made on our way to the current implementation. Second, a detailed implementation description reveals the preconditions that must be fulfilled by a language and its runtime environment to be able to host our DSU approach. This knowledge builds the base to apply the approach to other languages than Java. However, people not interested in the implementational details of JAVADAPTOR may skip this chapter or may only read the first paragraph of each section which briefly describes the current update step.

The structure of this chapter is as follows. For a better understanding of the implementational details, we first describe the usage and architecture of JAVADAPTOR. Then, we give an overview about the used external libraries and sketch the update steps JAVADAPTOR internally processes. Afterwards, we continue revealing the details of each of those update steps.



```
 * Painter.java

package paint;

import java.awt.Graphics;
import java.awt.Image;
import java.awt.image.BufferedImage;

import basics.Field;
import entitys.Bug;
import entitys.Entity;
import entitys.Fly;
import entitys.Snake;

public class Painter {

    protected int xPicture=0;
    protected int yPicture=0;
    private FieldPainter painter = null;
    EntityPainter ep = null;

    public Painter(int xPictureI,int yPictureI) {
        xPicture=xPictureI;
        yPicture=yPictureI;
        setPainter(new FieldPainter(xPicture,yPicture));
        ep = new EntityPainter();
        EntityPainter.xPicture = xPictureI;
        EntityPainter.yPicture = yPictureI;
    }

    public void newLevel(){
        FieldPainter.newLevel();
    }
```

Figure 4.1: Usage of JAVADAPTOR.

4.1 Tool Description

The current implementation of our tool comes as a plug-in which smoothly integrates into the *Eclipse*[1] IDE (conceptually JAVADAPTOR could be integrated into any other IDE or even used without an IDE).

The implementation of the required program updates conforms to the usual software development process, i.e., the developer implements the required functions using the Eclipse IDE and compiles the changed program sources. When the developer is done with the program changes and wants to apply the changes to the running application, she connects JAVADAPTOR with the JVM executing the application, pushes the update button of JAVADAPTOR, and the tool immediately applies the update. After the update, JAVADAPTOR can be disconnected from the application. The described process can be repeated as often as required (see Figure 4.1). A demo video showing JAVADAPTOR in action and demonstrating its usage, can be found on YouTube.[2]

From the architectural point of view, JAVADAPTOR establishes a connection, via the Java Debug Interface (JDI), to the Java Virtual Machine Tool Interface (JVMTI) of the targeted JVM (see Figure 4.2). Once the update is triggered, JAVADAPTOR prepares the classes

[1] http://www.eclipse.org/

[2] http://www.youtube.com/watch?v=jZm0hvlhC-E

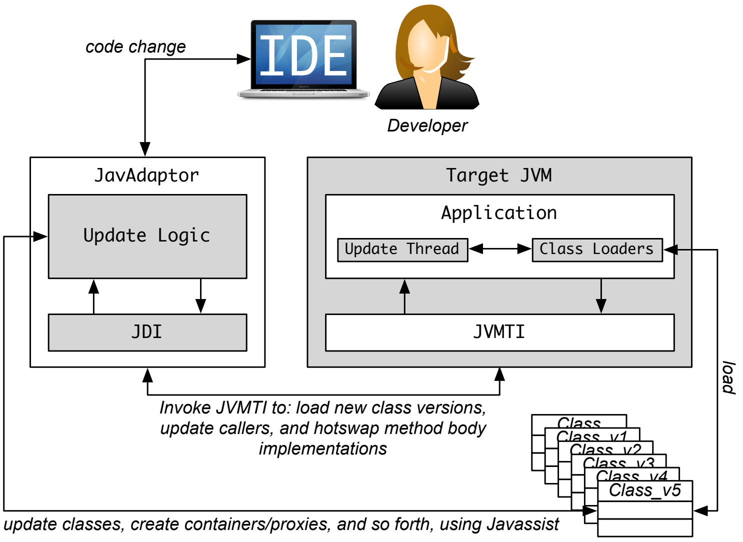

code change
IDE
Developer
JavAdaptor
Update Logic
JDI
Target JVM
Application
Update Thread
Class Loaders
JVMTI
load
Invoke JVMTI to: load new class versions, update callers, and hotswap method body implementations
Class
Class_v1
Class_v2
Class_v3
Class_v4
Class_v5
update classes, create containers/proxies, and so forth, using Javassist

Figure 4.2: Architecture of JAVADAPTOR.

changed within Eclipse and applies them using the functions provided by the JVMTI. In order to load and instantiate new class versions, a special update thread is added to the target application. This thread is only active when the running program is updated and, thus, causes no performance penalties during normal program execution.

4.2 Applied Libraries

As described above, JAVADAPTOR makes use of the JVMTI, which is standard for all major Java virtual machines. All requests of JAVADAPTOR to the JVMTI of the virtual machine executing the application to be updated are carried out via the Java Debug Interface, which is provided as an external library.

The second library vital to JAVADAPTOR is Javassist[3] (for further information see [CN03, Chi00, TSCI01]). Through Javassist, we carry out the bytecode modifications our update approach requires. Javassist is an easy to use bytecode modification tool, which allows us to modify the program's bytecode in any possible way and at different abstraction levels, i.e., at source level and at bytecode level. Using the source level API, bytecode modifications can be processed without any knowledge of the Java bytecode and its structure. However, even if the source level API is easy to use, it does not cover the whole bandwidth of possible bytecode modifications. In order to process bytecode modifications not possible with the source-level API, the developer can make use of the bytecode level API.

[3] http://www.csg.is.titech.ac.jp/~chiba/javassist/

4.3 Overview

JAVADAPTOR processes different steps in order to dynamically update an application (as depicted in Figure 4.3). First, JAVADAPTOR prepares (note that preparation in this context means to anticipate the possibility of an update and not to anticipate specific updates) the application for dynamic software udpates (see Action 1.1 of Figure 4.3). Afterwards, the application can be started. Every other JAVADAPTOR workflow step (i.e., Actions 1.2 – 1.5, Figure 4.3) belongs to the update of the running application.

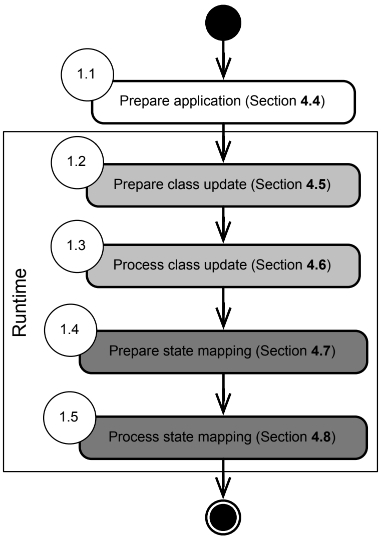

1.1
Prepare application (Section **4.4**)
Runtime
1.2
Prepare class update (Section **4.5**)
1.3
Process class update (Section **4.6**)
1.4
Prepare state mapping (Section **4.7**)
1.5
Process state mapping (Section **4.8**)

Figure 4.3: Overview of JAVADAPTOR.

For the update, JAVADAPTOR connects to the application and begins to gain information about what classes have changed, what classes must be reloaded, and what classes are due for reference updates because of the class reloadings (Action 1.2, Figure 4.3). Based on the gained information, JAVADAPTOR goes on to process the bytecode modifications required to update the program and finally loads/reloads the changed classes into the JVM (see Action 1.3, Figure 4.3).

If all class specific updates are applied to the JVM, JAVADAPTOR continues to map the state from outdated classes (and their instances) to their up-to-date counterparts (i.e., the updated classes and their instances) and updates the caller side (i.e., the referring classes and their instances), where with Action 1.4 (see Figure 4.3), we gain the required mapping information and with Action 1.5, we process the state mappings and the corresponding caller-side updates.

Within the following sections, we will detail each update step depicted in Figure 4.3.

4.4 Application Preparation

Before we can dynamically update an application using our concepts, we have to slightly change the application. In a nutshell, we have to add to each application class a container field and change the launch configuration such that the application is started with JVMTI/HotSwap enabled. Both actions, i.e., adding the container fields as well as enabling JVMTI including the HotSwap feature are central to our approach.

Figure 4.4 sketches the implementational details of the preparation process. The JVMTI including HotSwap is automatically loaded when debugging is enabled. That is, we have to change the original launch configuration in such way that the application is started in debug mode (Action 2.1 of Figure 4.4). The corresponding option we add to the launch configuration is:

agentlib:jdwp=transport=dt_socket, suspend=n, server = y, address="port number"

The option *jdwp* constitutes the loading of the debugging library (i.e., the JVMTI). For reasons of platform independence we choose option *dt_socket* (socket-based communication) for communication between JAVADAPTOR and the target JVM instead of *dt_shmem* (windows specific communication via shared memory) . Option *suspend=n* ensures that the application executes no matter whether JAVADAPTOR is connected to the target JVM or not. The *server=y* option lets the target JVM listen for incoming debug requests. With option *address*, we define the port used for communication. All other JVM and program options remain the same as defined in the original launch configuration.

JAVADAPTOR then gives the user the opportunity to decide whether to store the new launch configuration in an IDE specific format or as a script which allows the user to start the application independent from any IDE (Decision 2.2, Figure 4.4).

After launch configuration modification, JAVADAPTOR creates a properties file storing information about the project related to the application, such as project name, classpath, or location of the modified launch script (Action 2.4, Figure 4.4).

Next, JAVADAPTOR copies the binaries (class files, configuration files, required libraries, etc.) of the application into a new directory from where the program is executed later on (Action 2.5, Figure 4.4). Through Action 2.5, we decouple application development from the actually executed program and its binaries. That is, the developer (user) modifies the application according the usual static software development process (i.e., modifying the sources and compiling the program) while the changes do not influence the running program and its binaries until the user triggers the update. Another reason for organizing the application into development and execution location is that we want to keep JAVADAPTOR transparent to the developer. That is, if the developer explores the sources or class files of the project from within the IDE, she should not find any JAVADAPTOR related stuff, which

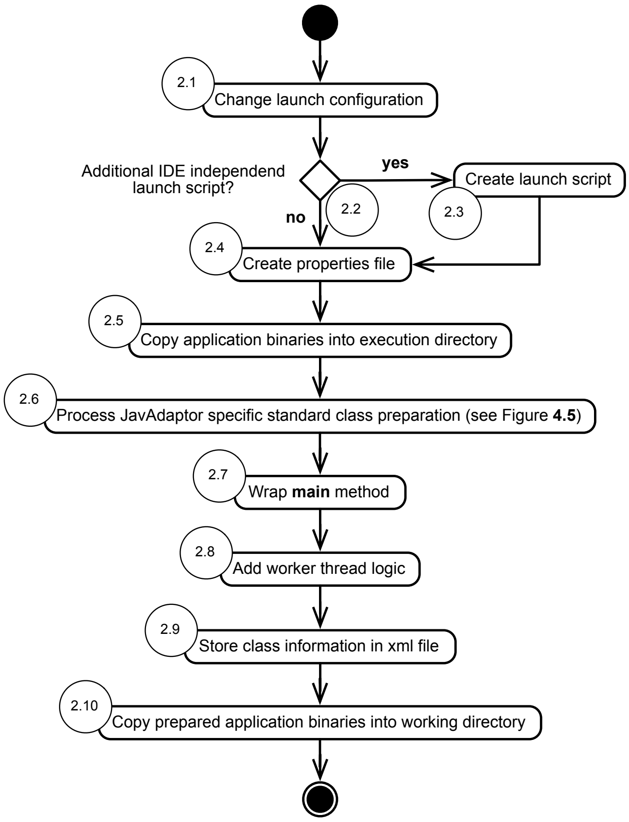

2.1
Change launch configuration
Additional IDE independend launch script?
yes
no
2.2
Create launch script
2.3
2.4
Create properties file
2.5
Copy application binaries into execution directory
2.6
Process JavAdaptor specific standard class preparation (see Figure **4.5**)
2.7
Wrap **main** method
2.8
Add worker thread logic
2.9
Store class information in xml file
2.10
Copy prepared application binaries into working directory

Figure 4.4: Prestart application preparation.

otherwise could confuse her. Last but not least, keeping the sources and binaries of the development location untouched, gives us a clean starting point for each program update no matter if first or consecutive update.

Having copied the application binaries into the execution location, we prepare all application class files stored in the execution location for our update approach (Action 2.6, Figure 4.4). The required class file preparation steps are depicted in Figure 4.5. First, we check whether the class is an interface or not (Decision 3.2, Figure 4.5). If the class is not an interface, we add the container field vital to our container concept and required to access the new version of a class due for an update (Action 3.3, Figure 4.5) to the application class.

The remaining application class modifications are not essential to our update approach, but improve the efficiency of the update process. At first, we remove possible `final` modifiers from static fields (Actions/Decision 3.4 – 3.6, Figure 4.5). This is because static fields are initialized at class load time and in case they are `final`, their values can only be changed through class reloading. The reason why we do not remove the `final` modifier from non-static fields is that the Java reflection API permits us to change the values of those fields even

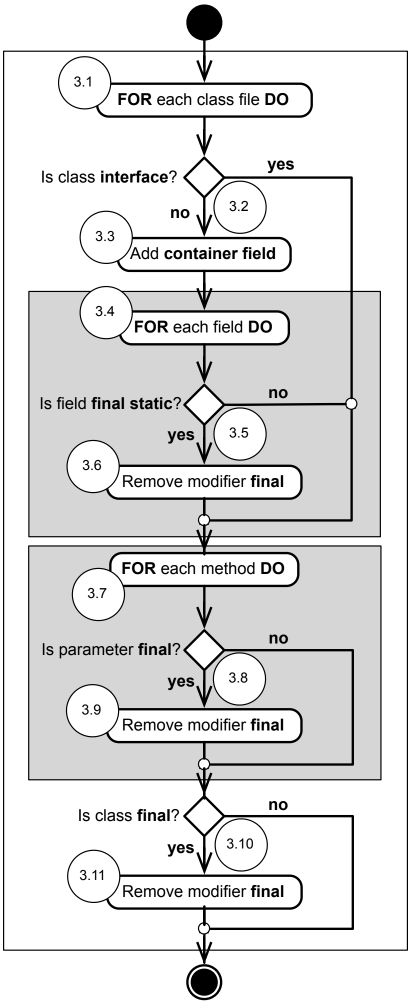


Figure 4.5: JAVADAPTOR specific standard class modifications (subpart of Figure 4.4).

after initialization. Next, JAVADAPTOR checks for each application class whether the class owns methods whose parameters are `final` and if true removes the `final` modifiers, which is done to efficiently implement our proxy concept (Actions/Decision 3.7 – 3.9). Last but not least, we remove possible `final` modifiers (Decision/Action 3.10 – 3.11) from the currently processed class, which is necessary to extend the class (note that our proxy concept requires to extend classes scheduled for reloadings).

Once the class files are prepared for our update approach, JAVADAPTOR creates a new `main` method that wraps the original `main` method of the application (Action 2.7, Figure 4.4). This is done because the JVM disallows the developer to redefine `main` methods via Java HotSwap. But, through creating a new `main` method, the JVM recognizes the original `main` method as an ordinary method and thus no longer declines to hotswap it. In a next step (see Action 2.8, Figure 4.4), JAVADAPTOR adds the worker thread required for our runtime updates to the newly created `main` method. Next (see Action 2.9, Figure 4.4), we store all class information (such as class name, superclasses, implemented interfaces, class version, etc.) gained during prestart phase. In a final step, we copy the prepared application binaries of the execution location into another location called working directory (Action 2.10, Figure 4.4). All class file modifications specific to our update approach will be processed in the working directory and not in the execution directory. Doing so, no intermediate results of our update process will be stored in the execution directory which otherwise could have side effects on the running application.

With finishing the last application preparation step, the application is prepared for dynamic software updates and could be started immediately through executing the changed launch configuration.

4.5 Class Update Preparation

Once the prepared application is started, the developer can modify the program's source code and apply the changes to the running application via JAVADAPTOR. To dynamically update the application, JAVADAPTOR connects to the Java Virtual Machine Tool Interface (JVMTI) of the JVM executing the application due for an update (such as we already described in Section 4.1). If the connection between JAVADAPTOR and the running application is established, JAVADAPTOR gains information about the program parts that must be updated. That is, the tool identifies the class files changed by the developer, identifies the classes that have to be reloaded because of changed class schemas, and looks for references to those classes. The information will be used later on to rename the classes that must be reloaded and to update their referring classes using HotSwap, containers, and/or proxies.

In the following sections, we will reveal the implementational details of the class update preparation process.

4.5.1 Identification of Changed Classes

When the developer triggers the update, JAVADAPTOR first identifies what classes have changed. Changed classes are those that are explicitly changed by the developer and those that the Java compiler recompiled because they depend on the explicitly changed classes. The goal of JAVADAPTOR is to apply these changed classes to the running application on the basis of the concepts described in Chapter 3 .

To simplify matters, we currently ask Eclipse for the changed classes (see Action 4.1, Figure 4.6). We point out that for reasons of platform independence, we could easily identify the changed classes on the basis of timestamps, checksums, and so forth. In the next

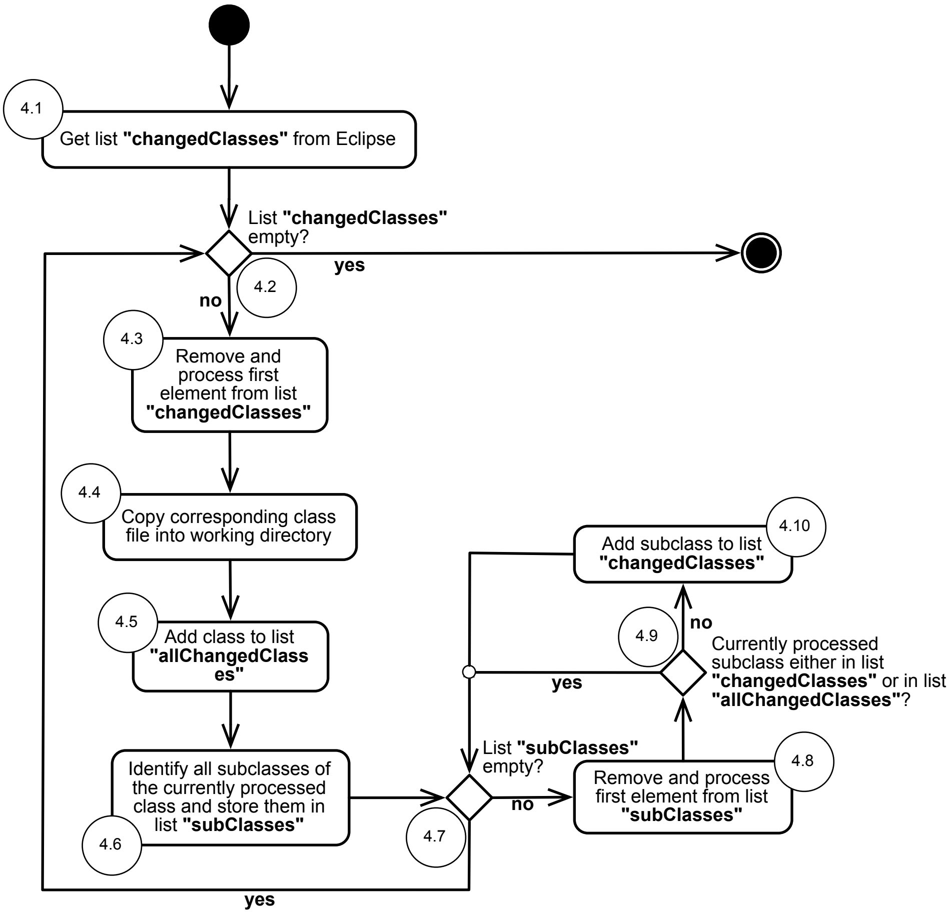

4.1
Get list "changedClasses" from Eclipse
List "changedClasses" empty?
4.2
yes
no
4.3
Remove and process first element from list "changedClasses"
4.4
Copy corresponding class file into working directory
4.5
Add class to list "allChangedClass es"
4.6
Identify all subclasses of the currently processed class and store them in list "subClasses"
4.7
List "subClasses" empty?
no
yes
4.8
Remove and process first element from list "subClasses"
4.9
Currently processed subclass either in list "changedClasses" or in list "allChangedClasses"?
yes
no
4.10
Add subclass to list "changedClasses"

Figure 4.6: Identification of changed classes.

step (Action 4.2, Figure 4.6), we check whether the requested list of changed classes is empty or not. If not empty, we remove the first element of this list (Action 4.3, Figure 4.6) and copy the class file related to the list element to the working directory (Action 4.4, Figure 4.6).

However, we do not only have to identify the classes explicitly and implicitly changed by the developer. Additionally, we must identify all subclasses of those changed classes (see Actions/Decisions 4.6 – 4.10), which (as we already described in Section 3.7.1) is due to the fact that, if the changed classes must be reloaded (because their schema has changed), we have to reload the subclasses as well. In this context, we characterize every class as a subclass if it `extends` and/or `implements` a changed class. Once JAVADAPTOR identified all subclasses, it has all information required to proceed with the next update step, i.e., the identification of classes with changed schemas.

4.5.2 Identification of Classes with Changed Schemas

After obtaining the changed classes and their subclasses, we identify the classes whose schema has changed. Of course, we could update even classes with unchanged schemas through class reloading, but updating those classes via Java HotSwap is far more efficient. This is because with HotSwap, class versions remain the same and as a result, no state mappings and class reference updates through containers and proxies are required.

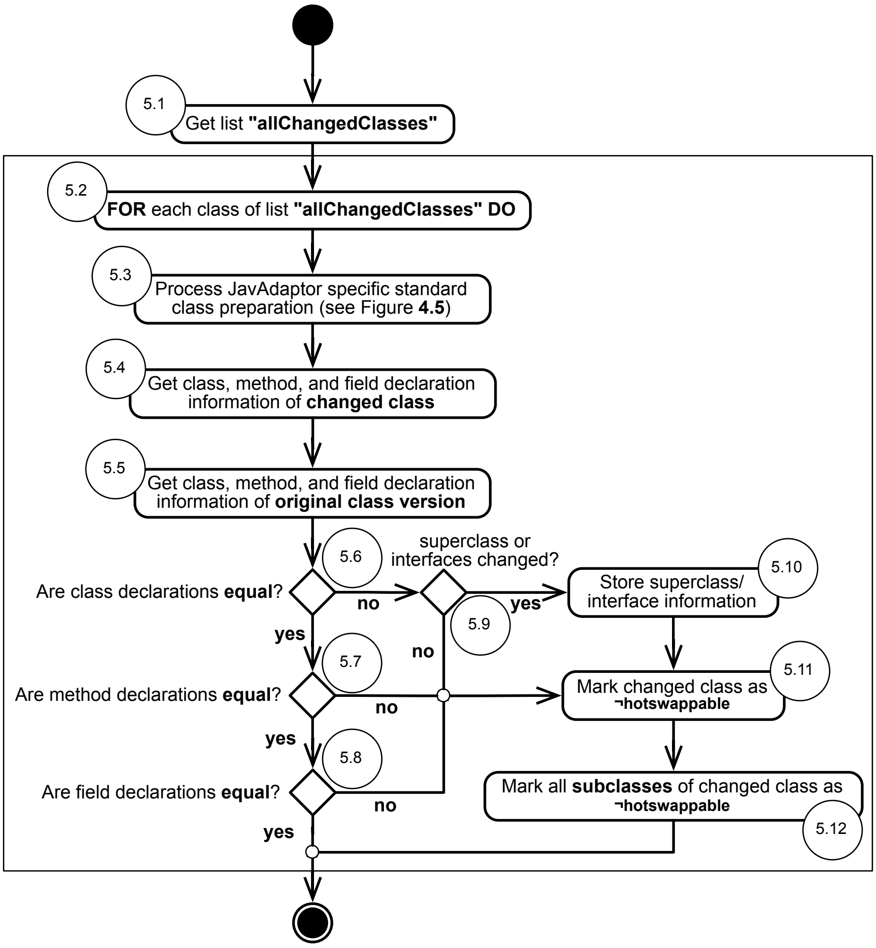

5.1
Get list **"allChangedClasses"**
5.2
FOR each class of list **"allChangedClasses" DO**
5.3
Process JavAdaptor specific standard class preparation (see Figure **4.5**)
5.4
Get class, method, and field declaration information of **changed class**
5.5
Get class, method, and field declaration information of **original class version**
5.6
Are class declarations **equal**?
no
superclass or interfaces changed?
5.9
yes
no
5.10
Store superclass/ interface information
yes
5.7
Are method declarations **equal**?
no
5.11
Mark changed class as **¬hotswappable**
yes
5.8
Are field declarations **equal**?
no
Mark all **subclasses** of changed class as **¬hotswappable**
5.12
yes

Figure 4.7: Test class for schema change.

However, before we start identifying classes with changed schemas, we prepare the changed classes for our update approach (Action 5.3, Figure 4.7), i.e., we add the container fields and possibly remove `final` modifiers (as previously described in Section 4.4). This must be done, because otherwise we could not update those classes with later program updates. However, there is also another reason for preparation. That is, in the following, we check whether changed class and original class share the same class schema. In earlier update process steps we already prepared the original class for our update approach. If we now compare prepared original class and unprepared changed class, JAVADAPTOR would detect a schema change, because the unprepared changed class misses the container field and may own some `final` modifiers not present in the prepared original class. That is, JAVADAPTOR would schedule the class for an update through class reloading, even if the changes to the class did not change the class schema. This is inefficient and thus must be avoided.

Ones prepared, we retrieve schema information of the changed class and the original class (Actions 5.4 and 5.5, Figure 4.7) and check whether they share the same class schema or not, i.e., we test whether the changed class must be reloaded or could be updated using Java HotSwap.

But, what elements of a class determine the class schema? The basic element of every class is the *class declaration*, which constitutes the general class schema. According the *Java Language Specification*, a class declaration is of the following form [GJSB05]:

$$\begin{aligned}
\text{ClassDeclarations} &= \text{Modifiers}_{\text{opt}}\ \text{class/interface/enum Identifier TypeParameters}_{\text{opt}} \\
&\ \text{Super}_{\text{opt}}\ \text{Interfaces}_{\text{opt}}\ \text{Body} \\
\text{Modifiers} &= \{\text{Modifier, Modifiers Modifier}\} \\
\text{Modifier} &= \{\text{Annotation, public, protected, private, abstract, static, final,} \\
&\ \ \text{strictfp}\}
\end{aligned}$$

Mandatory to every class declaration are keywords *class/interface/enum*, the class name (*Identifier*), and the body of the class (*Body*). Optional elements are the modifiers (*Modifiers*), *TypeParameters*, superclass (*Super*), and implemented interfaces (*Interfaces*). Every change to the class declaration changes the class schema.

Other elements relevant to the class schema are *method declarations*, which according the Java Language Specification have the following form [GJSB05]:

$$\begin{aligned}
\text{MethodDeclaration} &= \text{MethodHeader MethodBody} \\
\text{MethodHeader} &= \text{MethodModifiers}_{\text{opt}}\ \text{TypeParameters}_{\text{opt}}\ \text{ResultType} \\
&\ \text{MethodDeclarator Throws}_{\text{opt}} \\
\text{MethodBody} &= \{\{\ \},\ ;\} \\
\text{ResultType} &= \{\text{Type, void}\} \\
\text{MethodDeclarator} &= \text{Identifier (FormalParameterList}_{\text{opt}}) \\
\text{MethodModifiers} &= \{\text{MethodModifier, MethodModifiers, MethodModifier}\} \\
\text{MethodModifier} &= \{\text{Annotation, public, protected, private, abstract, static,} \\
&\ \ \text{final, synchronized, native, strictfp}\}
\end{aligned}$$

Here, the return type (*ResultType*) and *MethodDeclarator* are mandatory to every method declaration. Like with class declarations, changes to the elements of a method declaration change the class schema. Interestingly, Java HotSwap permits to add or remove element *throws* for which reason we do not consider a class schema as changed if *throws* was added/removed.

Finally, declarations of instance or class fields are vital to the class schema as well. Their form is specified as followed [GJSB05]:

$$\begin{aligned}
\text{FieldDeclaration} &= \text{FieldModifiers}_{\text{opt}}\ \text{Type VariableDeclarators ;}\\
\text{VariableDeclarators} &= \{\text{VariableDeclarator},\\
&\quad\ \{\text{VariableDeclarators, VariableDeclarator}\}\}\\
\text{VariableDeclarator} &= \{\text{VariableDeclaratorId},\\
&\quad\ \text{VariableDeclaratorId} = \text{VariableInitializer}\}\\
\text{VariableDeclaratorId} &= \{\text{Identifier, VariableDeclaratorId [ ]}\}\\
\text{VariableInitializer} &= \{\text{Expression, ArrayInitializer}\}\\
\text{FieldModifiers} &= \{\text{FieldModifier}, \{\text{FieldModifiers FieldModifier}\}\}\\
\text{FieldModifier} &= \{\text{Annotation, public, protected, private, static,}\\
&\quad\ \text{final, transient, volatile}\}
\end{aligned}$$

Only optional elements of field declarations are field modifiers (*FieldModifiers*). All modifications of field declarations will cause class schema changes. Exceptions are variable initializers (*VariableInitializer*), which are automatically moved by the compiler into constructors or static blocks and thus could be changed and updated at runtime via Java HotSwap.

After JavAdaptor retrieved the class, method, and field declaration information for both, original class and changed class, it compares the declarations pair-wise and checks whether they are equal or not. JavAdaptor first compares the class declarations (Decision 5.6, Figure 4.7), goes on checking the method declarations (Decision 5.7, Figure 4.7), and finally analyzes the field declarations (Decision 5.8, Figure 4.7).

With the first check that indicates a class schema change, we are allowed to terminate the test and mark the changed class and all its subclasses as ¬*hotswappable* (Actions 5.11 and 5.12, Figure 4.7). In case the class declaration has changed, JavAdaptor checks whether the changed class extends a different superclass or implements different interfaces than the original class and if true stores this information (Decision/Action 5.9 and 5.10, Figure 4.7). The information will be used in later update steps such as the state mapping.

If we know whether the class schema has changed or not, we continue to analyze the next class's schema.

4.5.3 Class Reference Identification

With the previous update step, we found out what classes can be updated through Java HotSwap and what classes must be reloaded under a fresh name to update them. With this update process step, we identify all references to classes that must be reloaded. This is necessary, because we must update the references such that they point to the new class versions to let the new class versions become part of program execution (see Section 3.5).

To identify the references to classes scheduled for reloading, we have to go through all application classes (Action 6.2, Figure 4.8) and check whether they reference classes with changed schemas or not. But, before we parse a class for class references due for an update, we check if the processed class itself is going to be replaced (reloaded) because its schema has changed (note that those classes will be handled different than classes with unchanged schemas). If this is not the case (i.e., if the class is not marked as ¬*hotswappable*, see Decision 6.3 of Figure 4.8), JAVADAPTOR retrieves all class references occurring in the class via Javassist method `getRefClasses` (Action 6.4, Figure 4.8) and analyzes each element of the resulting list (Action 6.5, Figure 4.8). First, JAVADAPTOR checks whether the referenced class is marked as ¬*hotswappable* (Action 6.6, Figure 4.8) or not. In case it is ¬*hotswappable*, we put the currently analyzed application class in list *allChangedClasses* if not already done (Decision/Action 6.7 – 6.8, Figure 4.8).

However, generally knowing that the application class's references must be updated is not enough. In addition, we must figure out if the class only refers to short-lived instances of ¬*hotswappable* classes (those references could be easily updated through method body redefinitions) or if it needs containers (to update references to long-living instances of ¬*hotswappable* classes) and/or proxies in addition. Classes that only hold references to short-lived instances of ¬*hotswappable* classes do not require containers or proxies.

In order to get to know if we need proxies (remember that in order to keep the method owner's schema, we use proxies to guide up-to-date objects through methods that expect objects of outdated class versions, see Section 3.5.4), we check the application class's method declarations for references to the currently considered class and related array classes (Decision 6.10, Figure 4.8). In case JAVADAPTOR finds references, it may create a proxy for the referenced (array) class (Decision/Action 6.11 – 6.12, Figure 4.8).

To check whether the processed application class needs a container, JAVADAPTOR tests if any of the declared fields are of type of the referenced class or related array classes. If so, the tool may create a container and puts all fields of (array) type of the referenced class into this container (Actions/Decisions 6.13 – 6.17, Figure 4.8).

Once we identified all references to classes that must be reloaded, we are done with the class update preparation step and can immediately process the class updates on the basis of the up-to-now gained information.

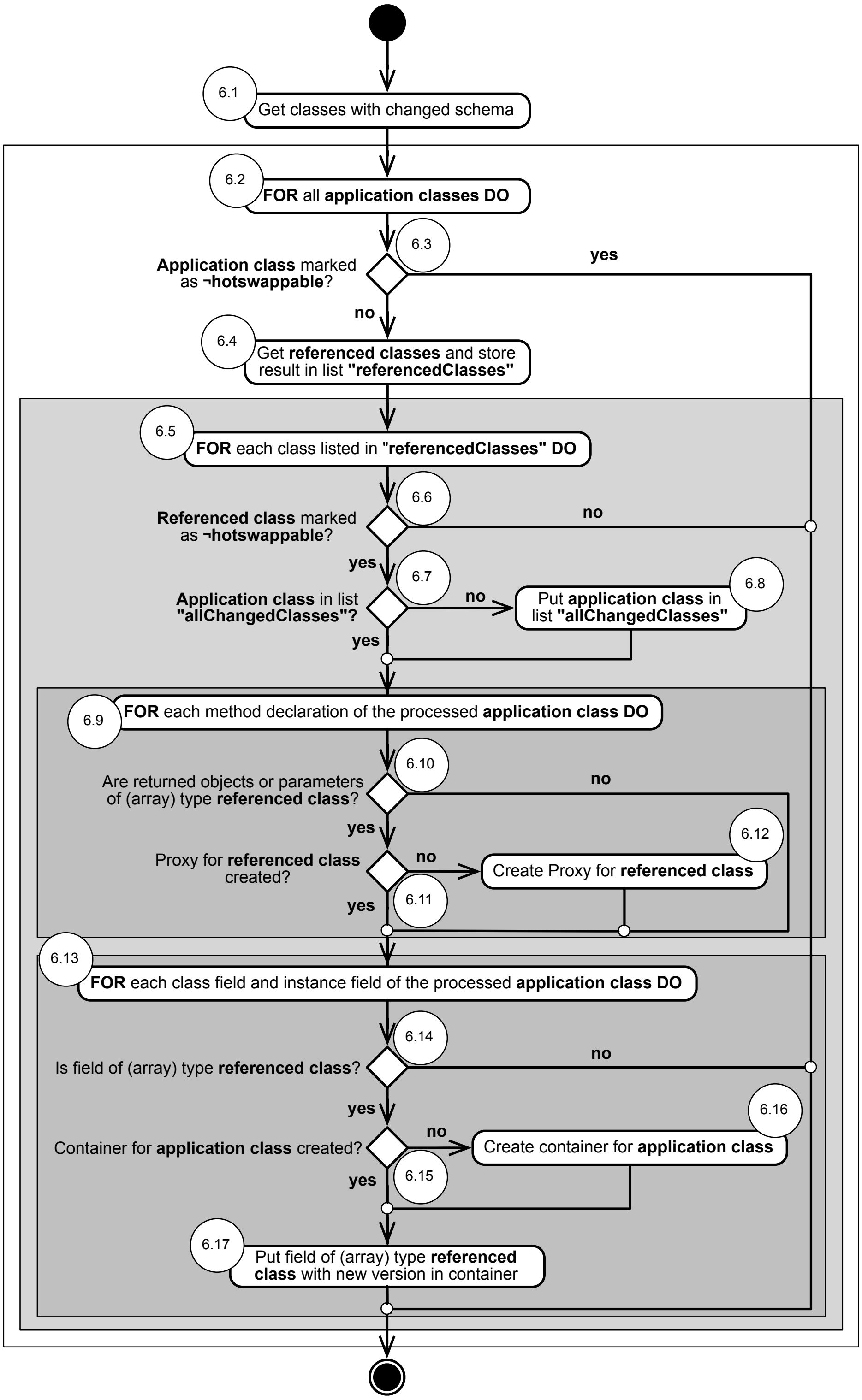

6.1
Get classes with changed schema
6.2
FOR all application classes DO
6.3
Application class marked as ¬hotswappable?
yes
no
6.4
Get referenced classes and store result in list "referencedClasses"
6.5
FOR each class listed in "referencedClasses" DO
6.6
Referenced class marked as ¬hotswappable?
no
yes
6.7
Application class in list "allChangedClasses"?
no
6.8
Put application class in list "allChangedClasses"
yes
6.9
FOR each method declaration of the processed application class DO
6.10
Are returned objects or parameters of (array) type referenced class?
no
yes
6.12
Proxy for referenced class created?
no
Create Proxy for referenced class
6.11
yes
6.13
FOR each class field and instance field of the processed application class DO
6.14
Is field of (array) type referenced class?
no
yes
6.16
Container for application class created?
no
Create container for application class
6.15
yes
6.17
Put field of (array) type referenced class with new version in container

Figure 4.8: Identify class references.

4.6 Class Update Proceeding

In the previous update process steps, we found out what classes have changed, what classes must be reloaded, and what references we have to update in order to execute the reloaded classes. That is, we exactly know what classes we have to modify in order to update the running application with our DSU approach. Within the following sections, we describe how JAVADAPTOR modifies the bytecode of the classes to be updated. That is, we show how JAVADAPTOR *updates class, field, and method declarations*, *creates container and proxy classes*, and *updates method bodies*. Finally, we detail how JAVADAPTOR gets the modified class files into the JVM.

4.6.1 Declaration Updates

The first bytecode modification step aims at updating the class, field, and the method declarations of all classes to be updated, i.e., all classes listed in *allChangedClasses* (see Figure 4.9). For that purpose, JAVADAPTOR creates a Javassist `ClassMap` which contains all class names we must update. But, before JAVADAPTOR updates the class names and declarations using Javassist method `replaceClassName`, it checks whether the currently processed class is an interface or not (Decision 7.3, Figure 4.9). In the latter case, JAVADAPTOR buffers the method body definitions (note that interfaces only declare methods but never contain method body definitions and thus must be not considered) of the methods declared in the class and removes the method's body definitions (Actions 7.4 and 7.5, Figure 4.9). This is done for one important reason. Method `replaceClassName` would update even the class names occurring in the method bodies, which we have to prevent because those names must be updated in a different way. Through buffering the method body definitions and removing them from the currently processed class, we can apply method `replaceClassName` to the currently processed class while keeping the method bodies untouched.

Next (Decision 7.6, Figure 4.9), we figure out whether the currently processed class's schema has changed (i.e., the class is marked as ¬*hotswappable*) or not (i.e., the class is marked as *hotswappable*) .

¬*Hotswappable* Classes. In case the class is ¬*hotswappable* and must be reloaded with a new version, JAVADAPTOR applies the previously created `ClassMap`, which converts the class name and all occurrences of the class name to the *new version* (Action 7.7, Figure 4.9), such as illustrated in Figure 4.10 where with *Update 2* the name of class `TempDisplay` is converted to `TempDisplay_v2`. Afterwards, JAVADAPTOR retrieves all ¬*hotswappable* referenced (array) classes and updates the corresponding references to the *new version* (Action 7.10), which is also illustrated with *Update 2* of Figure 4.10. Furthermore, JAVADAPTOR must update all remaining class references to their *current version* (Action 7.12, Figure 4.9).

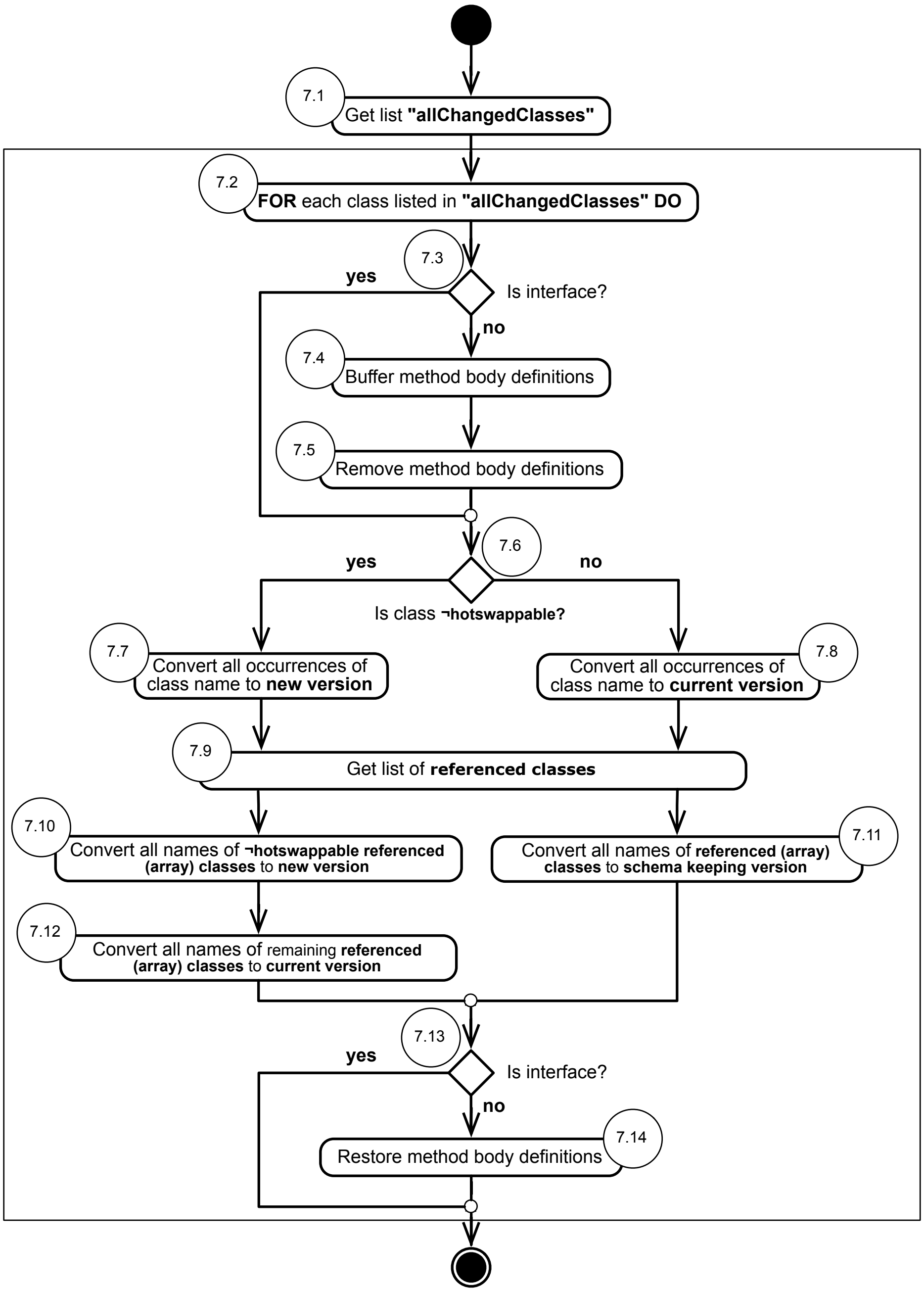
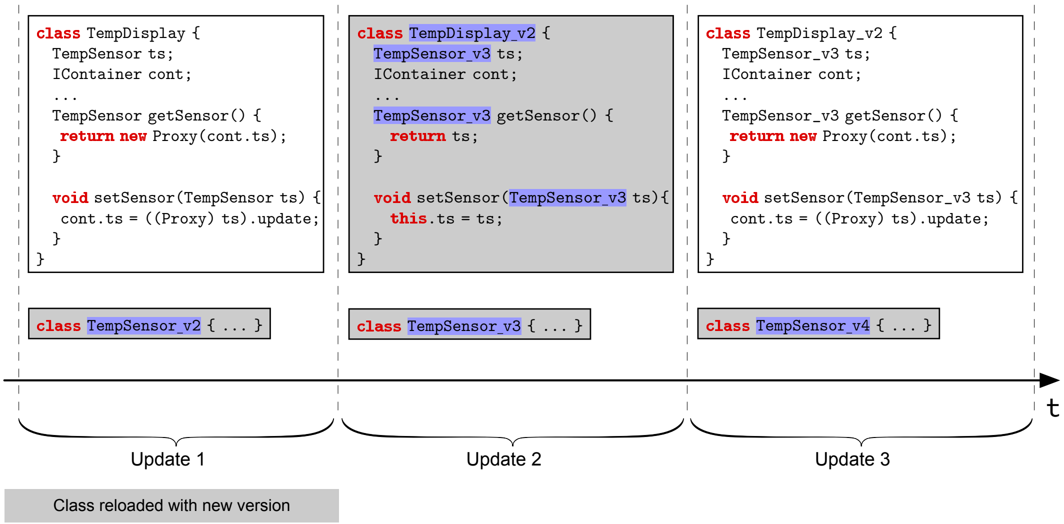

7.1
Get list "allChangedClasses"
7.2
FOR each class listed in "allChangedClasses" DO
7.3
yes
Is interface?
no
7.4
Buffer method body definitions
7.5
Remove method body definitions
7.6
yes
no
Is class ¬hotswappable?
7.7
Convert all occurrences of class name to new version
7.8
Convert all occurrences of class name to current version
7.9
Get list of referenced classes
7.10
Convert all names of ¬hotswappable referenced (array) classes to new version
7.11
Convert all names of referenced (array) classes to schema keeping version
7.12
Convert all names of remaining referenced (array) classes to current version
7.13
yes
Is interface?
no
7.14
Restore method body definitions

Figure 4.9: Update of class, field, and method declarations.



```
class TempDisplay {
  TempSensor ts;
  IContainer cont;
  ...
  TempSensor getSensor() {
   return new Proxy(cont.ts);
  }

  void setSensor(TempSensor ts) {
   cont.ts = ((Proxy) ts).update;
  }
}
```

```
class TempSensor_v2 { ... }
```

```
class TempDisplay_v2 {
  TempSensor_v3 ts;
  IContainer cont;
  ...
  TempSensor_v3 getSensor() {
    return ts;
  }

  void setSensor(TempSensor_v3 ts){
    this.ts = ts;
  }
}
```

```
class TempSensor_v3 { ... }
```

```
class TempDisplay_v2 {
  TempSensor_v3 ts;
  IContainer cont;
  ...
  TempSensor_v3 getSensor() {
   return new Proxy(cont.ts);
  }

  void setSensor(TempSensor_v3 ts) {
   cont.ts = ((Proxy) ts).update;
  }
}
```

```
class TempSensor_v4 { ... }
```

Figure 4.10: Class versioning.

This must be done, because we retrieve the class files to be updated from the application's development location (i.e., the eclipse project location) where the classes have their original names without any version add-on.

***HotSwappable* Classes.** *Hotswappable* classes must be handled in a different way than ¬*hotswappable* classes. Different from ¬*hotswappable* classes, we do not have to rename *hotswappable* classes. Another difference is, that in those classes we could not simply update the references to ¬*hotswappable* classes to new versions, because this would change the referring classes's schemas and thus render themselves ¬*hotswappable*. That is, in order to keep the classes *hotswappable*, we convert all references occurring in field or method declarations to versions, which are already part of the declarations of the currently loaded class (Action 7.11, Figure 4.9). This is exemplified by means of *Update 3* shown in Figure 4.10. Here, class `TempSensor` must be reloaded with a new version (i.e., with version `TempSensor_v4`). However, referring class `TempDisplay` was previously loaded with version `TempDisplay_v2`. Thus, in order to keep the schema of class `TempDisplay_v2`, we must not update the references to version `TempSensor_v4` but to version `TempSensor_v3` instead.

Once possible class, field, and method declarations are up-to-date, we restore the method body definitions. That is, we put the buffered definitions back into their origin class (Decision 7.13 and Action 7.14, Figure 4.9), which concludes the current update process step.

4.6.2 Container Creation Phase

In Section 3.5.4, we described how we update long-living references to classes to be reloaded using our container concept. This section reveals how JAVADAPTOR creates the container classes belonging to the concept.

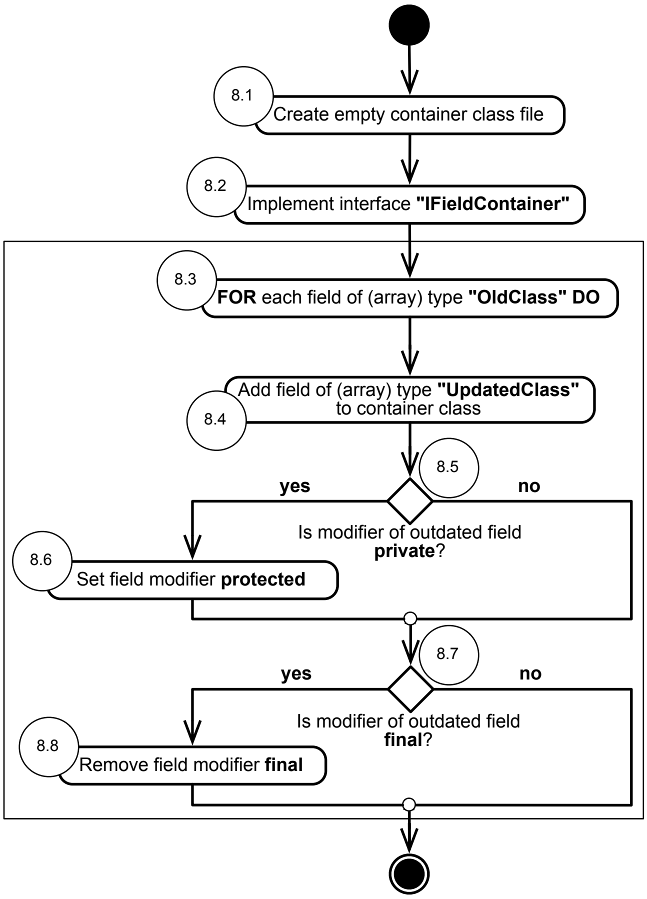

8.1
Create empty container class file
8.2
Implement interface **"IFieldContainer"**
8.3
FOR each field of (array) type **"OldClass" DO**
Add field of (array) type **"UpdatedClass"** to container class
8.4
8.5
yes
no
Is modifier of outdated field **private**?
8.6
Set field modifier **protected**
8.7
yes
no
Is modifier of outdated field **final**?
8.8
Remove field modifier **final**

Figure 4.11: Container class creation.

First, JAVADAPTOR creates an empty container class file (Action 8.1, Figure 4.11). Next, the tool lets the container class implement the interface `IFieldContainer` (Action 8.2, Figure 4.11), which is the standard type of every application classes's container field added during prestart phase (note that the container must implement this interface in order to be type compatible to those container fields, i.e., in order to allow us to assign container instances to those container fields). Then, JAVADAPTOR adds for each field of type of the outdated class a field of type of the corresponding up-to-date class version to the container (Action 8.4, Figure 4.11) and changes the field modifier from `private` to `protected` if necessary (Decision 8.5 and Action 8.6, Figure 4.11). This must be done in order to access those container fields from within the corresponding referring class. Finally, JAVADAPTOR removes possible `final` modifiers from the container fields, which is required to render subsequent state mappings possible (Decision 8.7 and Action 8.8, Figure 4.11).

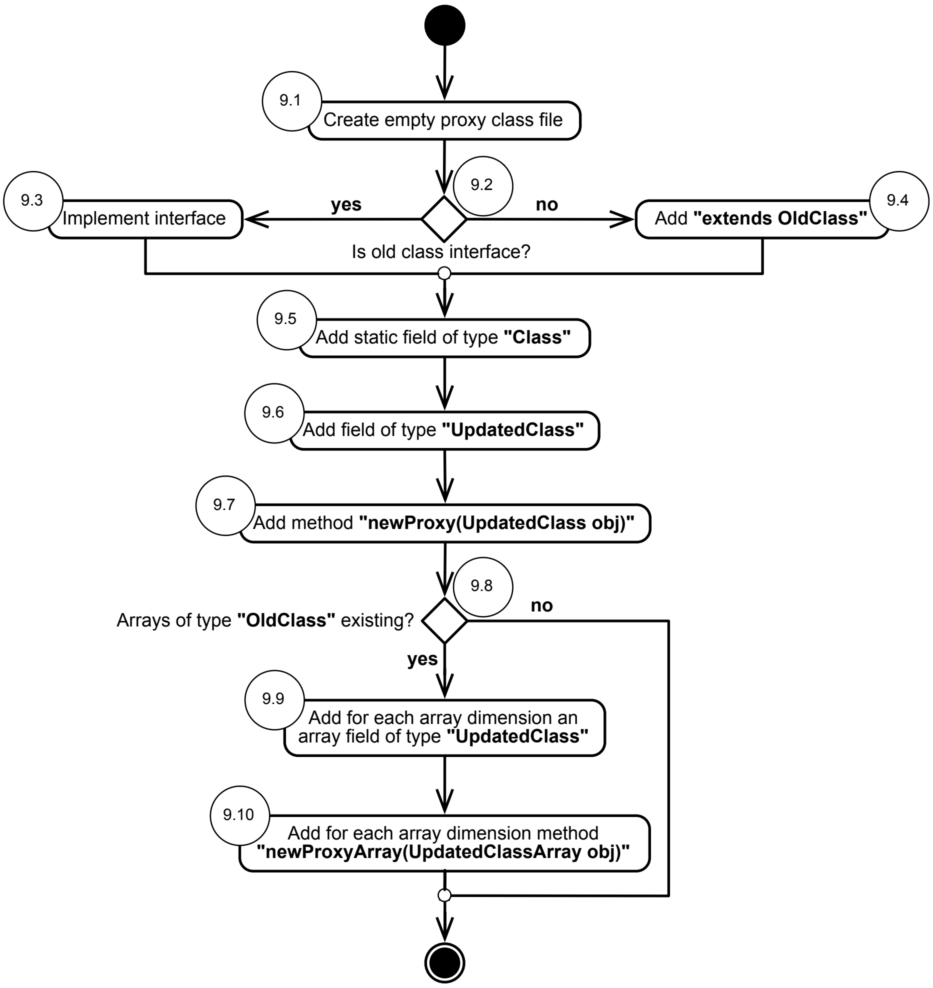

9.1
Create empty proxy class file
9.2
yes
no
Is old class interface?
9.3
Implement interface
9.4
Add "extends OldClass"
9.5
Add static field of type "Class"
9.6
Add field of type "UpdatedClass"
9.7
Add method "newProxy(UpdatedClass obj)"
9.8
Arrays of type "OldClass" existing?
no
yes
9.9
Add for each array dimension an array field of type "UpdatedClass"
9.10
Add for each array dimension method "newProxyArray(UpdatedClassArray obj)"

Figure 4.12: Proxy class creation.

4.6.3 Proxy Creation Phase

Within previous update process steps, we figured out which ¬*hotswappable* classes require proxies (remember that ¬*hotswappable* classes need a proxy if at least one *hotswappable* class owns a method which returns and/or takes objects of type of the ¬*hotswappable* class). With this update process step, we create the proxies for those classes.

We start with creating an empty proxy class file (Action 9.1, Figure 4.12). Next, we check whether the class for which we create the proxy is an interface or not (Decision 9.2, Figure 4.12). If it is an interface, we let the proxy class implement this interface (Action 9.3, Figure 4.12). Otherwise, we extend the `OldClass` class (Action 9.4, Figure 4.12). Then, JAVADAPTOR adds a static field of type `Class` to the proxy, which is internally used later on to create the proxy instances (Action 9.5, Figure 4.12).

Now, we go on adding the fields of type `UpdatedClass` whose instances we want to guide through outdated methods using the proxy (Action 9.6, Figure 4.12). Furthermore, we add method `newProxy` to the proxy class, which is responsible for creating a proxy instance (using method `allocateInstance` of class `sun.misc.Unsafe`) and for wrapping the passed up-to-date instance (Action 9.7, Figure 4.12).

In order to enable the proxy for guiding even array instances of type of up-to-date classes through outdated methods, we must add for each array class (i.e., for each array dimension) the corresponding field and proxy creation method to the proxy (Action 9.9 and 9.10, Figure 4.12). Those array related fields and methods do the same job and work in the same manner as their corresponding standard counterparts created with Actions 9.6 and 9.7 depicted in Figure 4.12 .

4.6.4 Method Body Definition Updates

In Section 4.6.1, we pointed out that method body definitions must be updated in a different way than class, method, and field declarations in order to update the references to classes to be reloaded. Therefore, we paid attention to keep them untouched during the declaration update phase. Within this update process step, we go on to update the method body definitions of the classes to be updated according to our update approach.

Therefore, we go through all changed classes and check whether they are interfaces or not (Actions/Decision 10.1 – 10.3, Figure 4.13). This is, because interfaces only declare methods but do not include method body definitions and thus must not be considered. With the next step, we create a Javassist `ExpressionEditor`[4] (Action 10.4, Figure 4.13). The `ExpressionEditor` is responsible for updating the method body definitions.

Ones the `ExpressionEditor` is created, we go through each method of the currently processed class, check whether the methods are abstract or not, and apply the `ExpressionEditor` to all non-abstract methods (Decision/Actions 10.6 – 10.8, Figure 4.13). The `ExpressionEditor` updates every method call (with and without proxies), field access, constructor call (i.e., method call in combination with keyword `new`), `this` call, and `super` call occurring in the method bodies according to our update approach.

An abstract of the bytecode modifications processed by the `ExpressionEditor`, here to update method calls and field accesses, is shown in Figure 4.14. At its heart, the

[4] The real class name in Javassist is `ExprEditor`. But, for clarity reasons, we use class name `ExpressionEditor`.

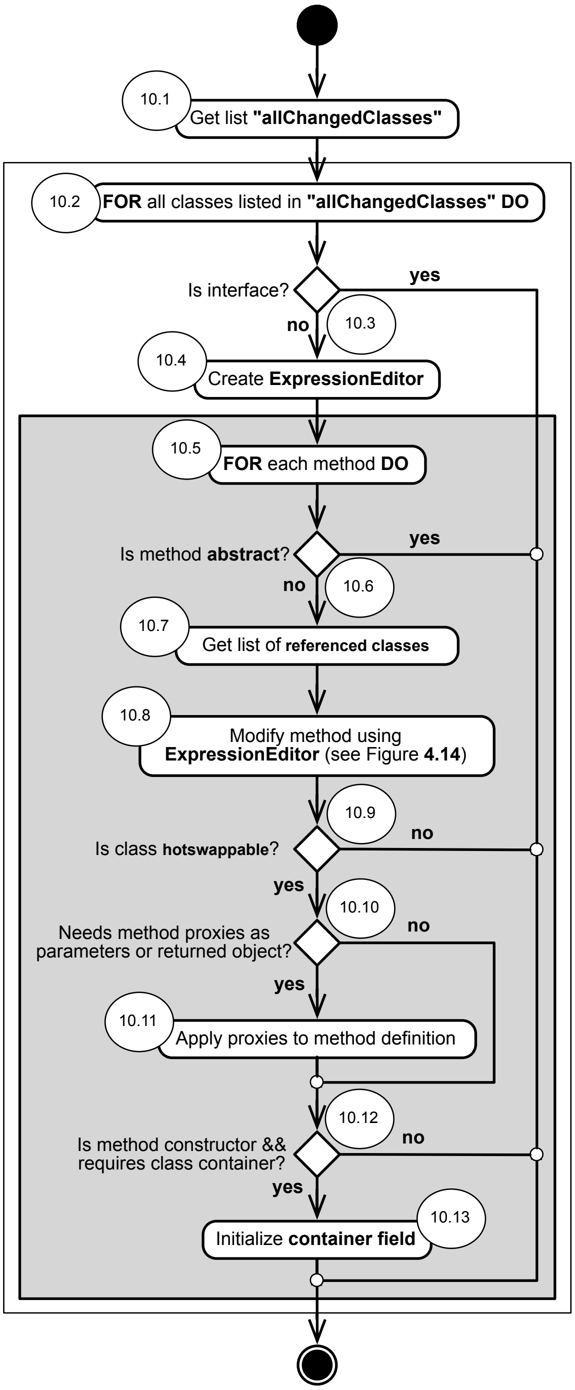

10.1
Get list "allChangedClasses"
10.2
FOR all classes listed in "allChangedClasses" DO
Is interface?
yes
no
10.3
10.4
Create ExpressionEditor
10.5
FOR each method DO
Is method abstract?
yes
no
10.6
10.7
Get list of referenced classes
10.8
Modify method using ExpressionEditor (see Figure 4.14)
10.9
Is class hotswappable?
no
yes
10.10
Needs method proxies as parameters or returned object?
no
yes
10.11
Apply proxies to method definition
10.12
Is method constructor && requires class container?
no
yes
10.13
Initialize container field

Figure 4.13: Update process of method body definitions.

ExpressionEditor redirects all references occurring in the method body to the up-to-date class versions and, if necessary, applies our container and proxy approach. In doing so, the ExpressionEditor does not only consider the references to classes that must be reloaded with a new version in the current update step, but also the references to classes reloaded in earlier application update steps.

```
foreach expression in method body do
    if expression is MethodCall then
        call method of up-to-date class version;
        if method has parameters then
            change parameter types to current class versions;
            if parameters require proxies then
                wrap parameters in proxies;
            end
            if method returns an object then
                if object is proxy then
                    unwrap object wrapped in proxies;
                end
                change object reference type to current class version;
            end
        end
    end
    else if expression is FieldAccess then
        change type of reference to declaring class to up-to-date class version;
        change field reference type to up-to-date class version;
        if field is in container then
            access field through container of up-to-date declaring class;
        end
    end
    else if expression is ... then
        ...
    end
end
```

Figure 4.14: Bytecode modifications processed by ExpressionEditor.

After ExpressionEditor application, we are done with the method body updates of the ¬*hotswappable* classes. But, to finish the method body updates of the *hotswappable* classes, JAVADAPTOR may have to process additional tasks (Decisions/Actions 10.9 – 10.13, Figure 4.13). First, JAVADAPTOR must check whether methods of the currently processed class receive or return proxies, because the methods parameter/return types are outdated. If this is true, the tool adds routines to unwrap the up-to-date parameters and/or to wrap the up-to-date returned objects to every method in need (Action 10.11, Figure 4.13).

Listing 4.1: Unwrapping of up-to-date instances.

```
 1 void setSensor(TempSensor ts) {
 2    0 aload_1
 3    1 checkcast #23 <TempSensor_Proxy_1>
 4    4 getfield #34 <TempSensor_Proxy_1.call>
 5    7 astore_1
 6    8 aload_0
 7    9 aload_1
 8   10 astore_3
 9   11 astore_2
10   12 aload_2
11   13 getfield #36 <TempDisplay.fieldContainer1265725244704>
12   16 checkcast #17 <TempDisplay_Cont_1>
13   19 aload_3
14   20 putfield #38 <TempDisplay_Cont_1.ts>
15   23 return
16 }
```

How to modify the bytecode in order to unwrap proxy-based parameters (here of method `setSensor` of example class `TempDisplay`) is depicted in Listing 4.1 (Lines 2-5). First, we load the parameter stored in a local variable (Line 2). Second, we cast the parameter to the related proxy type (Line 3). Third, we unwrap the up-to-date instance (here of class `TempSensor_v2`) stored in field `call` of the proxy object (Line 4). Fourth, to avoid recurring unwrappings, the unwrapped instance is stored in the local variable that previously stored the proxy (Line 5).

Listing 4.2 shows the bytecode modifications (here of method `getSensor` of example class `TempDisplay`) required to wrap returned up-to-date instances. First, we call method `newProxy` (Line 12) of the proxy class which takes as parameter an instance of the reloaded class (here of class `TempSensor_v2`), wraps the instance by a newly created proxy instance, and returns the proxy. Second, the returned proxy is casted to the type of the old callee class (here of example class `TempSensor`, Line 13).

After we applied the proxies to all methods in need, JAVADAPTOR checks whether the currently processed *hotswappable* class requires containers or not (Decision 10.12, Figure 4.13). In case containers are needed, we initialize the container field within every constructor of the class (Action 10.13, Figure 4.13). That is, we make sure that during class instantiation a container instance is created and all up-to-date instances of ¬*hotswappable* classes owned by the instantiated class are assigned to the container. Doing so, we apply our container approach even to not yet created instances.

Once we are done with the method body updates, all changed classes are prepared for our update approach and could be loaded into the JVM.

Listing 4.2: Wrapping of up-to-date instances.

```
TempSensor getSensor() {
  0 aload_0
  1 astore_1
  2 aconst_null
  3 astore_2
  4 aload_1
  5 getfield #15 <TempDisplay.fieldContainer1265725244704>
  8 checkcast #17 <TempDisplay_Cont_1>
 11 getfield #21 <TempDisplay_Cont_1.ts>
 14 astore_2
 15 aload_2
 16 invokestatic #27 <TempSensor_Proxy_1.newProxy>
 19 checkcast #29 <TempSensor>
 22 areturn
}
```

4.6.5 Class Reloading and HotSwapping

In previous update process steps, we figured out what classes must be updated, considered inheritance hierarchy changes, identified hotswappable and not hotswappable classes, updated class versions (and the corresponding references) regarding compatibility and up-to-dateness, and, if necessary, created container and proxy classes. That is, we modified the classes's bytecode in such way that we could apply the program changes, made by the developer, to the running application.

With this update process step, we load the new class versions into the JVM and update all changed classes with an untouched schema via Java HotSwap. After retrieving list *allChangedClasses*, we sort its elements according their *hotswappability* (Actions 11.1 and 11.2, Figure 4.15). Next, we pause the application (Action 11.3, Figure 4.15). We do this in order to ensure that *all* updated class files are loaded into the JVM before resuming it and to avoid state losses during state mapping because the JVM concurrently produces new objects not considered to be mapped.

Then, we proceed to load the new versions of ¬*hotswappable* classes and their possible proxy classes (Actions 11.5 and 11.6, Figure 4.15). Through loading those classes before hotswapping the others, we ensure that the classes are already present within the JVM and thus could be referenced by the *hotswappable* classes without problems. Next, we load the new class versions in order to process the state mappings. After loading the new class versions, we go on to load their possible container classes and hotswap the *hotswappable* classes (Actions 11.8 and 11.9, Figure 4.15), which concludes the update process step.

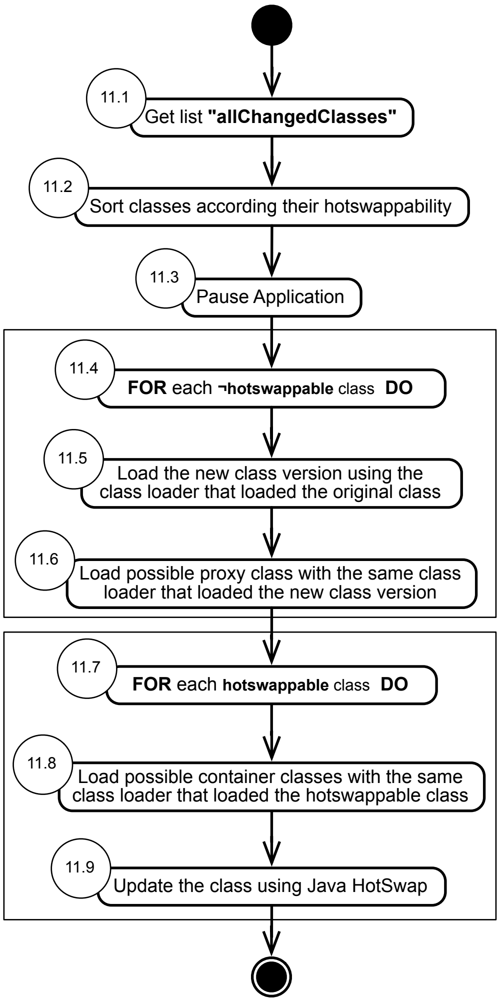

11.1
Get list **"allChangedClasses"**
11.2
Sort classes according their hotswappability
11.3
Pause Application
11.4
FOR each ¬**hotswappable** class **DO**
11.5
Load the new class version using the class loader that loaded the original class
11.6
Load possible proxy class with the same class loader that loaded the new class version
11.7
FOR each **hotswappable** class **DO**
11.8
Load possible container classes with the same class loader that loaded the hotswappable class
11.9
Update the class using Java HotSwap

Figure 4.15: Application of the program update.

4.7 State Mapping Preparation

At this stage of our update process, all classes affected by the scheduled runtime program update are prepared for the update and already present within the JVM. Now it is time to map the state from the outdated class versions to their up-to-date counterparts and for the assignment of the latter ones to the referring program parts.

However, before we can process the state mapping, we need to know what state must be mapped. For this purpose, we retrieve all ¬*hotswappable* classes, go through the resulting list of ¬*hotswappable* classes, and request their instances from the JVM via the JVMTI (Actions 12.1 – 12.3, Figure 4.16). Furthermore, we retrieve for each instance of the ¬*hotswappable* class the referring instances (Action 12.5, Figure 4.16) using JVMTI method `getReferringObjects`. With Decision 12.7, JAVADAPTOR then checks for each referring instance if it belongs to an application class or not. If this is true, we store the referring instance in list *allInstances* (Action 12.8, Figure 4.16).

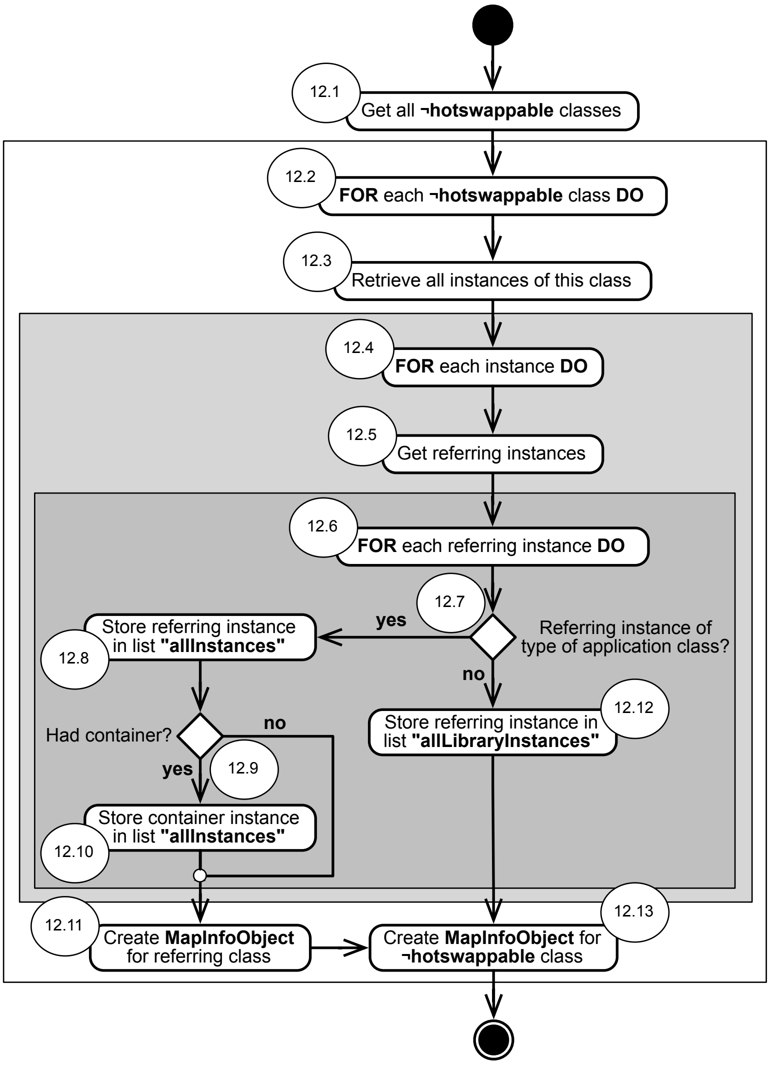

12.1
Get all ¬**hotswappable** classes
12.2
FOR each ¬**hotswappable** class **DO**
12.3
Retrieve all instances of this class
12.4
FOR each instance **DO**
12.5
Get referring instances
12.6
FOR each referring instance **DO**
12.7
Referring instance of type of application class?
yes
no
Store referring instance in list **"allInstances"**
12.8
Had container?
no
yes
12.9
Store container instance in list **"allInstances"**
12.10
Store referring instance in list **"allLibraryInstances"**
12.12
12.11
Create **MapInfoObject** for referring class
Create **MapInfoObject** for ¬**hotswappable** class
12.13

Figure 4.16: Gain mapping information.

Next, we retrieve the possible previous container instance for the referring instance and store it in list *allInstances*, too (Decision 12.9 and Action 12.10, Figure 4.16). This is necessary, because otherwise the state assigned to the container instance would be ignored and thus be lost. Once JAVADAPTOR identified all instances (including container instances) that belong to the currently processed referring application class, it creates a `MapInfoObject` object for this class (Action 12.11, Figure 4.16). The `MapInfoObject` object will be used in later state mapping steps and contains information like:

- information about possible containers.
- container class name.
- fields stored in the *old* container.
- fields that will be stored in the *new* container.

With Actions/Decision 12.8 – 12.11 (see Figure 4.16), JavAdaptor collected all instances of application classes including containers that refer to instances of ¬*hotswappable* classes. Anyway, ¬*hotswappable* classes cannot only be referred by application classes, but also by the Java standard library or external libraries. Those libraries are not aware of the actual type of the ¬*hotswappable* class, but could refer to it through variables of super type of the ¬*hotswappable* class (also known as *polymorphism*). This is illustrated in Listing 4.3, where instance field `obj` of class `LibraryClass` is initialized with an instance of `ApplicationClass` (see Line 11).

Listing 4.3: References in library classes.

```
 1 class LibraryClass {
 2   Object obj;
 3 
 4   LibraryClass(Object obj) {
 5     this.obj = obj;
 6   }
 7   ...
 8 }
 9 
10 class ApplicationClass {
11    LibraryClass lib = new LibraryClass(new ApplicationClass());
12    ...
13 }
```

If the currently processed referring instance belongs to a library class, we store it in list *allLibraryInstances* (Action 12.12, Figure 4.16). Afterwards, we create for the currently processed ¬*hotswappable* class the corresponding `MapInfoObject` (Action 12.13, Figure 4.16), which will be used later on and stores information such as:

- information about possible containers.
- container class name.
- fields stored in the container.
- class version history.
- superclasses shared by *old* and *new* class version.
- fields (including superclass fields) owned by *old* as well as *new* class version.

4.8 State Mapping Proceeding

With the information gained in the previous update process step, we have all pieces together to process the state mapping, which concludes the update process.

The state mapping consists of two different steps. First, we must map the (class and instance specific) state from old to new version of a ¬*hotswappable* class. Second, we have to update the state of the referring program parts. That is, the (class and instance specific) state of the new ¬*hotswappable* class version must be assigned to the referring classes and instances to let the new ¬*hotswappable* class version become part of program execution.

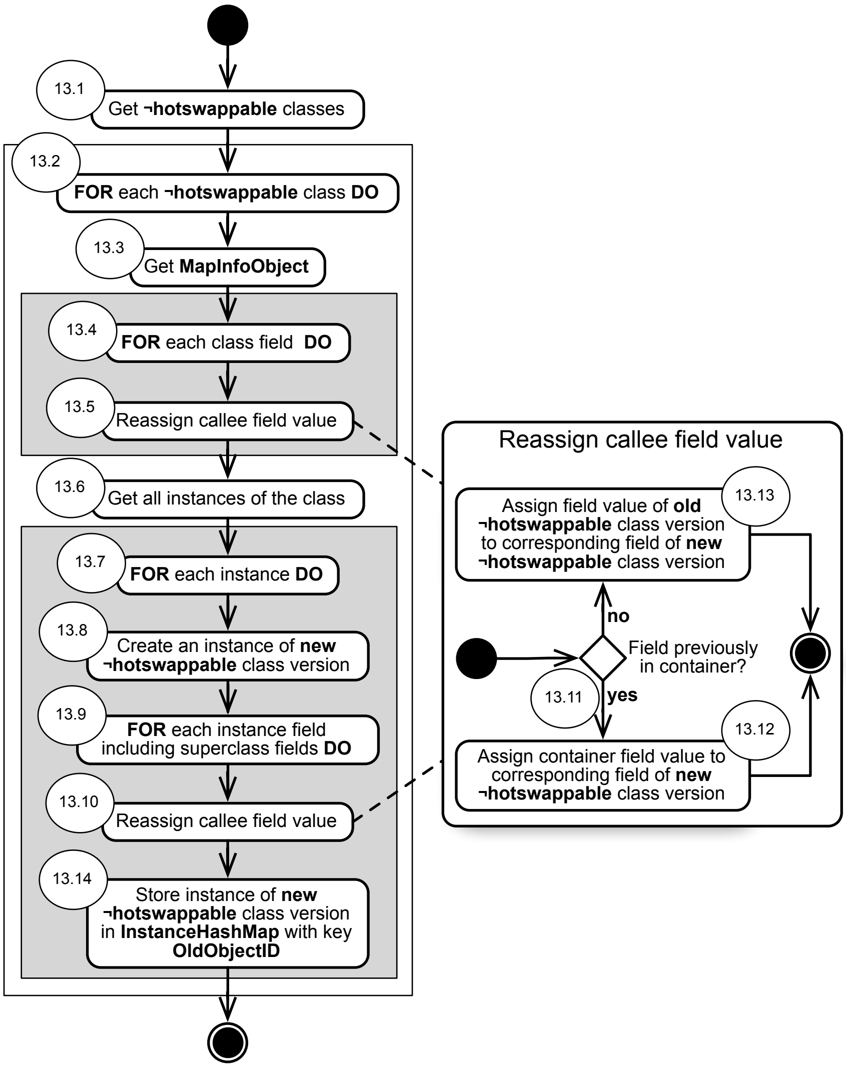

13.1
Get ¬hotswappable classes
13.2
FOR each ¬hotswappable class DO
13.3
Get MapInfoObject
13.4
FOR each class field DO
13.5
Reassign callee field value
13.6
Get all instances of the class
13.7
FOR each instance DO
13.8
Create an instance of new ¬hotswappable class version
13.9
FOR each instance field including superclass fields DO
13.10
Reassign callee field value
13.14
Store instance of new ¬hotswappable class version in InstanceHashMap with key OldObjectID
Reassign callee field value
13.13
Assign field value of old ¬hotswappable class version to corresponding field of new ¬hotswappable class version
no
Field previously in container?
13.11
yes
13.12
Assign container field value to corresponding field of new ¬hotswappable class version

Figure 4.17: State mapping of ¬*hotswappable* classes.

4.8.1 State Mapping of ¬*HotSwappable* Classes

In Figure 4.17, we sketch the first state mapping step, i.e., the state mapping from old to new ¬*hotswappable* class version. For each ¬*hotswappable* class version, JAVADAPTOR retrieves the corresponding `MapInfoObject` (Action 13.3, Figure 4.17). Next, JAVADAPTOR maps the class specific state field by field to the new ¬*hotswappable* class version (Actions 13.4 and 13.5, Figure 4.17).

The concrete mapping of a value assigned to a field is sketched by subdiagram *"Reassign callee field value"* of Figure 4.17. First, JAVADAPTOR checks whether the field was previously outsourced to a container (Decision 13.11, Figure 4.17). If true, we assign the value of the container field to the corresponding field of the new ¬*hotswappable* class version (Action 13.12, Figure 4.17). In case the field was regularly owned by the old ¬*hotswappable* class version, JAVADAPTOR assigns the field's value to the corresponding newly created ¬*hotswappable* class version's field (Action 13.13, Figure 4.17).

When we are done with the class specific state mapping, we go on mapping the instance specific state from old to new ¬*hotswappable* class version. Therefore, we retrieve all instances of the old ¬*hotswappable* class version and create for each instance an instance of the new ¬*hotswappable* class version (Actions 13.6 – 13.8, Figure 4.17).

The instantiation is triggered by JAVADAPTOR. The corresponding code is depicted in Listing 4.4. Method `mapCalleeWithMappingObject` of JAVADAPTOR invokes method `mapAllObjects` of class `Mapping` in the target application which in turn calls method `createNewObject` of the same class. Listing 4.5 shows a code snippet of this method. Via method `forName`, we retrieve the class object of the new ¬*hotswappable* class version (Line 22, Listing 4.5). In the next step, we create the new class version's objects. For this purpose, we could have used method `newInstance` of the retrieved class object. However, the problem with method `newInstance` is that it initializes the object fields during object creation, which is both, time consuming and unnecessary, because we will overwrite the values in later state mapping steps. Therefore, we call method `allocateInstance` of class `sun.misc.Unsafe` instead of method `newInstance` (Line 22, Listing 4.5). It creates the object, but does not initialize any field, which reduces the time required for object creation.

After creating the up-to-date instance for an outdated instance, we map the state field by field from old instance to the newly created one (Actions 13.9 and 13.10, Figure 4.17). The concrete state mapping is sketched by subdiagram *"Reassign callee field value"* of Figure 4.17 and works the same way as for class fields, whose state mapping procedure we already described above. Finally, we store the newly created instance in data structure `InstanceHashMap` (Action 13.14, Figure 4.17). The newly created instance can be afterwards retrieved from `InstanceHasMap` using the *object ID* of the outdated instance.

Listing 4.4: JavAdaptor – instantiation.

```
class Saver {
  ClassObjectReference mappingClass;
  ...
  void mapCalleeWithMappingObject( ... ) {
    ...
    mappingClass.invokeMethod(threadRef, ’’mapAllObjects’’, parameterList, options);
    ...
  }

}
```

Listing 4.5: Target VM – instantiation.

```
class Mapping extends Thread {
  Unsafe unsafe;
  ...
  void mapAllObjects(Object[] oldCallees, Object[] caller, String msg){
    ...
    createNewObject(mapInfo.getNewName(), oldCallee.getClass().getClassLoader());
    ...
  }

  Object createNewObject(String className, ClassLoader classLoader) {
    ...
    return unsafe.allocateInstance(classObj));
  }
}
```

4.8.2 State Update of Referring Program Parts

Once we are done with the state mappings, we continue with the assignment of the new ¬*hotswappable* class versions and their instances to the referring classes and instances, i.e., we update the state of the caller side. The state-update strategy of JavAdaptor strongly depends on the type of the referring program part, i.e., it depends on whether an old ¬*hotswappable* class version is referenced by an application class or a library class (Decision 14.3, Figure 4.18).

Different from state updates of application classes, state updates of referring library classes require no containers. This is, because library class fields that refer to ¬*hotswappable* classes are normally of super type of the ¬*hotswappable* classes (see Listing 4.3). Thus, we can directly assign instances of the new ¬*hotswappable* class version to those fields without any container. Only requirement is that old and new ¬*hotswappable* class version share the superclass through which the library class accesses the ¬*hotswappable* class. By contrast, application class fields may be of type of the ¬*hotswappable* class and thus require container-based reference updates.

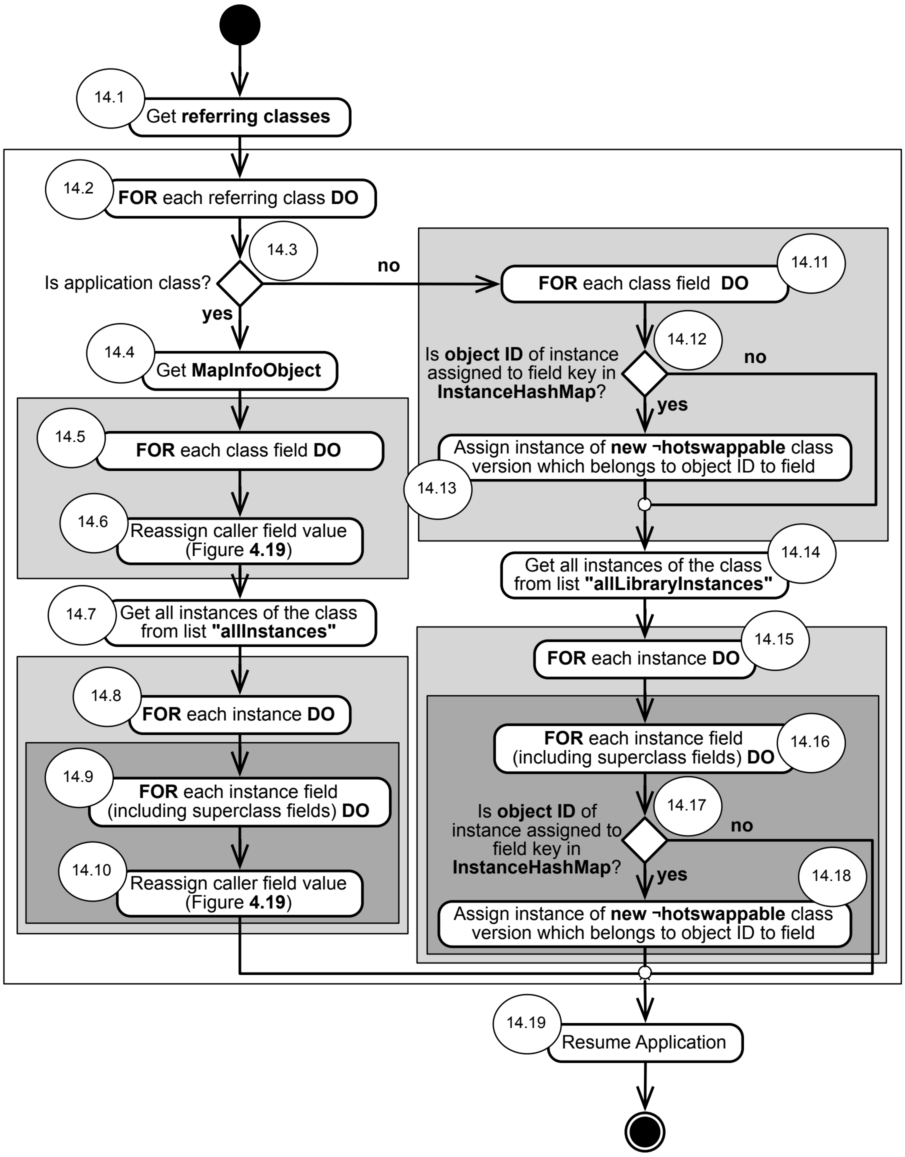

14.1
Get **referring classes**
14.2
FOR each referring class **DO**
14.3
Is application class?
no
yes
14.4
Get **MapInfoObject**
14.5
FOR each class field **DO**
14.6
Reassign caller field value (Figure **4.19**)
14.7
Get all instances of the class from list **"allInstances"**
14.8
FOR each instance **DO**
14.9
FOR each instance field (including superclass fields) **DO**
14.10
Reassign caller field value (Figure **4.19**)
14.11
FOR each class field **DO**
14.12
Is **object ID** of instance assigned to field key in **InstanceHashMap**?
no
yes
Assign instance of **new ¬hotswappable** class version which belongs to object ID to field
14.13
14.14
Get all instances of the class from list **"allLibraryInstances"**
14.15
FOR each instance **DO**
14.16
FOR each instance field (including superclass fields) **DO**
14.17
Is **object ID** of instance assigned to field key in **InstanceHashMap**?
no
yes
14.18
Assign instance of **new ¬hotswappable** class version which belongs to object ID to field
14.19
Resume Application

Figure 4.18: Assignment of new instances to referring classes/instances.

Application-Class-Specific State Updates. If the currently processed referring class is an application class, JavAdaptor updates the state as sketched by Actions 14.4 – 14.10 of Figure 4.18. First, we retrieve the `MapInfoObject` that belongs to the currently processed class (Action 14.4, Figure 4.18). Next, we begin with the update of possible class fields of type of ¬*hotswappable* classes (Actions 14.5 and 14.6, Figure 4.18). The corresponding subprocess *"Reassign caller field value"* (Action 14.6, Figure 4.18) is sketched in Figure 4.19 and works as follows:

> In subprocess *"Reassign caller field value"*, JavAdaptor first checks whether the currently processed field was outsourced into a container or not (Decision 15.1, Figure 4.19).
>
> If the field was not outsourced, the tool retrieves the *object ID* of the object assigned to the field and checks if the ID is a key value in `InstanceHashMap` (Decision 15.2, Figure 4.19), i.e., JavAdaptor tests if the referenced object is outdated (of type of an old ¬*hotswappable* class version). In case the referenced object is outdated, we get its up-to-date counterpart from `InstanceHashMap` and, depending on Decision 15.3 (see Figure 4.19), assign the up-to-date instance either to the currently processed (caller) field (Action 15.4, Figure 4.19) or to the corresponding field of the new container (Action 15.8, Figure 4.19).

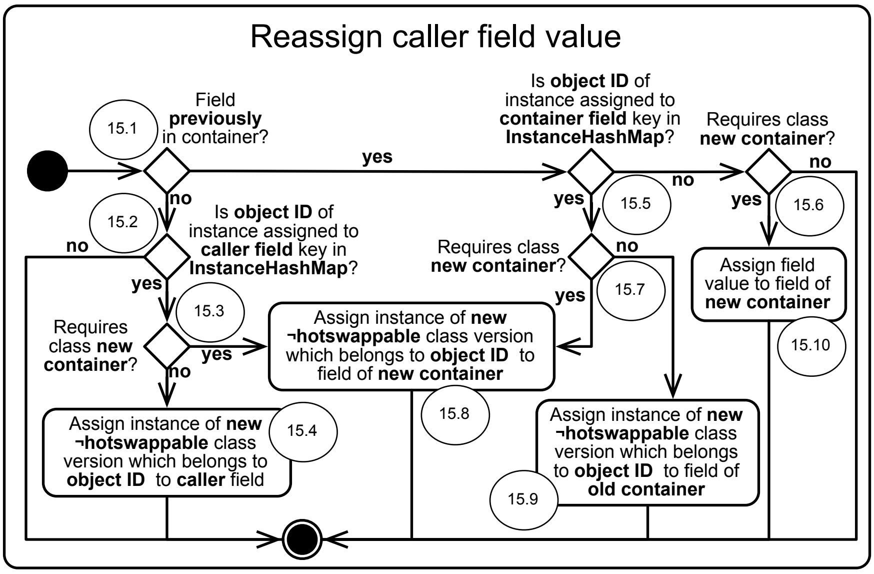

Reassign caller field value
15.1
Field **previously** in container?
yes
Is **object ID** of instance assigned to **container field** key in **InstanceHashMap**?
Requires class **new container**?
no
no
yes
15.5
yes
15.6
no
15.2
Is **object ID** of instance assigned to **caller field** key in **InstanceHashMap**?
no
Requires class **new container**?
no
yes
15.7
Assign field value to field of **new container**
yes
15.3
Requires class **new container**?
yes
no
Assign instance of **new ¬hotswappable** class version which belongs to **object ID** to field of **new container**
15.10
15.8
Assign instance of **new ¬hotswappable** class version which belongs to **object ID** to **caller** field
15.4
Assign instance of **new ¬hotswappable** class version which belongs to **object ID** to field of **old container**
15.9

Figure 4.19: Subprocess "Reassign caller field value" of Figure 4.18.

> Decisions/Actions 15.5 – 15.9 (see Figure 4.19) sketch how to update the currently processed field if it was (as a result of earlier program updates) shifted

into a container. Different from Decision 15.2, we do not have to check the *object ID* of the instance assigned to the class field, but of the instance assigned to the corresponding container field (Decision 15.5, Figure 4.19). If JAVADAPTOR finds out that the instance's *object ID* is a key value in `InstanceHashMap`, i.e., if the instance is outdated, it (as a function of Decision 15.7, Figure 4.19) assigns the compatible up-to-date instance either to the corresponding field of the new container (Action 15.8, Figure 4.19) or to the currently processed old container field (Action 15.9, Figure 4.19). However, we must not only handle fields which refer to outdated instances. Additionally, we have to assign the unchanged instances referenced by the old container to the possible new container (Decision 15.6 and Action 15.10, Figure 4.19).

If JAVADAPTOR is done with the class specific state updates of the currently processed application class, it continues to update the state of the class's instances. Therefore, the tool retrieves all instances of the currently processed application class (Action 14.7, Figure 4.18). Next, it updates the state instance by instance and field by field (Actions 14.8 – 14.10, Figure 4.18). The corresponding field update process is depicted in Figure 4.19, which we already described above.

Library-Class-Specific State Updates. As mentioned at the beginning of this section, state updates of referring library classes differ from state updates of referring application classes. The big difference is that state updates of referring library classes require no containers.

Actions/Decisions 14.11 – 14.18 of Figure 4.18 describe how JAVADAPTOR updates the state of referring library classes and their instances. First, the tool updates the static fields of the currently processed library class (Actions/Decision 14.11 – 14.13, Figure 4.18). In case the object assigned to the currently processed field is outdated (i.e., its *object ID* appears to be a key value in *InstanceHashMap*), JAVADAPTOR assigns the corresponding up-to-date instance to this field. Next, we retrieve all instances of the library class (Action 14.14, Figure 4.18) and update them instance by instance and field by field the same way we updated the static fields of the library class (Decision/Action 14.15 – 14.18, Figure 4.18).

Ones we have updated the state of all referring classes and their instances, JAVADAPTOR is done with the application of the program update, i.e., the program changes are fully applied to the running program and JAVADAPTOR resumes the application (Action 14.19, Figure 4.18). This last step concludes the application update.

4.9 Summary

This chapter revealed the internals of our current JAVADAPTOR implementation. We first described JAVADAPTOR from the developers/users point of view. We further went through every part of JAVADAPTOR's update workflow, detailed its implementation, and described possible implementational pitfalls. The information given in this chapter may help people, interested in JAVADAPTOR, to reproduce our solution. The chapter further imparts the knowledge required to apply our approach to languages different from Java.

5 Evaluation

This chapter shares material with the SPE'2011 paper "JavAdaptor – Flexible Runtime Updates of Java Applications" [PKC+12] and the ICSE'2011 paper "JavAdaptor: Unrestricted Dynamic Software Updates for Java" [PGS+11].

Our goal was to develop a dynamic software update approach that is *highly flexible*, *performant*, *platform independent*, *architecture independent*, and *fine-grained*. In this chapter, we evaluate whether JAVADAPTOR meets the stated goals. We will start with the presentation of some non-trivial case studies showing the flexibility and fine update granularity of JAVADAPTOR. Next, we measure the runtime and update performance of JAVADAPTOR and demonstrate that our tool can be applied to runtime environments different from the environment used during implementation and testing (i.e, we demonstrate its platform independence). Furthermore, we discuss why JAVADAPTOR is architecture independent and present some results underpinning our arguments. In addition to the evaluation of the stated goals, we analyze the memory consumption of JAVADAPTOR, because large memory footprints may hinder the usage of JAVADAPTOR on systems with less (main) memory.

5.1 Case Studies

In order to demonstrate that JAVADAPTOR offers highly flexible and fine-grained updates, we applied it to three non-trivial case studies. With the first two case studies (HyperSQL and Snake), we simulated real-world update scenarios. With the third case study (Refactorings), we give evidence of the generality of our update approach.

5.1.1 HyperSQL

In our first case study, we dynamically updated *HyperSQL*[1] (amongst others used by *Open Office*) from version 1.8.0.9 to version 1.8.0.10. We chose HyperSQL, because it is a database management system for which runtime adaptation promises benefits of no downtime. To ensure that we do not deliberately choose an application for which we a priori know that JAVADAPTOR is capable to dynamically update it, we did not previously check what kinds of updates would be required. This is even true for the HyperSQL version we decided to

[1] http://hsqldb.org/

update (note that we chose to update HyperSQL version 1.8.0.9 to version 1.8.0.10 because they were the freshest versions available at the time we performed the case study).

Replaced Class	**Reference Updates**		
Kind of Update	**Short-Lived Obj. (# of Ref.)**	**Container (# of Ref.)**	**Proxy (# of Ref.)**
FontDialogSwing structural update	8 (9)	0 (-)	0 (-)
HsqlDatabaseProperties functional update	11 (98)	2 (25)	11 (23)
LockFile functional update	1 (9)	10 (5×)	11 (47)
LockFile$HeartbeatRunner functional update	2 (2)	0 (-)	0 (-)
Logger structural update	22 (93)	3 (93)	3 (4)
NIOLockFile changed inherit. hierarchy	0 (-)	0 (-)	0 (-)
ScriptReaderZipped functional update	3 (3)	0 (-)	0 (-)
SimpleLog structural update	9 (105)	3 (27)	0 (-)
Token structural update	5 (671)	0 (-)	0 (-)
Trace structural update	80 (1306)	0 (-)	0 (-)
Transfer structural update	4 (6)	0 (-)	0 (-)
View functional update	3 (37)	3 (13)	3 (16)

Table 5.1: HyperSQL: Required class reloadings because of schema changes. The table lists all classes to be reloaded. It furthermore provides information on the required caller updates, i.e., how many referring classes are updated in the context of short-lived objects, containers, or proxies. The number of updated references is given as well (in brackets).

We downloaded HyperSQL version 1.8.0.9, as well as version 1.8.0.10, from the website and started version 1.8.0.9. After program start, we ran the open-source database benchmark *PolePosition*[2] in order to generate and query some data, which ensured that HyperSQL was fully activated and deployed. Afterwards, we applied all changes required to evolve the

[2] http://polepos.sourceforge.net/

running application from version 1.8.0.9 to version 1.8.0.10 without shutting it down.

The new version of HyperSQL (released 9 month after version 1.8.0.9) comes with a bunch of changes. It fixes major bugs that cause null-pointer exceptions, problems with views, timing issues, corrupted data files, and deadlocks. Additionally, new and improved functionality such as new lock-file implementations and performance improvements to the web server are included. To lift the running program from version 1.8.0.9 to the new version 1.8.0.10, we had to update 33 of 353 classes. In case of 21 out of 33 classes, the changes did not affect the class schema, i.e., JAVADAPTOR could apply the changes solely using Java HotSwap. Apart from that, 12 classes were affected by schema-changing program modifications. JAVADAPTOR replaced them using class reloadings. The corresponding state mappings span one-to-one mappings, added, and removed fields, i.e., they were automated by our tool. Table 5.1 lists all classes that had to be replaced. Note that updating class `NIOLockFile` also included changes to the inheritance hierarchy. In addition, with class `LockFile$HeartbeatRunner`, we had to update even a nested class. Table 5.1 provides also information about the required caller updates, i.e., how many caller classes are updated in the context of short-lived objects, containers, or proxies. The number of references within method bodies that have to be changed to update the caller classes is given as well (in brackets). In 148 out of 197 cases (75.1 %), we had to update callers because of references to short-lived callee objects (via Java HotSwap). In case of 21 caller classes (10.7 %) JAVADAPTOR was forced to apply containers. 28 caller class updates (14.2 %) required proxies.

In order to verify that HyperSQL was still correctly working (in a consistent state) after the update, we reran the PolePosition benchmark. In the result, HyperSQL passed the benchmark without errors, i.e., all database operations were correctly executed after the update. In a second test, we checked whether the updates were applied and active. Therefore, we hooked the JVM profiler *VisualVM*[3] into the running application and checked what classes/methods were executed during the PolePosition benchmark. We found out that 5 of the 12 replaced classes were active and central part of program execution during the PolePosition benchmark which confirms that they were updated correctly. The remaining 7 classes were correctly loaded into the JVM, but inactive during the benchmark. Thus, we could not verify their correct execution.

To get to know whether JAVADAPTOR solely updated the classes we identified to be due for updates (via comparing the source code of both HyperSQL versions), we proceeded as follows. We let JAVADAPTOR log the names of the classes it reloaded and the names of the classes whose references it updated. Afterwards, we compared the log files with the results derived from our source code reviews and even reviewed the bytecode of the class files changed by JAVADAPTOR. The outcome of the comparison was, that JAVADAPTOR

[3] https://visualvm.dev.java.net/

only updated the classes we expected to be updated and nothing else. That is, JavAdaptor applied the updates at the desired fine level of granularity.

5.1.2 Snake

As the second case study, we update the well-known arcade game *Snake* [PGS$^+$11]. With this case study, we demonstrate that JavAdaptor is not only beneficial in terms of updates of highly available applications but also during development. Note that, typically, developers stepwisely enhance a program and test whether the added code is correct or not, which could be an annoying task if for each test the program must be restarted. In addition, we chose a graphical application because in Java, graphical functions are strongly coupled with the API and changes to those functions influence wide parts of the system. This renders proper updates of graphical applications difficult and lets incomplete DSU approaches fail in such scenarios. That is, graphical applications are predestinated to substantiate that a DSU approach is capable to correctly update program parts which affect many different parts of the system.

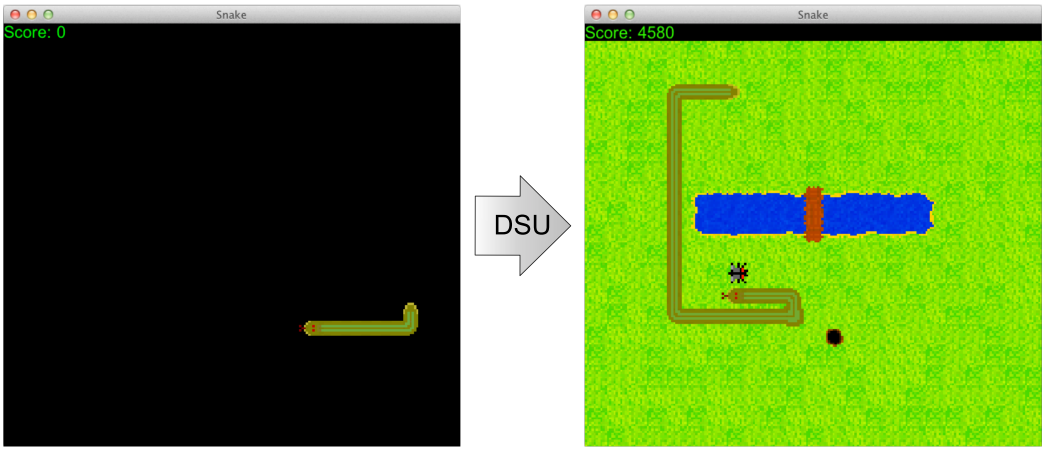


Figure 5.1: Dynamic update of *Snake*.

Different from our HyperSQL case study, we did not update Snake from one version to another version. Instead, we aimed at successively updating the application in a developer scenario. In order to not bias the case study, we proceeded as follows. First, we predefined the number of update steps, which (arbitrarily) was 4. Next, we predefined for each update step what functionality it should add to the running program, while we avoided to figure out the program parts affected by the updates. That is, at the time of predefinition, we had no clue what program parts must be changed in order to apply the new functionality. After predefining the program enhancements, we identified the source code sections which constitute the enhancements, commented them out, and started the now very basic Snake program.

In the update process, we made updates from small changes that only change a method body (that would already been supported by Java HotSwap) to massive changes that introduce new methods, fields, or even change inheritance hierarchies (which is not possible with any standard JVM). Figure 5.1 illustrates Snake before (left side) and after (right side) the 4 update steps. A video, showing the update steps we processed using JAVADAPTOR, is available on YouTube.[4]

As a result of our Snake case study, all updates could be successfully applied to the running program, which demonstrates the usefulness of JAVADAPTOR during development and its capability to update program parts that have a deep impact on the system.

To get evidence of the update granularity, we proceeded in the same manner as we did in our HyperSQL case study. That is, we let JAVADAPTOR log what classes it reloads and what classes it updates because of references to the reloaded classes. Next, we compared the log file with the results from our source code reviews and additionally checked the bytecode of the changed class files. As with our HyperSQL case study, JAVADAPTOR only updated the classes we previously identified to be due for updates, i.e., we achieved the desired fine update granularity.

5.1.3 Refactorings

The HyperSQL as well as the Snake case study show the flexibility and practicability of JAVADAPTOR. However, we could continue indefinitely making such case studies demonstrating the capabilities of our tool and would end up each time with just another case study. The problem with case studies such as HyperSQL and Snake is that they present specific update scenarios, which may not cover all eventualities and thus do not allow us to draw conclusions on the general applicability of JAVADAPTOR.

To get a better understanding of JAVADAPTOR's general applicability, we followed a different path and checked if the tool would be able to dynamically apply common program updates, i.e, updates, that frequently occur in practice and do not rely on certain application scenarios. But, what are common program updates and how could we unbiased test if JAVADAPTOR is able to apply them to running applications? We found *Refactorings* [OJ90] to be appropriate for our analysis. Actually, Dig and Johnson [DJ06] found out that:

> *Refactorings cause more than 80 % of API changes that were not backwards-compatible.*

[4] http://www.youtube.com/watch?v=jZm0hvlhC-E

Once we decided to demonstrate the general applicability of JAVADAPTOR on the basis of refactorings, we had to reason about a test setup which ensures the tests to be unbiased. Our tests base on the refactorings presented by Fowler [Fow06], which is the standard reference regarding refactorings. To achieve an unbiased test setup, we simply took the example programs from Fowler and refactored them at runtime. JAVADAPTOR was able to process all refactorings including possible state mappings. Table 5.2 lists all 72 *Refactorings* presented by Fowler and gives information about possible class reloadings (column *Requirements*), what kinds of *Reference Updates* were required (i.e., *HotSwap*, *Containers*, and/or *Proxies*), and whether we could automatically map the state or had to define *Mapping* methods.

In a nutshell, JAVADAPTOR was able to successfully apply all 72 refactorings at runtime. Class reloadings were necessary in 61 out of 72 cases. That is, approximately 84 % of the refactorings required class schema changes and thus were way beyond the capabilities of Java HotSwap. Reference updates because of class reloadings, required containers in 57 out of 61 cases (ca. 93 %) and proxies in 3 out of 61 cases (ca. 5 %). State mappings could be automatically processed in 41 out of 61 cases (ca. 67 %), while mapping methods were required in 20 out of 61 cases (ca. 33 %).

Refactoring	Requirements	Ref. Update	Mapping
Add Parameter	Class Reloading	Container	Automatic
Change Bidirectional Association to Unidirectional	Class Reloading	Container	Automatic
Change Reference to Value	Class Reloading	HotSwap	Automatic
Change Unidirectional Association to Bidirectional	Class Reloading	Container	Automatic
Change Value to Reference	Class Reloading	HotSwap	Automatic
Collapse Hierarchy	HotSwap	–	–
Consolidate Conditional Expression	Class Reloading	Container	Automatic
Consolidate Duplicate Conditional Expression	HotSwap	–	–
Convert Procedural Design to Objects	Class Reloading	Container	Automatic
Decompose Conditional	Class Reloading	Container	Automatic
Duplicate Observed Data	Class Reloading, New Class	Container	Method
Encapsulate Collection	Class Reloading	Container	Automatic
Encapsulate Downcast	Class Reloading	Container	Automatic
Encapsulate Field	Class Reloading	Container	Automatic
Extract Class	Class Reloading, New Class	Container	Automatic
Extract Hierarchy	Class Reloading	Container	Method
Extract Interface	Class Reloading, New Class	Container	Method
Extract Method	Class Reloading	Container	Automatic
Extract Subclass	Class Reloading, New Class	Container	Automatic

Refactoring	Requirements	Ref. Update	Mapping
Extract Superclass	Class Reloading, New Class	Container	Automatic
Form Template Method	Class Reloading	Container	Automatic
Hide Delegate	Class Reloading	Container	Automatic
Hide Method	Class Reloading	Container	Automatic
Inline Class	Class Reloading, New Class	Container	Method
Inline Method	Class Reloading	Container	Automatic
Inline Temp	HotSwap	–	–
Introduce Assertion	HotSwap	–	–
Introduce Explaining Variable	Class Reloading	Container	Automatic
Introduce Foreign Method	Class Reloading	Container	Automatic
Introduce Local Extension	Class Reloading, New Class	HotSwap	Automatic
Introduce Null Object	Class Reloading, New Class	Container	Automatic
Introduce Parameter Object	Class Reloading	Container, Proxy	Automatic
Move Field	Class Reloading	Container	Method
Move Method	Class Reloading	Container, Proxy	Automatic
Parameterize Method	Class Reloading	Container	Automatic
Preserve Whole Object	Class Reloading	Container, Proxy	Automatic
Pull Up Constructor Body	Class Reloading	Container	Method
Pull Up Field	Class Reloading	Container	Method
Pull Up Method	Class Reloading	Container	Method
Push Down Field	Class Reloading	Container	Method
Push Down Method	Class Reloading	Container	Method
Remove Assignments to Parameters	HotSwap	–	–
Remove Control Flag	HotSwap	–	–
Remove Middle Man	Class Reloading	Container	Automatic
Remove Parameter	Class Reloading	Container	Automatic
Remove Setting Method	Class Reloading	Container	Automatic
Rename Method	Class Reloading	Container	Automatic
Replace Array with Object	Class Reloading, New Class	Container	Method
Replace Conditional with Polymorphism	Class Reloading	Container	Automatic
Replace Constructor with Factory Method	Class Reloading, New Class	Container	Automatic
Replace Data Value with Object	Class Reloading, New Class	HotSwap	Method
Replace Delegation with Inheritance	Class Reloading	Container	Method
Replace Error Code with Exception	HotSwap	–	–
Replace Exception with Test	HotSwap	–	–

Refactoring	Requirements	Ref. Update	Mapping
Replace Inheritance with Delegation	Class Reloading	Container	Method
Replace Magic Number with Symbolic Constant	Class Reloading	Container	Automatic
Replace Method with Method Object	Class Reloading, New Class	Container	Automatic
Replace Nested Conditional with Guard Clauses	HotSwap	–	–
Replace Parameter with Explicit Methods	Class Reloading	Container	Automatic
Replace Parameter with Method	Class Reloading	Container	Automatic
Replace Record with Data Class	Class Reloading, New Class	Container	Method
Replace Subclass with Fields	Class Reloading, New Class	Container	Method
Replace Temp with Query	Class Reloading	Container	Automatic
Replace Type Code with Class	Class Reloading, New Class	Container	Method
Replace Type Code with State/Strategy	Class Reloading, New Class	Container	Method
Replace Type Code with Subclasses	Class Reloading, New Class	Container	Method
Self Encapsulate Field	Class Reloading	Container	Automatic
Separate Domain from Presentation	Class Reloading	Container	Automatic
Separate Query from Modifier	Class Reloading	Container	Automatic
Split Temporary Variable	HotSwap	–	–
Substitute Algorithm	HotSwap	–	–
Tease Apart Inheritance	Class Reloading	Container	Method

Table 5.2: Runtime refactorings using JAVADAPTOR.

To sum up, the results of our refactoring case study show that JAVADAPTOR covers a large bandwidth of different update scenarios and chances are high that the tool performs well in most real-world scenarios.

5.2 Performance

Having demonstrated JAVADAPTOR's ability to update complex real-world applications at a fine level of granularity, it is time to take a look at possible performance penalties induced by our approach, i.e., its impact on the program execution speed and the time it needs to apply the update. For our performance examinations, we use the previous studies and synthetic benchmarks. All benchmarks are performed on a machine hosting an Intel 2,2 GHz i7 Quad Core CPU and running Windows 7 64 bit. As Java Runtime Environment, we use officially released *Java 7* (version 1.7.0) based on Oracle's HotSpot VM (64 bit).

5.2.1 Statistical Significance

Benchmarking of Java applications is a difficult task, because the benchmark results could be influenced in many different ways. As Goerges et al. [GBE07] state, one problem is, that the operating system may or may not favor tasks over the benchmarked Java task, which lets the benchmark numbers differ from run to run. Furthermore, the just-in-time compiler of the JVM requires some time to optimize the program (referred to as warm-up time) and on top of that may optimize method A in one run and method B in another run, resulting in varying benchmark numbers, too. Another interference factor are concurrent threads that may be scheduled differently during the runs.

In order to eliminate the impact of the interference factors described above, i.e., to get statistical significant results, we proceed with our benchmarks as follows. First of all, we warm-up the program before we benchmark it, in order to avoid varying results because benchmark and JIT compiler compete for CPU cycles. Furthermore, we run the benchmarks several times and on different program instances and compare the results (note that we in the following present only the numbers of a representative benchmark run, which is for clarity reasons). This reduces the impact of prioritized processes, different JIT compiler results, and differently scheduled threads.

5.2.2 Execution Speed

To measure possible execution speed penalties, we took the following actions. We ran the PolePosition benchmark on HyperSQL immediately after runtime updating the application to version 1.8.0.10 and compared the results with the benchmark results of HyperSQL version 1.8.0.10 not updated at runtime. We could not measure any statistically significant difference (we run the test as described in Section 5.2.1 and always got comparable results), i.e, the benchmark results of the HyperSQL instance updated at runtime were as good as the results of the HyperSQL instance not updated at runtime. In other words, the dynamic updates performed by us did not affect the execution speed of HyperSQL in a measurable way.

Execution Speed in the Presence of Containers and Local References. Even if we did not measure runtime performance penalties because of our update approach in a real-world scenario, we assumed that our approach does not come entirely without runtime performance overhead. For instance, our container approach adds one level of indirection between reloaded class and caller and thus may cause performance penalties. To get evidence about this assumption, we additionally implemented a synthetic benchmark that is able to detect even minimal performance penalties. It measures the costs of crossing the version barrier from old program parts (i.e., callers) to the new ones (i.e., callees). In other words, the micro benchmark measures the time required to access the fields and methods of a reloaded callee. The complete set of callee accesses we measured is as follows:

$$\begin{aligned}\text{Access} \quad = \quad \{&\text{void meth}(),\ \text{void meth}(\text{primitive}),\ \text{primitive meth}(),\\ &\text{primitive meth}(\text{primitive}),\ \text{void meth}(\text{Object}),\ \text{Object meth}(),\\ &\text{Object meth}(\text{Object}),\ \text{void meth}(\text{Callee}),\ \text{Callee meth}(),\\ &\text{Callee meth}(\text{Callee}),\ \text{write Callee field},\ \text{read Callee field}\}\end{aligned}$$

To get reliable results, we repeatedly ran ten samples of one million invocations of all invocation types of set *Access* and for each calculated the average access time in nanoseconds. As a result, for none of the invocation types (which require containers and/or updates of references addressing short-lived objects) a statistically significant performance overhead was measurable, i.e., programs updated using containers and/or Java HotSwap perform as fast as the original program. One reason for the good results is the just-in-time compiler of the JVM that is able to optimize the code used to instrument the containers.

Execution Speed and Proxies. In Section 3.5.4, we described the need for proxies to avoid implicit caller replacements in case the callee appears to be an argument of a caller method, is returned by a caller method, or both. To figure out possible statistically significant execution speed penalties due to our proxy approach, we again repeatedly ran ten samples of one million (get-, set-, and set&get-) method invocations and recorded the method access times. Next, we analyzed the measured access times and found that the median access time was fast 0 nanoseconds, which is due to the optimizations done by the excellent JIT compiler of the JVM. But, we also noticed that in some cases accesses were not optimized and produced access times beyond the median. Moreover, we found those unoptimized accesses to be the reason why proxies cause performance penalties. Therefore, we did not only calculate the median and mean for the measured access times, but aimed at visualizing the unoptimized accesses (i.e., outliers), too, which in our case could be done best with box plots. In order to additionally visualize the concentration of overlapping outliers, we combine the box plots with sunflower plots.

Figure 5.2 shows the results of our proxy performance benchmark (in nanoseconds). With *No Update* (left part of Figure 5.2), we measured mean access times that range from 13,73 ns to 13,93 ns, with a median access time value of 0 ns and only 2,7 % to 3 % outliers. When we reload the *Callee* and thus have to use proxies, the average method access times increase (middle of Figure 5.2), now ranging from 38 ns to 53,5 ns, while the median is still at 0 ns. The reason for the increased mean values is that our proxy approach causes more outliers (i.e., 7,3 % – 9,4 %) and above that slower unoptimized accesses (see the box plots depicted in the middle of Figure 5.2). That is, dynamic updates involving proxies introduce slight execution speed penalties.

	Method	Median in ns	Mean in ns	Outliers in %
No Update	get	0	13,8	2,7
	set	0	13,93	3
	set&get	0	13,73	2,9
Callee	get	0	46	9,4
	set	0	38	7,3
	set&get	0	53,5	9,1
Caller	get	0	12,79	2,9
	set	0	15,61	3,3
	set&get	0	19,79	4,2

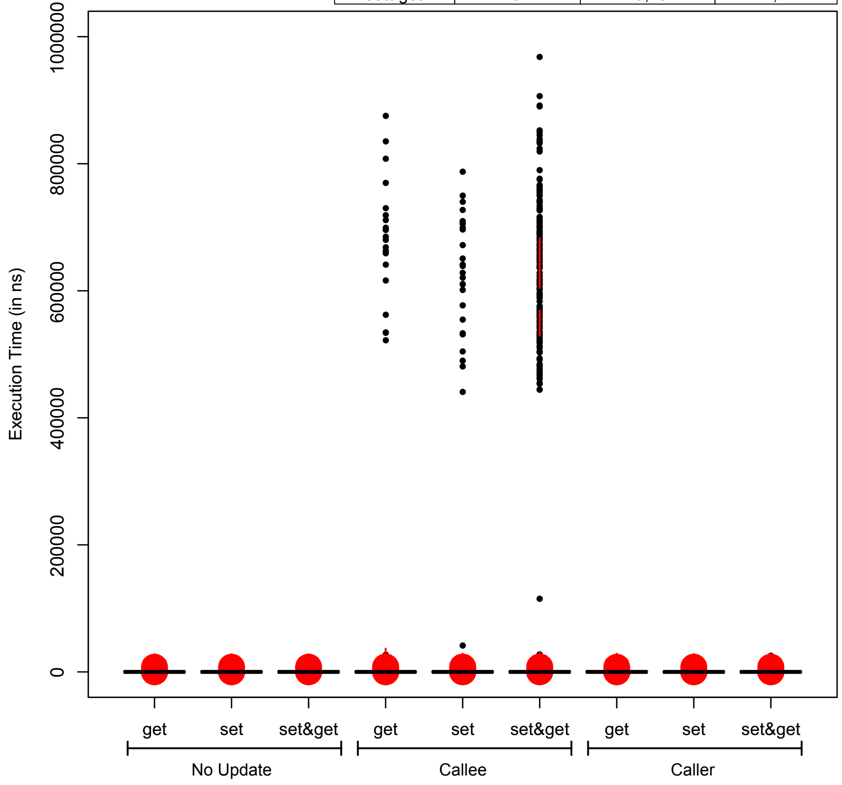
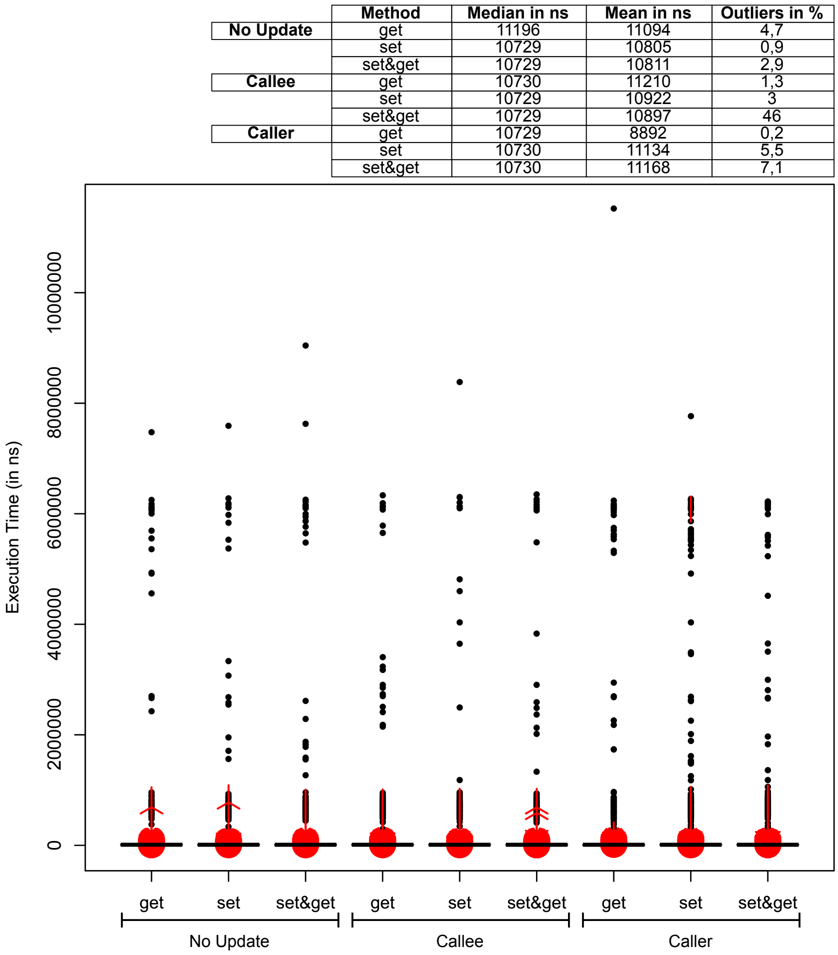


Figure 5.2: Method execution times in the presence of proxies. Meaning of the plotted elements: —— = box plot, ● = outlier, | = low concentration of overlapping outliers, ● = high concentration of overlapping outliers.

In order to get to know how the results scale, we put some workload on the methods and let them process statement `System.out.println("Hello JavAdaptor!")`. The results are shown in Figure 5.3. As one can see, the times to execute the method bodies are much higher than the pure method access times, which results in similar overall method execution times with and without proxies, ranging from 8892 ns to 11210 ns on average.

To sum up, performance penalties because of proxies are measurable, but workload on methods (which should be the common scenario) renders the performance penalties negligible. In addition, reloading the referring class (i.e., the *Caller*) as well, almost recovers the original method access times (see right part of Figure 5.2).



	Method	Median in ns	Mean in ns	Outliers in %
No Update	get	11196	11094	4,7
	set	10729	10805	0,9
	set&get	10729	10811	2,9
Callee	get	10730	11210	1,3
	set	10729	10922	3
	set&get	10729	10897	46
Caller	get	10729	8892	0,2
	set	10730	11134	5,5
	set&get	10730	11168	7,1

Figure 5.3: Method execution times in the presence of proxies and workload. Meaning of the plotted elements: —— = box plot, ● = outlier, | = low concentration of overlapping outliers, ● = high concentration of overlapping outliers.

Execution Speed and Recurring Updates. So far, we only examined the program execution speed after one single update step. But, in most real-world scenarios an application must be updated more than one time. One of the characteristics of our dynamic software update approach is, that we reload new class versions under a fresh name and with the same class loader that already loaded the original class. That is, older class versions remain in the JVM. Thus, we suspected the times to access reloaded classes to be growing with every update because the JVM has to manage the relationships between an increasing number of classes. Therefore, we additionally checked whether recurring dynamic software updates on the basis of JAVADAPTOR cause performance penalties.

To come to the point, times to access reloaded classes even after many updates remain

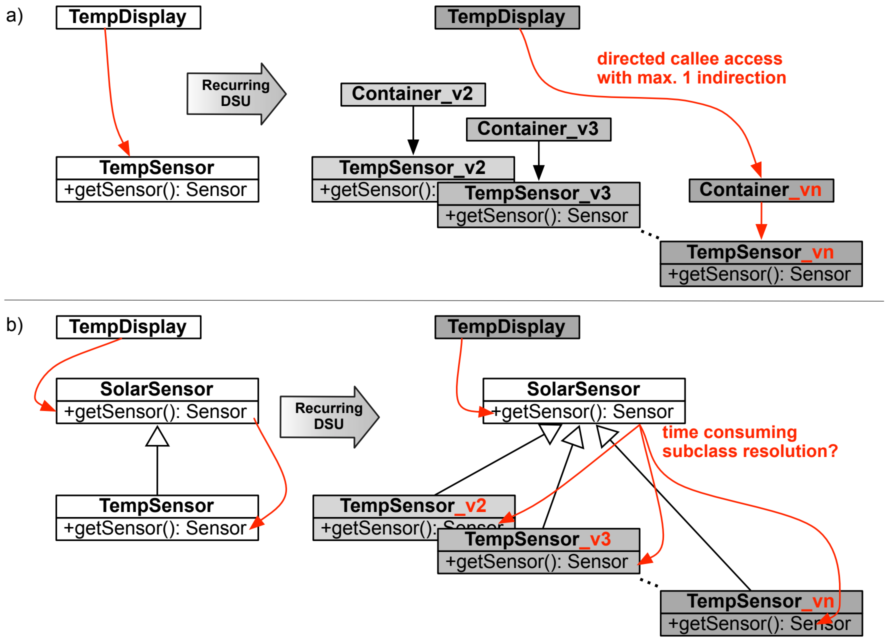


Figure 5.4: Recurring class reloadings and subclass resolution.

comparable to the access times with no update. The reason why the access times remain comparable is, that at any point in time and no matter how often the program was updated, the referring classes (such as class `TempDisplay` depicted in Figure 5.4a) can access the reloaded classes (such as class `TempSensor`, Figure 5.4a) directly with at most one additional indirection (caused by possible containers or proxies). Furthermore, references to outdated classes will be completely replaced with references to the up-to-date class, i.e., even if still loaded, the outdated classes will be not referenced anymore which reduces the effort to manage them.

Despite the above mentioned reasons for why JAVADAPTOR does not decrease the program execution speed after many updates, we assumed scenarios causing performance penalties where referring classes access reloaded classes via the reloaded classes's super type. Figure 5.4b sketches such a scenario. Here, referring class `TempDisplay` accesses method `getSensor` of class `TempSensor` via superclass `SolarSensor`. What seems to be problematic is that every class replacement (here of class `TempSensor`) adds a new subclass to the superclass (in our example to class `SolarSensor`), which may render the resolution of the up-to-date subclass version time-consuming after many class reloadings. In order to confirm our assumption, we repeatedly ran a benchmark accessing method `getSensor` of class `TempSensor` through superclass `SolarSensor` after 0, 1, 10, 100, 1000, and 10000 reloadings of class `TempSensor` (again with ten samples of one million invocations to get statistically significant results) and measured the method access times. As a result, the method access times remained unchanged even after 10000 class replacements confirming the JVMs capability to efficiently resolve up-to-date classes from a large set of possible candidates.

All in all, the results of the PolePosition benchmark on HyperSQL and our additional synthetic benchmarks confirm that dynamic software updates by JAVADAPTOR produce only minimal runtime performance overhead. Only proxies produce a measurable overhead. Whereas reference updates through local changes and containers do not cause measurable performance drops. Last but not least, the program execution speed remains unchanged even after many program updates.

5.2.3 Update Speed

Having evaluated how JAVADAPTOR affects the program execution speed, we measure how much time the tool needs to apply updates. As we already described, with our current JAVADAPTOR version we pause the application, while we apply the update. We do this for consistency reasons, i.e., in order to ensure that all classes within the JVM are up-to-date before we resume the application and to avoid state losses during state mapping because the JVM concurrently produces new objects not considered to be mapped. In order to figure out the update speed of JAVADAPTOR, we simply measure the time period the application must be paused. To get statistically significant results, we reran our tests 10 times and compared the measured update times. All runs produced comparable update times, which is why we subsequently discuss the results of one representative run.

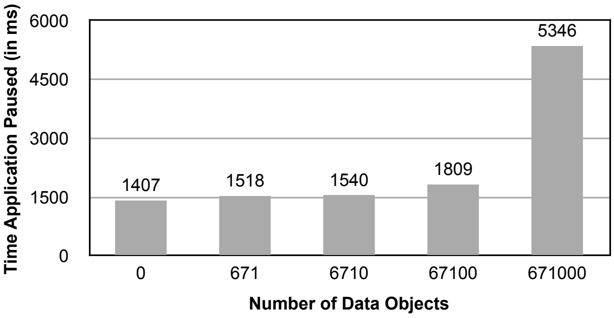


Figure 5.5: HyperSQL: Update speed.

At first, we measured the time required to update our HyperSQL case study under different conditions. With our first test, we measured the time period required to update HyperSQL with an empty database (i.e., without any data object stored), which was 1407 milliseconds (see Figure 5.5). In further tests, we ran the PolePosition benchmark creating 671, 6710, 67100, and 671000 of data objects before the update (note that the standard configuration of PolePosition creates 671000 data objects and we simply decreased those numbers stepwise by factor 10). As shown in Figure 5.5, the corresponding update times ranged from 1518 milliseconds to 5346 milliseconds. This seems to be not extremely fast but sufficient in many scenarios. By contrast, restarts and reinitializations of HyperSQL (e.g., filling caches, reloading data objects, creating views, creating users, etc.), as we simulated them using PolePosition, took more time. More precisely, the times ranged from 4339 ms (for 671 data objects) to 88067 ms (for 671000 data objects).

The other application for which we measured the update times, was *Snake*. Compared to the update of HyperSQL, which affects wide parts of the system (the update spans changes made during 9 months of development), each Snake update step consists only of small changes to few classes. Thus, the Snake updates represent scenarios common to the software development process, i.e., frequent minor changes and immediate application of the changes. As our demo video (available on YouTube[5]) suggests, the update times are rather short. The exact update times for one representative run (we again reran the update ten times and always got similar results) ranged from 28 milliseconds to 142 milliseconds.

All in all, the update times we measured suggest that our current JAVADAPTOR implementation could be beneficial in many different scenarios. The bottleneck of the current implementation is JVMTI method `referringObjects`, which JAVADAPTOR uses during state mapping to identify the callers of an outdated object. The execution times of this method notably increase the more objects are present in the JVM, even if the number of objects to be updated remains unchanged. We do not know if the performance of method `referringObjects` will be improved with future JVM versions. Therefore, we are currently working on a JAVADAPTOR version which avoids to use this method and thus offers dramatically improved update speeds (we will discuss possible improvements of JAVADAPTOR and first benchmark results in Chapter 6).

5.3 Platform Independence

In Section 1, we argued that platform independence is crucial because forcing developers to use specific JVMs contradicts the idea of Java (i.e., its platform independence). Even if the case studies and the implementational details given in Section 4 suggest that JAVADAPTOR satisfies this criterion, we want to underpin this impression with some data and arguments.

To confirm that JAVADAPTOR meets the platform independence criterion, we ran JAVADAPTOR on top of all major standard (certified) JVM versions publicly available and successfully updated Snake at runtime. We preferred Snake over HyperSQL and our refactoring case study, because it gives immediate (visual) feedback about the correctness of the update and (even if a small case study) covers all kinds of updates essential to flexibly update running applications. Table 5.3 lists all tested JVMs and the Java version we used to confirm that

[5] `http://www.youtube.com/watch?v=jZm0hvlhC-E`

JVM Name	Availability	Java Version	
		JDK 6	JDK 7
Oracle HotSpot VM 32/64-bit	Windows, Linux, Solaris	confirmed	confirmed
OpenJDK HotSpot VM 32/64-bit	Windows, Linux, Solaris, Mac OS	confirmed	confirmed
Apple HotSpot VM 32/64-bit	Mac OS	confirmed	confirmed
Oracle JRockit VM 32/64-bit	Windows, Linux, Solaris	confirmed	not available
IBM VM 32/64-bit	Linux, AIX, z/OS	confirmed	confirmed

Table 5.3: Tested platforms.

JAVADAPTOR works properly. We point out that JAVADAPTOR straightaway worked with all JVMs without any modification.

5.4 Architecture Independence

The last criterion we claimed to support with JAVADAPTOR is architecture independence, which is important because different application scenarios require different program architectures.

Similar to the flexibility criterion, it is virtually impossible to conclude the discussion regarding JAVADAPTOR's architecture independence. Nevertheless, our case studies already gave some insights to this question. For example, with Snake and our refactoring programs we successfully updated standalone applications. By contrast, with HyperSQL we successfully updated an application which is part of a server client architecture, whereas HyperSQL was the server and PolePosition the client. We point out that we neither prepared JAVADAPTOR for the one or the other architecture, i.e., JAVADAPTOR naturally supports both architectures. JAVADAPTOR was even able to successfully update a component-based version of Snake (i.e., a version engaging different class loaders). The reason why JAVADAPTOR supports many different architectures is that at JVM level it is all about class loaders, classes, methods, fields, objects, and how they relate to each other, which is the same for all programs no matter what architecture they base on. There are of course borderline cases not (yet) fully supported by JAVADAPTOR, such as program architectures that may engage not only Java but other languages, too (e.g., CORBA). However, we do not see general problems with the concept of JAVADAPTOR which may circumvent support for specific architectures.

5.5 Memory Consumption

Even if not part of the thesis goals, we additionally analyzed the memory consumption of JAVADAPTOR.

Our current JAVADAPTOR implementation loads a new class version with the same class loader that already loaded the original class. That is, even if we do no longer need the original class to be present within the JVM, we cannot unload it, because the class's loader responsible for the original class is still deployed (as we explained in Section 2.4, class unloading requires the corresponding class loader to be undeployed). In earlier sections, we already examined if the outdated classes still present within the JVM degenerate program execution speed after many updates, which is not the case. Even if we did not measure any execution speed penalties, we cannot ignore the fact that the still loaded outdated class versions waste the JVM's memory space and large memory footprints because of this may hinder the usage of JAVADAPTOR on systems with less (main) memory. Therefore, we analyzed the JVM's memory consumption after many updates. Our evaluation is based on the same benchmark that we used to figure out possible execution speed penalties due to frequent subclassing (see Figure 5.4). That is, we reload class `TempSensor` of our benchmark 1, 10, 100, 1000, and 10000 times and for each test check how much memory the JVM consumes. The class-file size of class `TempSensor`, was 1098 Bytes. To get statistically significant results, we reran the tests 10 times and compared the measured memory consumption numbers. The measured memory footprint remained comparable across all runs.

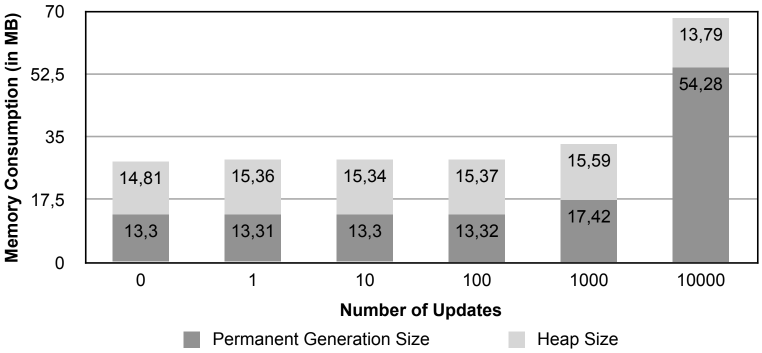


Figure 5.6: Memory consumption vs. recurring updates.

In Figure 5.6, we show the memory consumption of one representative run. The size of the heap (amongst others, responsible for storing objects) remains virtually unchanged (15,04 MB on average) which is no surprise, because JAVADAPTOR does only restrict the unloading of outdated classes but not the deletion of unused objects. By contrast, the size of the *Permanent Generation* (stores all class related information) was increased by 40,98 MB after 10000 reloadings of class `TempSensor`.

Even if in many real-world scenarios the classes scheduled for reloadings may have larger file sizes and thus consume more memory, we classify the memory consumption of JAVADAPTOR after recurring updates to be no problem, because modern JVMs are able to handle very large storage capacities.

5.6 Summary

Our goal was the development of a highly flexible, performant, platform independent, architecture independent, and fine-grained approach for dynamic software updates. Within this chapter, we evaluated whether JAVADAPTOR meets the goal. We demonstrated the practicability, flexibility, generality, and fine update granularity of our tool on the basis of three different non-trivial case studies (i.e., HyperSQL, Snake, and the Refactoring case study). We further measured the runtime performance of JAVADAPTOR. As a result, solely our proxies to avoid schema changes of referring classes introduce slight performance penalties. The runtime performance remains unchanged even after many updates (i.e., class reloadings). Furthermore, we evaluated the update speed of our current JAVADAPTOR implementation and found that it could be improved, but is sufficient in many different scenarios. In order to illustrate JAVADAPTOR's platform independence, we applied it to all publicly available standard JVM's. JAVADAPTOR instantly worked with all tested JVM's without any modification. Next, we argued why JAVADAPTOR is architecture independent. Even if not central part of our contribution, we last but not least measured the memory consumption of JAVADAPTOR after many updates. Our measurements show that the footprint is acceptable.

6 Enhancements and Optimizations

This chapter shares material with the SPE'2011 paper "JavAdaptor – Flexible Runtime Updates of Java Applications" [PKC$^+$12].

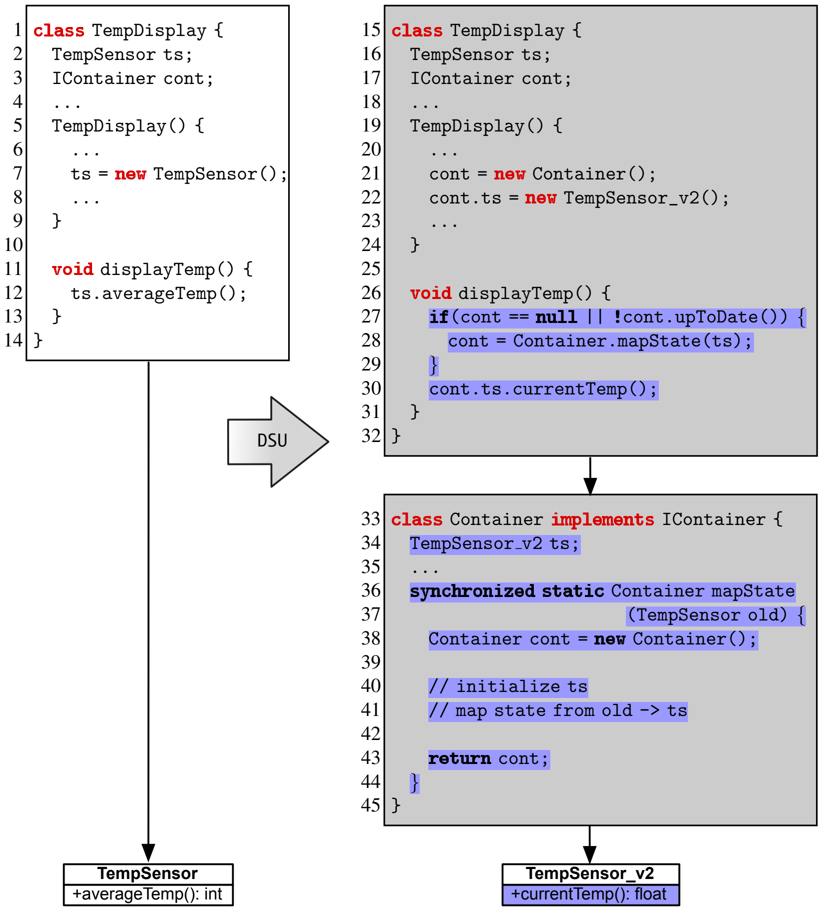
Within the previous chapter, we analyzed whether JAVADAPTOR meets the thesis goals. Even if the results of our evaluation confirm the fulfillment of the goals, there is still space for improvements.

With this chapter, we summarize work in progress to improve JAVADAPTOR. We point out that most of the here discussed improvements are inspired by existing work, such as presented by Kim [Kim09], or Gregersen [Gre10b]. However, we do not simply discuss related work, but describe how to combine it with the existing JAVADAPTOR concept. We start with descriptions on how to improve the update speed of our tool and present preliminary results of our efforts regarding this. Furthermore, we discuss possible tool enhancements towards consistent dynamic software updates. More precisely, we sketch how to achieve thread-safe updates, prevent (rarely) possible state losses, deal with binary-incompatible updates and improve support for reflective calls. Last but not least, we summarize the long-term objectives we pursue with JAVADAPTOR.

6.1 Update-Speed Improvements

In Section 5.2.3, we evaluated the update speed of JAVADAPTOR on the basis of our HyperSQL and Snake case studies. We found the current JAVADAPTOR implementation acceptable fast in this regard but stated that it could be further improved. From what we found out, the bottleneck of our current JAVADAPTOR implementation is JVMTI method `referringObjects`, which helps us to identify all objects referring to outdated objects. The problem with method `referringObjects` is, that it performs a full heap search every time we request the referring objects of an outdated object, which causes long program update times and thus long time periods of program unavailability if the program heap is large and/or many requests must be processed.

One naive solution to reduce the time periods of program unavailability might be to not pause the application and update the caller side while the program still provides its services. However, this may lead to wrong program behavior because objects of an old class created after the execution of method `getReferringObjects` may be not updated or objects to be updated may be garbage collected meanwhile. Due to the fact that wrong program behavior would challenge the benefits of dynamic software updates, we have to look for another solution.



```
class TempDisplay {
  TempSensor ts;
  IContainer cont;
  ...
  TempDisplay() {
    ...
    ts = new TempSensor();
    ...
  }

  void displayTemp() {
    ts.averageTemp();
  }
}
```

```
class TempDisplay {
  TempSensor ts;
  IContainer cont;
  ...
  TempDisplay() {
    ...
    cont = new Container();
    cont.ts = new TempSensor_v2();
    ...
  }

  void displayTemp() {
    if(cont == null || !cont.upToDate()) {
      cont = Container.mapState(ts);
    }
    cont.ts.currentTemp();
  }
}
```

```
class Container implements IContainer {
  TempSensor_v2 ts;
  ...
  synchronized static Container mapState
                           (TempSensor old) {
    Container cont = new Container();

    // initialize ts
    // map state from old -> ts

    return cont;
  }
}
```

Figure 6.1: Lazy state mapping. Different from the state mappings through method `getReferringObjects`, we now map the state on a per-access basis.

The better alternative to the previously sketched solution are *lazy state mappings* as Kim [Kim09] and Gregersen [Gre10b] use them in their DSU approaches. Different from our current implementation, in which we map the state and update the referring program parts in one atomic step, lazy state mappings operate on a per-access basis. That is, the state transfer between the outdated and up-to-date object and the update of the referring program parts is carried out from within the program if and only if an outdated object is accessed.

Figure 6.1 exemplifies how lazy state mappings work and how we are going to integrate them into JAVADAPTOR. In order to dynamically change our small weather station program such that it computes and displays *current* instead of the *average* temperatures, JavAdaptor updates the running program as follows. It processes all update steps we described in Chapters 3 and 4, but applies additional code to the program, which carries out the state mapping and updates the referring program parts without the need of method `referringObjects`. More precisely, JavAdaptor modifies the program code in such way that before each access to a potentially outdated object, it will be checked whether the object must be updated or not. In the example depicted in Figure 6.1, this applies to all references to field `ts`, which we must update using our container approach because we replaced class `TempSensor` with class version `TempSensor_v2` in order to add new method `currentTemp`. Concretely, before we access the up-to-date object (here of class `TempSensor_v2`) stored in the container, we check whether the container object already exists and is up-to-date or not (see Figure 6.1, Line 27). In the latter case, a mapping method (in our example method `mapState`) of the container class will be called (Figure 6.1, Line 28). This method maps the state from outdated object (here of type `TempSensor`) to the up-to-date object (i.e., of type `TempSensor_v2`), applies the newly created object to a container instance, and returns the container instance (see Figure 6.1, Lines 36 – 44). After the state mapping, the newly created object can be accessed as usual, i.e., via the container instance (see Line 30).

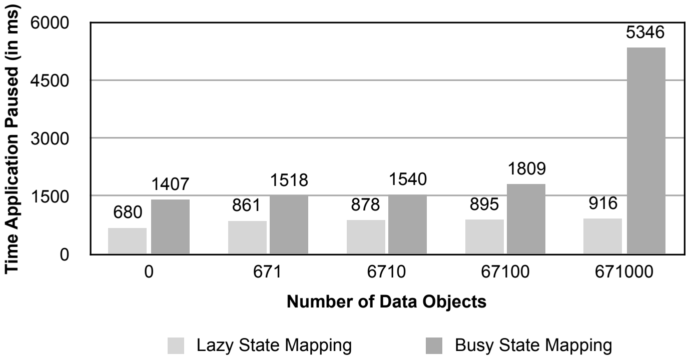


Figure 6.2: Update speed of HyperSQL: Lazy vs. busy state mapping.

After we described how we could provide our tool with lazy state mappings, let us present some update-speed numbers confirming that lazily mapping the state and thus avoiding to use method `referringObjects` significantly improves the update speed of JavAdaptor. In Section 5.2.3 we measured the update times of JavAdaptor regarding our HyperSQL case study with zero, hundreds, thousands, ten thousands, and hundred thousands of data objects. The numbers ranged from 1407 to 5346 milliseconds. With a JavAdaptor prototype which provides lazy state mappings as we sketched them in Figure 6.1, we were able to significantly reduce the update-speed times. Figure 6.2 contrasts the old update-speed times with the new ones based on lazy state mappings. What can be seen is that the update-speed numbers remain somewhat comparable as long as only few objects are on the heap of the JVM. But, in case of many objects on the heap (here hundred thousands of data objects), JavAdaptor based on lazy state mappings clearly outperforms our current (i.e., *busy*) state mapping implementation, i.e, the prototype requires to pause the application only 916 milliseconds, whereas current JavAdaptor causes an application pause time of 5346 milliseconds.

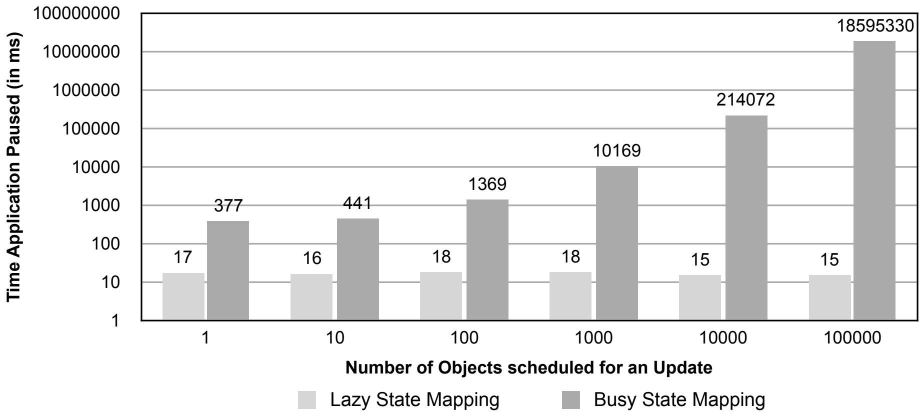


Figure 6.3: Update speed in dependence to the number of objects to be updated: Lazy vs. busy state mapping.

The numbers presented in Figure 6.3 further underpin the benefit of lazy state mappings. Different from our HyperSQL case study, where the number of objects to be updated remained unchanged with each benchmark configuration, the here presented results outline how the application pause times develop depending on the number of objects scheduled for an update. As shown in Figure 6.3, the application pause times caused by our current JavAdaptor implementation further increase dependent on the number of objects to be updated, which is because with each object update JavAdaptor must call method `referringObjects`. By contrast, the application pause times of our JavAdaptor prototype based on lazy state mappings are significantly shorter and moreover, remain virtually unchanged regardless of the number of objects that require an update.

Because the tests with our prototype show significant update-speed improvements, we are currently working to complete the integration of lazy state mappings into JavAdaptor. What is still missing, is support for lazy state mappings within the Java system classes. Nevertheless, we are optimistic to provide a fully working JavAdaptor version with lazy state mappings soon.

6.2 Solutions Toward Consistent Program Updates

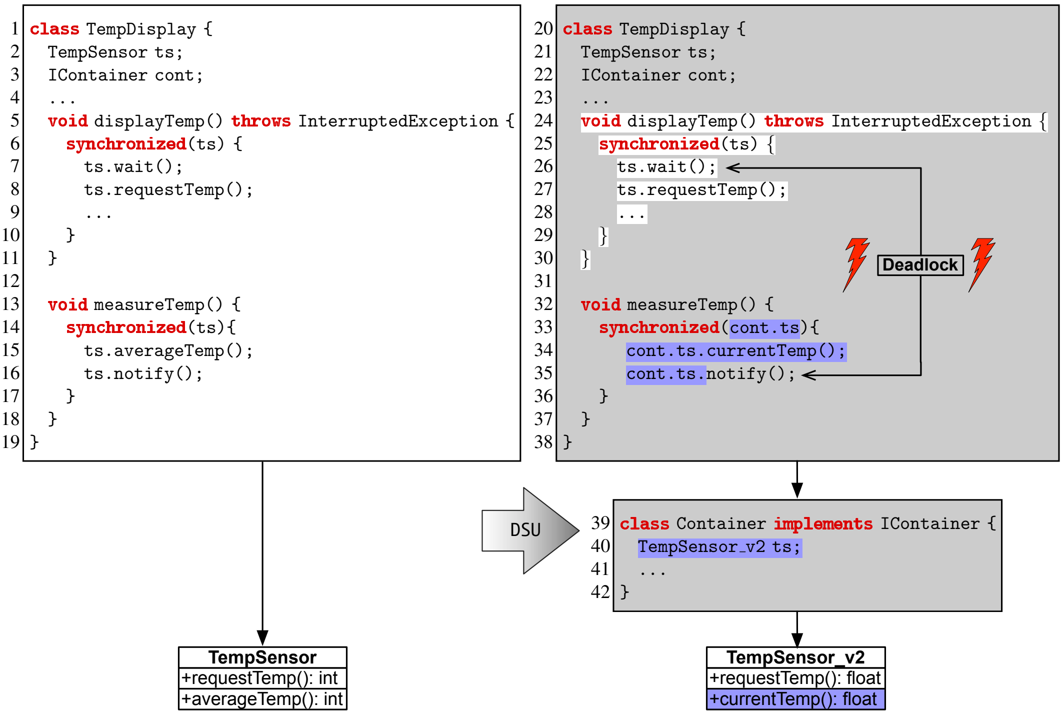
The HyperSQL as well as the Snake case study show that JAVADAPTOR could update programs without compromising their correctness, i.e., the programs consistency. This is, because JAVADAPTOR already includes mechanisms aiming at consistent program updates. For instance, JAVADAPTOR permits updates only if the program sources compile without errors. Another example is, that JAVADAPTOR pauses the application during the update in order to ensure that all changed program parts are present within the JVM. However, as other DSU approaches, the current JAVADAPTOR implementation does not ensure program consistency at all beyond the update. Therefore, we discuss how to improve our tool in this regard.

6.2.1 Thread-Safe Updates

One issue, we plan to tackle with future JAVADAPTOR versions is the lack of support for thread-safe updates of multi-threaded applications. Currently, updates of multi-threaded applications may cause deadlocks and thus inconsistencies under certain conditions. Such a scenario is depicted in Figure 6.4. In the example, two different threads alternately access `TempSensor ts` of class `TempDisplay` of our small weather station. The first thread periodically instructs `ts` to measure the temperature (by calling method `measureTemp`), whereas the second thread is responsible for displaying the measured temperature (by calling method `displayTemp`). Because measuring and displaying the temperature at the same time would cause unexpected program behavior, access to `TempSensor ts` must be synchronized (see Figure 6.4 Lines 6 – 10 and Lines 14 – 17).

What could happen when JAVADAPTOR updates a multi-threaded application such as shown in Figure 6.4 (note that for clarity reasons the lazy state mapping related code is hidden) is that for some methods the necessary method body redefinitions already took effect, while other methods remain unaffected, which is due to the principles of Java HotSwap (remember that method body redefinitions would not affect methods active on the stack at the moment of redefinition). In our example (see right side of Figure 6.4), method `measureTemp` (Lines 32 – 37) is already redefined and thus refers to an object of up-to-date class version `TempSensor_v2`, whereas method `displayTemp` (Lines 24 – 30) is still active on the stack with the old method body referring to outdated `TempSensor ts`. What appears to be the problem here is that method `notify` (Figure 6.4, Line 35) would not activate the thread executing method `displayTemp` because method `notify` is executed on a different object. In other words, we have a deadlock.

One solution for the described problem would be to postpone the update until no method scheduled for redefinition is active on the stack. However, particularly when it comes to updates of long-running methods, this condition may be never fulfilled or it takes a long time



```
class TempDisplay {
  TempSensor ts;
  IContainer cont;
  ...
  void displayTemp() throws InterruptedException {
    synchronized(ts) {
      ts.wait();
      ts.requestTemp();
      ...
    }
  }

  void measureTemp() {
    synchronized(ts){
      ts.averageTemp();
      ts.notify();
    }
  }
}
```

```
class TempDisplay {
  TempSensor ts;
  IContainer cont;
  ...
  void displayTemp() throws InterruptedException {
    synchronized(ts) {
      ts.wait();
      ts.requestTemp();
      ...
    }
  }

  void measureTemp() {
    synchronized(cont.ts){
      cont.ts.currentTemp();
      cont.ts.notify();
    }
  }
}
```

```
class Container implements IContainer {
  TempSensor_v2 ts;
  ...
}
```

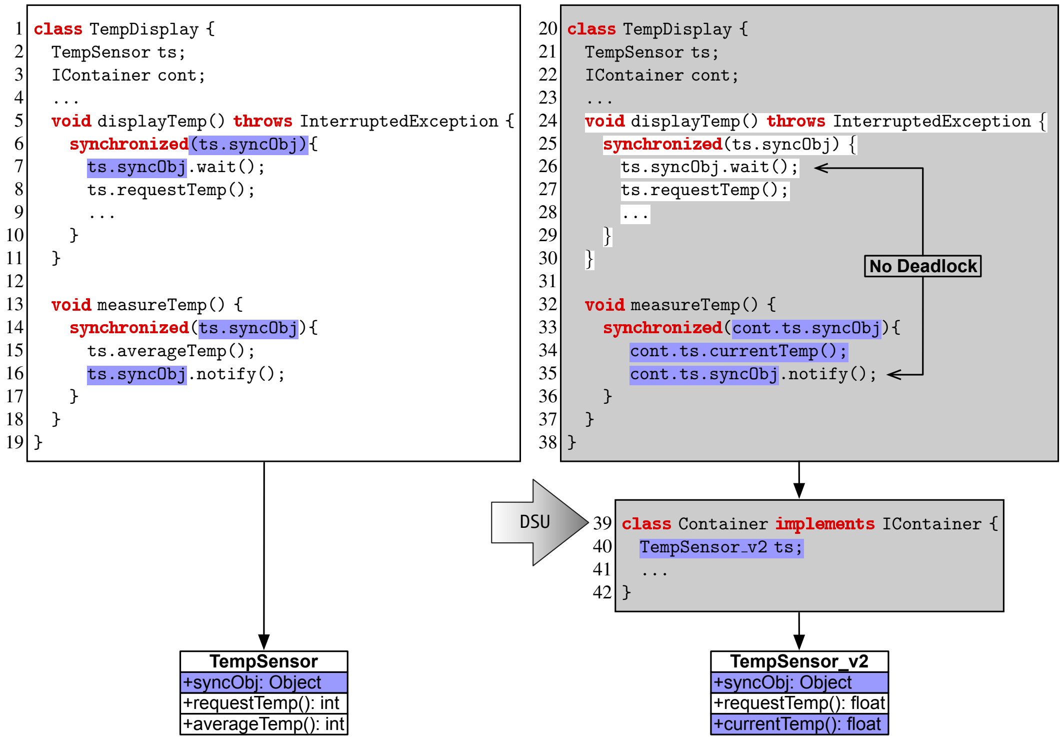
Figure 6.4: Deadlocks because of dynamic software updates.

until it comes true. Therefore, we have to find another strategy.

To prevent deadlocks in multi-threaded applications such as sketched above, Gregersen proposes special synchronization objects that could be shared beyond different class versions [Gre10b]. Figure 6.5 shows how those synchronization objects could be applied to JAVADAPTOR. Here, we add an additional field `syncObj` of type `Object` to class `TempSensor`, which, instead of the `TempSensor` object itself, is used for synchronization (see Figure 6.5, Lines 6 and 14). If the application must be updated and the necessary method body redefinitions take effect for one method (in our example for method `measureTemp`, see Figure 6.5, Lines 32 – 37), but not for the other (compare method `displayTemp`, Figure 6.5, Lines 24 – 30), no deadlock occurs. This is, because the outdated object (here of type `TempSensor`) and its up-to-date counterpart (in our example an object of type `TempSensor_v2`) share the same synchronization object (i.e., object `syncObj`).

6.2.2 State-Loss Prevention

Another shortcoming of our current JAVADAPTOR implementation (even true for our prototype based on lazy state mappings) is that it may cause program inconsistencies because of state losses. To illustrate the problem, we use a slightly different version of our small weather station program to be updated at runtime (see Figure 6.6). Here, we again have



```
class TempDisplay {
  TempSensor ts;
  IContainer cont;
  ...
  void displayTemp() throws InterruptedException {
    synchronized(ts.syncObj){
      ts.syncObj.wait();
      ts.requestTemp();
      ...
    }
  }

  void measureTemp() {
    synchronized(ts.syncObj){
      ts.averageTemp();
      ts.syncObj.notify();
    }
  }
}
```

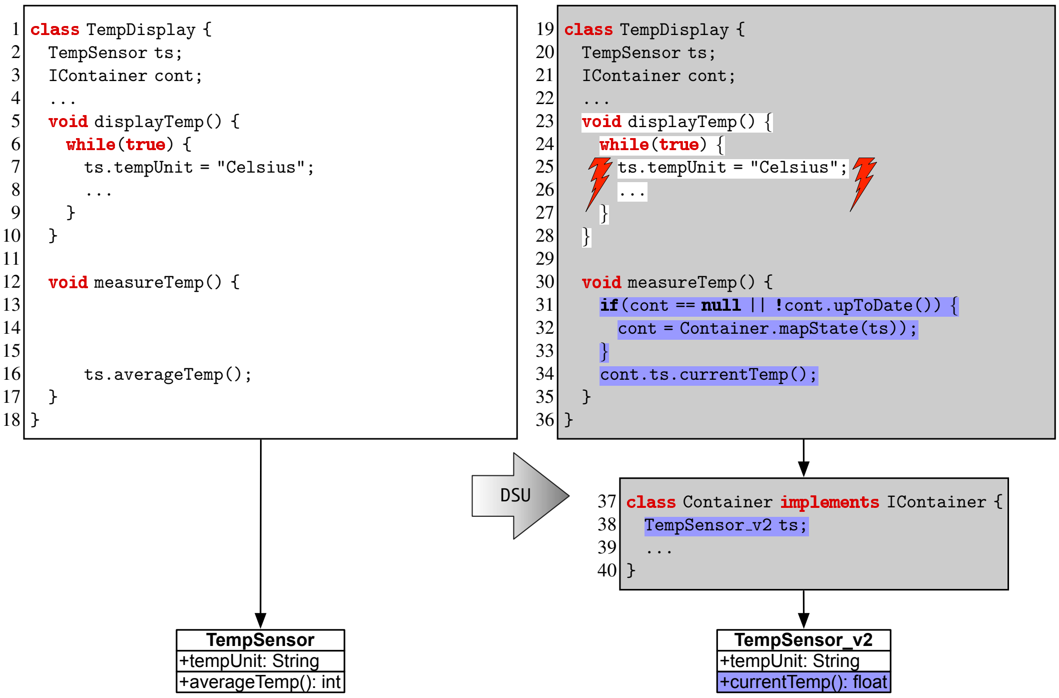
```
class TempDisplay {
  TempSensor ts;
  IContainer cont;
  ...
  void displayTemp() throws InterruptedException {
    synchronized(ts.syncObj) {
      ts.syncObj.wait();
      ts.requestTemp();
      ...
    }
  }

  void measureTemp() {
    synchronized(cont.ts.syncObj){
       cont.ts.currentTemp();
       cont.ts.syncObj.notify();
    }
  }
}
```

```
class Container implements IContainer {
  TempSensor_v2 ts;
  ...
}
```

Figure 6.5: Deadlock prevention through shared synchronization objects.

the situation that, for one method (i.e., method `measureTemp`) the necessary method body redefinition through Java HotSwap took effect, while the other method (in our example method `displayTemp`) is still active on the stack with the old method body. Now it could be the case, that the outdated method remains active on the stack while the state of the referred outdated object (in our case the `TempSensor` object referred by `ts`) is already mapped to an object of the new class version (here of type `TempSensor_v2`), because another thread executed the redefined method including the state mapping related code (see Figure 6.6, method `measureTemp`, Lines 31 – 33). The problem is that the still active outdated method may change the state of the outdated referred object (such as sketched in Line 25 of Figure 6.6) and because the state transfer already happened, those state changes would be lost on the new object.

A first naive solution for the problem depicted in Figure 6.6 would be to manually pop all active methods to be redefined from the stack before redefinition, which is supported by the JVM. This would prevent executions of old method versions and thus accesses to outdated objects after the state transfer causing state losses. Unfortunately, the JVM only allows us to manually pop the outdated methods from the stack, but not to manually push their up-to-date (redefined) counterparts back on the stack. Because we cannot push the redefined methods



```
 1 class TempDisplay {
 2   TempSensor ts;
 3   IContainer cont;
 4   ...
 5   void displayTemp() {
 6     while(true) {
 7       ts.tempUnit = "Celsius";
 8       ...
 9     }
10   }
11
12   void measureTemp() {
13
14
15
16       ts.averageTemp();
17   }
18 }
```

```
19 class TempDisplay {
20   TempSensor ts;
21   IContainer cont;
22   ...
23   void displayTemp() {
24     while(true) {
25       ts.tempUnit = "Celsius";
26       ...
27     }
28   }
29
30   void measureTemp() {
31     if(cont == null || !cont.upToDate()) {
32       cont = Container.mapState(ts));
33     }
34     cont.ts.currentTemp();
35   }
36 }
```

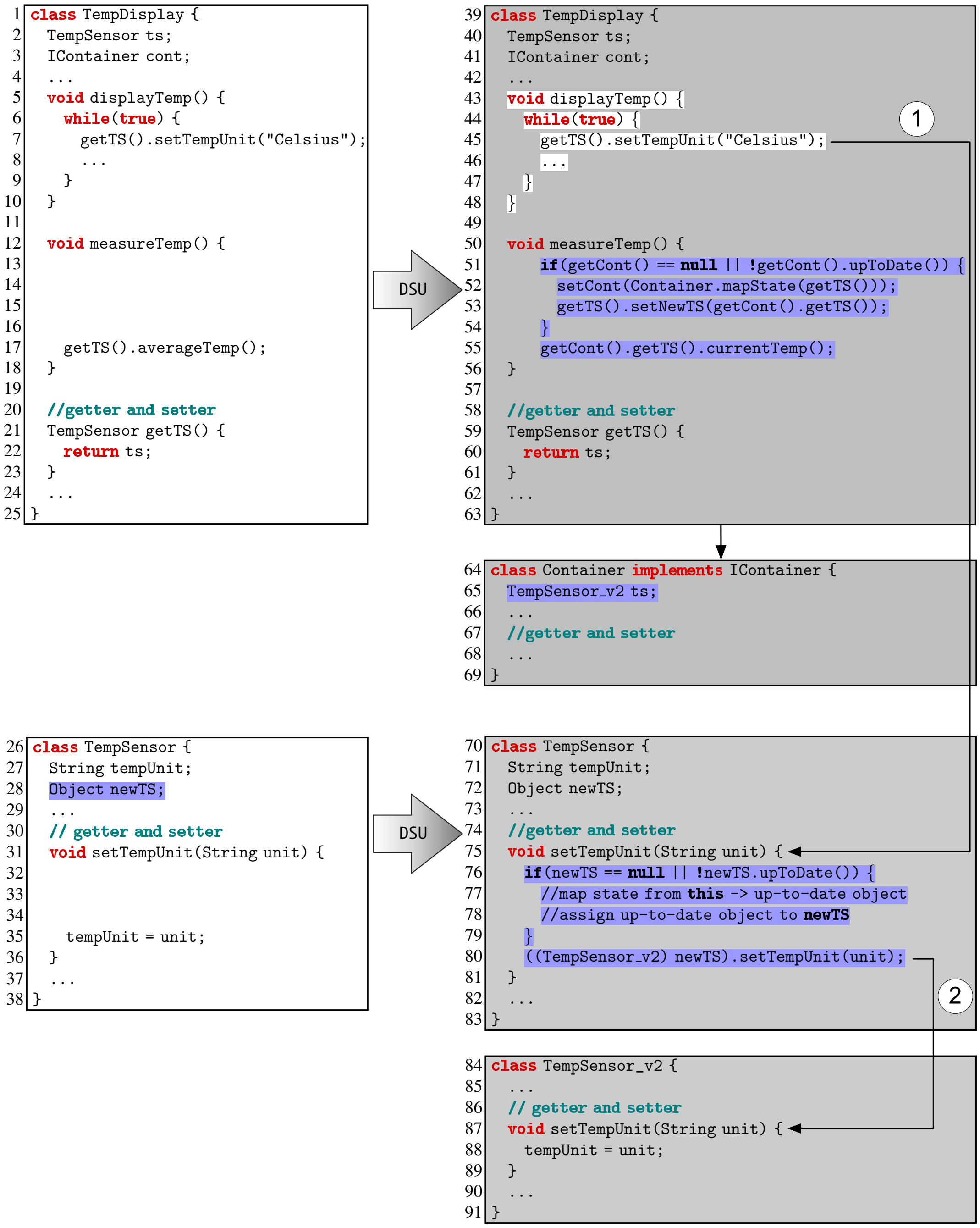
```
37 class Container implements IContainer {
38   TempSensor_v2 ts;
39   ...
40 }
```

Figure 6.6: State losses because of dynamic software updates.

back on the stack, the program's natural control flow will be disordered, which potentially results in wrong program behavior.

What may be the better strategy compared to pop operations is to intercept the access to an outdated object and to redirect this access to the corresponding up-to-date object. The challenge is, that the interception and redirection of direct object accesses (such as depicted in Line 25 of Figure 6.6) is not possible, because of the missing indirection between caller and callee required to hook into the access path. The solution for this problem is delivered by Fowler [Fow06] who argues that, compared to direct accesses, getter and setter methods allow us to flexibly manage accesses to objects.

Figure 6.7 shows how we plan to use getter and setter methods to prevent state losses because of redefinitions of active methods. Here, again method `displayTemp` scheduled for redefinition is active on the stack (see Lines 43 – 48 of Figure 6.7), while the redefinition of method `measureTemp` already took effect (Lines 50 – 56). Only difference to the example depicted in Figure 6.6 is that we now access all objects, especially the outdated object of type `TempSensor` referenced by field `ts`, via getter and setter methods (e.g., see Line 45 of Figure 6.7). To redirect object accesses from within outdated active methods to the up-to-date object, we redefine all methods of old class versions (in our example method `setTempUnit` of old class version `TempSensor`, Line 45) referenced by the outdated method as follows (see



```
class TempDisplay {
  TempSensor ts;
  IContainer cont;
  ...
  void displayTemp() {
    while(true) {
      getTS().setTempUnit("Celsius");
      ...
    }
  }

  void measureTemp() {




    getTS().averageTemp();
  }

  //getter and setter
  TempSensor getTS() {
    return ts;
  }
  ...
}
```

```
class TempDisplay {
  TempSensor ts;
  IContainer cont;
  ...
  void displayTemp() {
    while(true) {
      getTS().setTempUnit("Celsius");
      ...
    }
  }

  void measureTemp() {
      if(getCont() == null || !getCont().upToDate()) {
        setCont(Container.mapState(getTS()));
        getTS().setNewTS(getCont().getTS());
      }
      getCont().getTS().currentTemp();
  }

  //getter and setter
  TempSensor getTS() {
    return ts;
  }
  ...
}
```

```
class Container implements IContainer {
  TempSensor_v2 ts;
  ...
  //getter and setter
  ...
}
```

```
class TempSensor {
  String tempUnit;
  Object newTS;
  ...
  // getter and setter
  void setTempUnit(String unit) {



    tempUnit = unit;
  }
  ...
}
```

```
class TempSensor {
  String tempUnit;
  Object newTS;
  ...
  //getter and setter
  void setTempUnit(String unit) {
    if(newTS == null || !newTS.upToDate()) {
      //map state from this -> up-to-date object
      //assign up-to-date object to newTS
    }
    ((TempSensor_v2) newTS).setTempUnit(unit);
  }
  ...
}
```

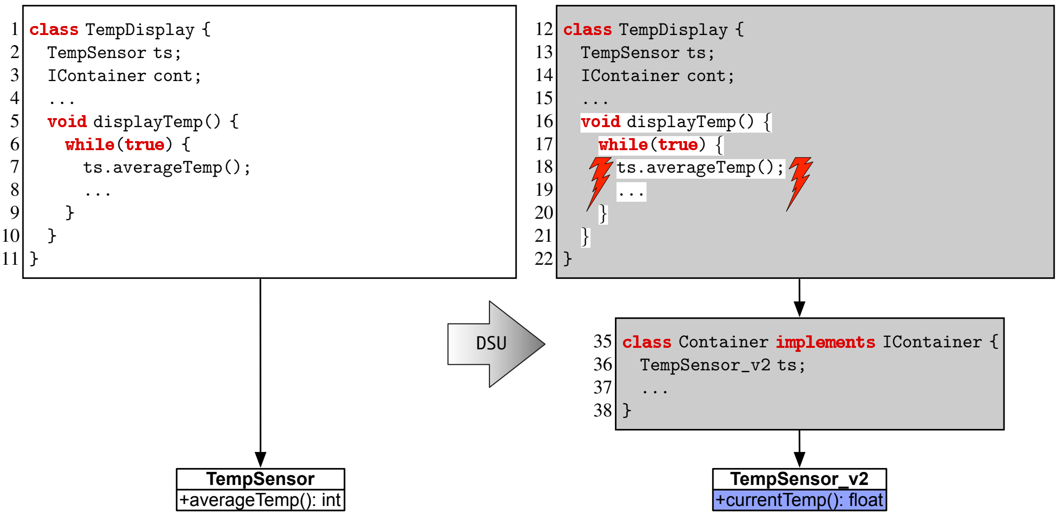
```
class TempSensor_v2 {
  ...
  // getter and setter
  void setTempUnit(String unit) {
    tempUnit = unit;
  }
  ...
}
```

Figure 6.7: State-loss prevention.

Lines 76 – 80, Figure 6.7). First of all, we check whether the state mapping already took place (Line 76), e.g., because of the execution of an up-to-date method (such as in our example method measureTemp, see Lines 51 – 54). In case the state mapping is pending, we process the state mapping (Line 77). Next, we couple the outdated and the up-to-date object by assigning the up-to-date object to a field of the outdated object (Line 78). Note that the field refers to the same object as the applied container, which ensures that outdated active method as well as up-to-date method access the same object. Finally, we forward the method call to the method of the up-to-date object (Line 80). After this is done, every access to a field of an outdated object from within an active outdated method (e.g., see Access 1, Figure 6.7) will be redirected to the corresponding up-to-date object (such as through Access 2 shown in Figure 6.7) and no state will be lost.

6.2.3 Handling of Binary-Incompatible Updates

So far, we discussed how getter and setter methods in conjunction with redefinitions of methods of outdated class version can help us to prevent state losses because of active methods scheduled for redefinition. But, getters, setters, and redefinitions of old methods could do a lot more for us. Coming back to our motivating example, where we are going to remove method averageTemp by method currentTemp and therefore have to replace class TempSensor and update calling class TempDisplay, conflicts such as depicted in Figure 6.8 can occur. As in the previous examples, method displayTemp to be redefined is active on the stack with the old method body. What is the problem here is that the method continues to call method averageTemp even if this method is removed in new class version TempSensor_v2, which is referred to as a *binary-incompatible update* [GJSB05].



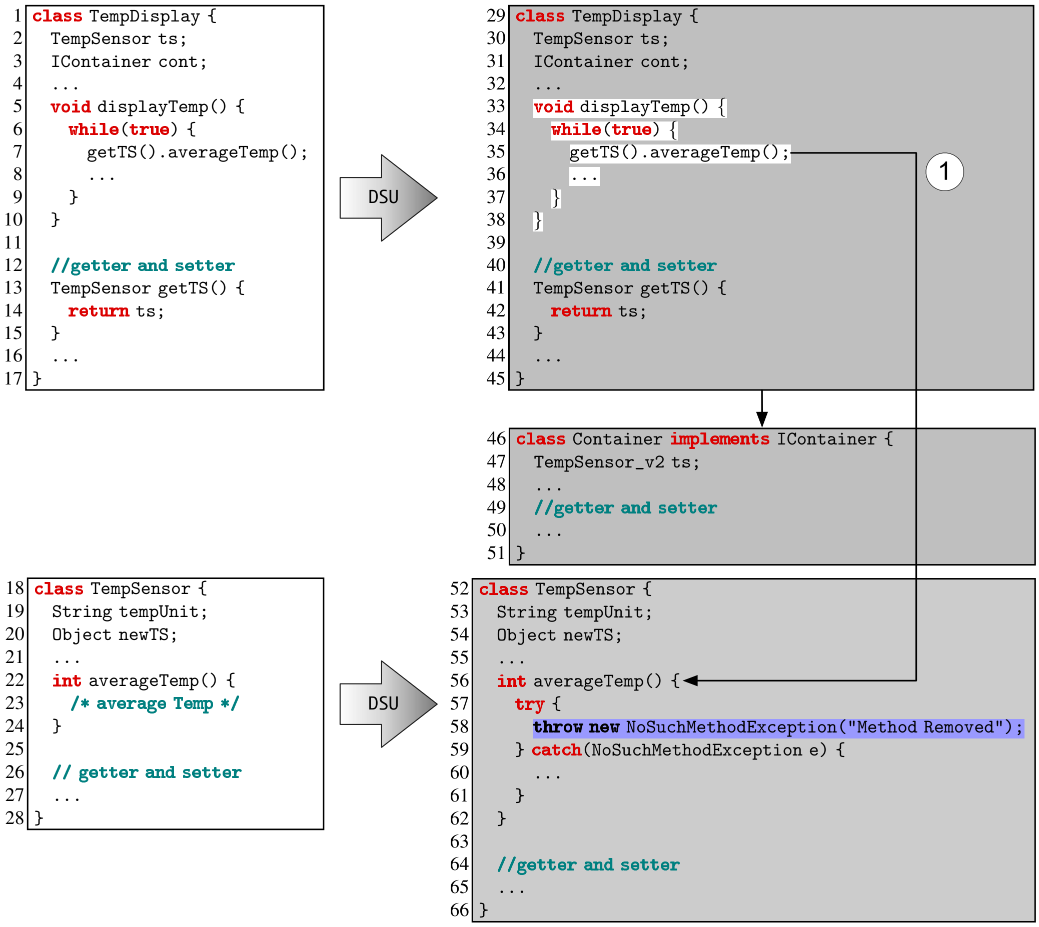
```
class TempDisplay {
  TempSensor ts;
  IContainer cont;
  ...
  void displayTemp() {
    while(true) {
      ts.averageTemp();
      ...
    }
  }
}
```

```
class TempDisplay {
  TempSensor ts;
  IContainer cont;
  ...
  void displayTemp() {
    while(true) {
      ts.averageTemp();
      ...
    }
  }
}
```

```
class Container implements IContainer {
  TempSensor_v2 ts;
  ...
}
```

Figure 6.8: Binary-incompatible updates.

Currently, we allow caller related methods such as method `displayTemp` to refer to removed methods, fields, or super types, which is no big deal as long as those accesses are read only and thus do not result in program state changes. However, read only accesses may be the exception and methods such as removed method `averageTemp` may alter the program state, which possibly results in wrong program behavior (e.g., method `averageTemp` could overwrite the temperature computed by up-to-date method `currentTemp` with average temperatures). To avoid inconsistencies because of binary-incompatible updates, we must somehow invalidate accesses to removed methods, fields, and super types.



```
 1 class TempDisplay {
 2   TempSensor ts;
 3   IContainer cont;
 4   ...
 5   void displayTemp() {
 6     while(true) {
 7       getTS().averageTemp();
 8       ...
 9     }
10   }
11
12   //getter and setter
13   TempSensor getTS() {
14     return ts;
15   }
16   ...
17 }
```

```
18 class TempSensor {
19   String tempUnit;
20   Object newTS;
21   ...
22   int averageTemp() {
23     /* average Temp */
24   }
25
26   // getter and setter
27   ...
28 }
```

```
29 class TempDisplay {
30   TempSensor ts;
31   IContainer cont;
32   ...
33   void displayTemp() {
34     while(true) {
35       getTS().averageTemp();
36       ...
37     }
38   }
39
40   //getter and setter
41   TempSensor getTS() {
42     return ts;
43   }
44   ...
45 }
```

```
46 class Container implements IContainer {
47   TempSensor_v2 ts;
48   ...
49   //getter and setter
50   ...
51 }
```

```
52 class TempSensor {
53   String tempUnit;
54   Object newTS;
55   ...
56   int averageTemp() {
57     try {
58       throw new NoSuchMethodException("Method Removed");
59     } catch(NoSuchMethodException e) {
60       ...
61     }
62   }
63
64   //getter and setter
65   ...
66 }
```

Figure 6.9: Support for binary-incompatible updates.

In Figure 6.9, we show how we intend to invalidate accesses to the removed elements. Just like for state-loss prevention purposes, we redefine the methods within the old class versions. But, we do not add state mapping code and forward the calls to the up-to-date class version. We simply remove the original method bodies and corresponding to whether the removed element is a field or a method, throw `NoSuchMethodException` (see Figure 6.9, Line 58) or `NoSuchFieldException`, which does not cause unwanted program-state changes and thus has no influence on the program's consistency.

6.2.4 Reflection Support

We do not only focus on improved update speeds, thread-safe updates, state-loss prevention, and the handling of binary-incompatible updates. Additionally, we are working on solutions to overcome several problems the different versions of a class present in the JVM may cause. The main issue to overcome is the limited support of our current JAVADAPTOR implementation for reflective calls of reloaded (updated) classes. Under certain conditions, those calls may address old versions of a reloaded class and not the latest class version, which may result in wrong program behavior. This would be for instance the case when the class object of the class to be reloaded was cached before the update. Each reflective call based on this cached class object would access the old class version.

With our solutions for state-loss prevention and binary-incompatible updates, which basically forward all requests (including the reflective ones) to the most recent version of a class/instance, we already cover many different kinds of reflective requests. What the approaches not yet fully cover are string-based reflective calls in combination with type checks (e.g., via `instanceof`). Those calls could be supported with two different strategies. First, we could modify the Reflection API in such way that it redirects even string-based reflective calls to the most recent class version. Second, we could parse the class files for occurrences of string-based reflective calls and change the strings representing a class name to the up-to-date class name. However, further investigations are necessary to find an optimal solution for the described problem.

6.3 Bridging the Gap between Practicability and Consistency

So far, we discussed solutions for issues already solved by other DSU approaches such as Kim's proxy-based DSU approach [Kim09] or Javeleon [Gre10b]. What remains an open question to the whole research community is how to reliably (immediately) apply updates and **fully** ensure program consistency beyond the updates. Gupta et al. state in [GJB96] that the consistency problem is undecidable. Nevertheless, a lot of related work exists facing the problem (see [VEBD07, KM90, SHB^{+}05, HN05, MBJ06, BAM^{+}09, KB02, Mak09, Wür11]). But, to our best knowledge, some approaches provide approximated solutions only, whereas others are not applicable in real-world scenarios (e.g., due to the lack of tool support, etc.) or may reject the scheduled update. That is, our big goal with JAVADAPTOR is to provide an update mechanism which fully ensures program consistency, is useful in practice, and reliably applies updates.

6.4 Application to other Languages

Within this work, we discussed the benefits of unrestricted dynamic software update capabilities and presented JAVADAPTOR which brings those capabilities to Java. However, unrestricted DSU are not only beneficial for Java. They are beneficial in general. Therefore, we plan to apply our solution to other popular languages and platforms. One candidate is Microsoft's .NET platform, which could execute programs written in languages such as C#, C++, J#, or Visual Basic. The .NET platform is ideal, because the underlying runtime (i.e., the Common Language Runtime) supports dynamic method body redefinitions as we use them in our DSU approach to update references to reloaded classes [EF05]. Another possible candidate is the C++ programming language. It does not offer the required method body redefinitions. However, Hjálmtÿsson and Gray described in [HG98] how proxies could be used to enable method body redefinitions in C++ and thus prepare the language for our update approach.

6.5 Discussion

When looking at the enhancements that we are going to integrate into JAVADAPTOR, one may wonder if those enhancements would compromise one of the contributions of JAVADAPTOR claimed in this paper, e.g., its performance. Particularly, the system-wide usage of getter and setter methods (note that the getters and setters have to be created for all class and instance fields of all classes including the system classes of Java) would probably cause significant performance penalties. But, contrary to expectations, first benchmark results show that this is virtually not the case, which is because of the excellent optimization capabilities of the JVM and its just-in-time compiler (we found that the JVM is able to optimize getter- and setter-based field accesses to such an extent, that they are as fast as direct field accesses). In addition, other DSU approaches such as Kim's proxy-based DSU approach [Kim09] and Javeleon [Gre10b], which base on lazy state mappings and use system-wide getter and setter methods for similar purposes as we will do, show that those kinds of enhancements must not cause significant performance drops. For instance Gregersen estimates in [GJ11] the performance overhead of Javeleon at moderate 15 % and we see no reason why future JAVADAPTOR versions should introduce bigger performance penalties (we rather expect the performance penalties to be significantly below those 15 %).

All in all, we are optimistic to provide a stable version of JAVADAPTOR with fast and thread-safe updates, improved state-loss prevention, optimized handling of binary-incompatible changes, and better support for reflective calls, soon. As already mentioned, preliminary results of experiments with JAVADAPTOR prototypes suggest that the planned enhancements must not heavily compromise the performance of the updated program. Another fact that makes us confident to fit JAVADAPTOR with high quality solutions for the mentioned issues is that we can (to some extent) build on solutions of related DSU approaches (such as presented in [Kim09] and [Gre10b]) facing similar problems.

6.6 Summary

In this chapter, we discussed work in progress and future work to improve JAVADAPTOR. We detailed how we are going to improve the update speed of JAVADAPTOR and presented first benchmark results showing significant improvements in this regard. We additionally described our plans to push JAVADAPTOR further toward consistent program updates. That is, we detailed how we could ensure thread-safety through special synchronization objects and exploit getters and setters to prevent state losses, deal with binary-incompatible updates, and provide full reflection support. We, last but not least, discussed long-term objectives such as full support for consistent updates and the application of our DSU approach to languages different from Java.

7 Related Work

Within this chapter, we give an overview of other work done in the domain of dynamic software updates. We consider such an overview as important, because it helps us to highlight the benefits of our solution compared to already existing work. At first, we provide an overview of recent work to overcome Java's limitations regarding dynamic software updates. Next, we summarize dynamic software update approaches for languages different from Java, e.g. C, C++, or SmallTalk.

Furthermore, we evaluate the quality of the related approaches regarding the criteria central to our work, i.e., update flexibility, performance, platform independence, architecture independence, and update granularity. We could not test every approach against the criteria by ourself, because some approaches are not freely available, do only support outdated runtime environments, or require specific program architectures (which renders the usage of one and the same benchmark impossible). However, we searched the literature and found detailed information on the capabilities of the related work regarding our criteria.

7.1 Dynamic Software Updates for Java

Researchers spent a lot of time to overcome Java's shortcomings regarding dynamic software updates. For better comparability and because of the broad range of related work ranging from theoretical to practical solutions, we predominantly focus on practice-oriented approaches which, like JAVADAPTOR and in contrast to theoretical solutions, can be directly applied in real-world scenarios. We group the related work into two groups which reflect the level of the approach's application, i.e., the group of approaches established at *Language Level* and the approaches aiming at *JVM Patches*.

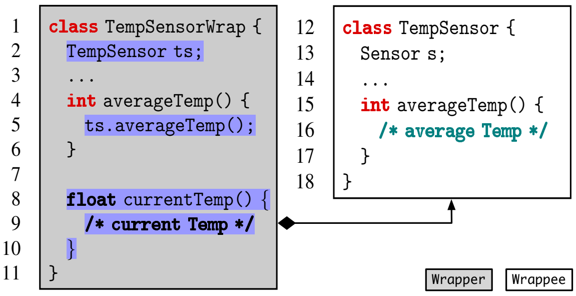
7.1.1 Language Level

The Java language, respectively the object-oriented paradigm, naturally offers different entry points to approximate dynamic software updates, e.g., the decorator pattern or the proxy pattern [GHJV04]. In the following, we will discuss DSU approaches that perform runtime updates at language level and exploit those entry points.

Object Wrappings

This section shares material with the APSEC'08 paper "Towards Unanticipated Runtime Adaptation of Java Applications" [PKS08] and the RAM-SE'08 paper "Object Roles and Runtime Adaptation in Java" [Puk08].

In previous work, we presented a DSU approach, which combines object wrappings with interfaces and, similar to JAVADAPTOR, uses Java HotSwap to update referring program parts [PKS08, Puk08].

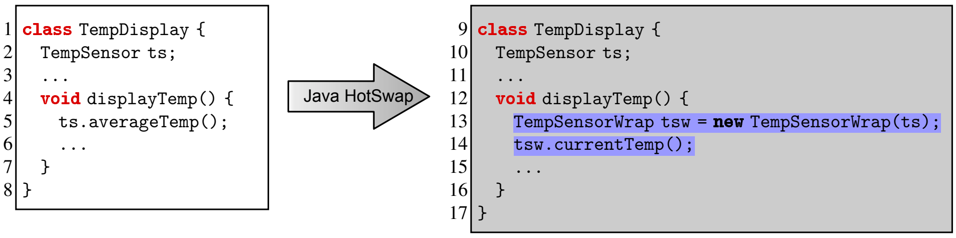


```
 1  class TempSensorWrap {
 2    TempSensor ts;
 3    ...
 4    int averageTemp() {
 5      ts.averageTemp();
 6    }
 7
 8    float currentTemp() {
 9      /* current Temp */
10    }
11  }
```

```
12  class TempSensor {
13    Sensor s;
14    ...
15    int averageTemp() {
16      /* average Temp */
17    }
18  }
```

Figure 7.1: Object wrapping.

Figure 7.1 shows how this approach deals with class schema changing program updates. To apply new methods, such as method `currentTemp` of our small weather station example (see Section 3.1), we use object wrappings. That is, we create a wrapper (see Figure 7.1, class `TempSensorWrap`) which applies the new functionality (in the example method `currrentTemp`, Lines 8 – 10) to the program, whereas other methods are delegated to the original class (here class `TempSensor`) to preserve the original behavior.



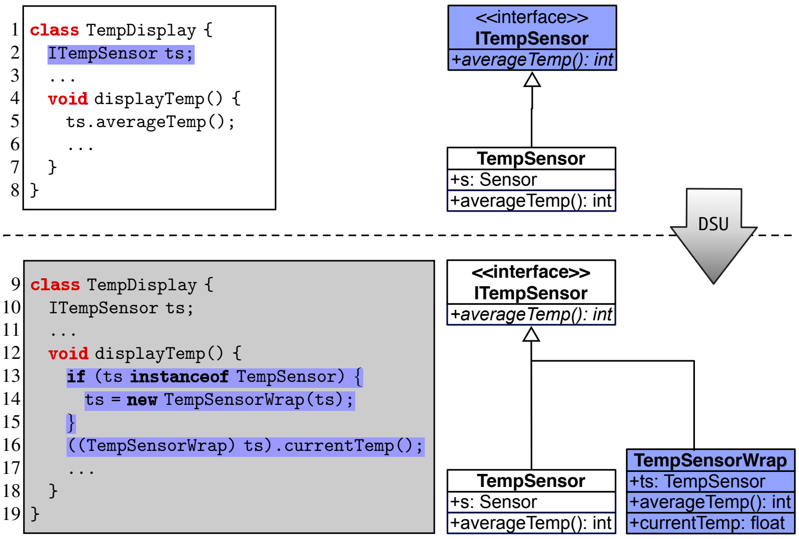
```
1  class TempDisplay {
2    TempSensor ts;
3    ...
4    void displayTemp() {
5      ts.averageTemp();
6      ...
7    }
8  }
```

```
 9  class TempDisplay {
10    TempSensor ts;
11    ...
12    void displayTemp() {
13      TempSensorWrap tsw = new TempSensorWrap(ts);
14      tsw.currentTemp();
15      ...
16    }
17  }
```

Figure 7.2: Caller update.

To use the newly added functionality, the referring program parts must be updated such as shown in Figure 7.2. That is, the methods (such as method `displayTemp`) of all referring classes (here class `TempDisplay`) must be redefined using Java HotSwap. The reimplementation consists of two parts. First, an instance of the wrapper that introduces the new functionality (in our case class `TempSensorWrap`) is created (Line 13). Second, the newly added method is called (Line 14).

We note that the mechanism described above only provides temporary wrappings. In order to make the wrapping persistent, we combined our approach with interfaces (see Figure 7.3). The static type of all class and instance fields (such as in Figure 7.3 instance field `ts` of class `TempDisplay`) is changed (before program start) to an interface type (in our example of type `ITempSensor`), which enables the fields (such as field `ts`, Line 2) for late binding. Thus, it is possible to assign objects different from the original type (in the example of class `TempSensor`) to those fields during runtime. The application of program changes itself also bases on method redefinitions via Java HotSwap and works as depicted down to the left of Figure 7.3. First, an instance of the wrapper (here of class `TempSensorWrap`) is created (Line 14). It takes the original value assigned to the field (here of field `ts`) to be updated as input (in our case an instance of `TempSensor`). Second, the wrapper instance is assigned to the field (e.g., in the example depicted in Figure 7.3 the runtime type of field `ts` switches from `TempSensor` to `TempSensorWrap` (Line 14). To call methods not declared in the interface (such as method `currentTemp` of class `TempSensorWrap`), the object referenced by the updated field (in the example field `ts`) must be casted to the current runtime type (Line 16).



```
1 class TempDisplay {
2   ITempSensor ts;
3   ...
4   void displayTemp() {
5     ts.averageTemp();
6     ...
7   }
8 }
```

```
 9 class TempDisplay {
10   ITempSensor ts;
11   ...
12   void displayTemp() {
13     if (ts instanceof TempSensor) {
14       ts = new TempSensorWrap(ts);
15     }
16     ((TempSensorWrap) ts).currentTemp();
17     ...
18   }
19 }
```

Figure 7.3: Wrapping of long-living objects.

Even if we demonstrated the practicability of object wrappings in [PKS08], we observed different problems. One problem the approach comes with is the self-problem, which describes situations that require to call the wrapper from within the wrapped object [Lie86]. Since Java does not support such kind of delegations, object wrappings fail in such situations, which influences the approach's flexibility. Next, the interface approach used to enable late bindings renders removals of interface methods impossible, which further reduces the flexibility of the approach. In addition, wrappers cause significant performance penalties [GHJV04]. In [GP09], we measured the performance penalties caused by long wrapping chains, which raise by up to 50 percent compared to the same program without wrappers. Nevertheless, our solution was (to our best knowledge) the first approach based on object wrappings, which offered unanticipated dynamic software updates for Java programs. Other wrapper-based approaches, such as presented by Truyen et al. [TVJ$^+$01], Kniesel [Kni99], Pawlak et al. [PDFS01], Hunt and Sitaraman [HS04], Büchi and Weck [BW00], or Bettini et al. [BCG07], require to predefine the object wrappings (i.e., updates) before program start and thus offer only anticipated dynamic software updates.

DSU for Parallel High Performance Applications

Kim et al. [KTR11, Kim09] present a DSU approach, which uses the proxy pattern [GHJV04]. That is, referring classes and referred classes do not communicate directly with each other, but access special proxy classes and let those proxies manage the communication, which permits to dynamically update the classes behind the proxies. To add or remove methods or fields, i.e., to change class schemas, they load helper classes which contain the newly added elements. Those new elements are accessed through generic invoke methods, which are part of every program class. To keep the invoke methods up-to-date, i.e., to let the invoke methods refer the newly added elements of the helper classes, they also use Java HotSwap. State migration between objects of old and new helper classes is carried out by special mapping methods.

The strength of the approach is its performance, which Kim et al. achieve through utilization of bytecode instructions `invokespecial` and `invokeinterface` (note that those bytecode instructions are optimized by the JIT compiler, which results in proxy access times comparable to the access times of direct calls). On the contrary, the proxy approach prevents inheritance hierarchy updates and hinders removals of methods that are part of a proxy, which contradicts our flexibility criterion.

DUSC

Orso et al. [BCG07] present *DUSC* (Dynamic Updating through Swapping of Classes), which, similar to the approach of Kim et al. [KTR11, Kim09], bases on proxy classes. That is, classes and their instances communicate with each other through proxies, which manage the communication. The basic elements of DUSC are the implementation classes (which correspond to the original classes), interface classes (responsible for accessing different versions of a class), wrapper classes (manage the inter-class communication and trigger the update), and the state classes (do the state migration job). DUSC allows the developer to add, remove, or change already loaded classes. To change a class, the tool simply swaps the corresponding implementation class.

Like Kim's proxy-based DSU approach, DUSC causes only minimal performance drops. However, it is less flexible than other DSU approaches. DUSC could only update a class if old and new class version share the same public interface, which renders inheritance changes and additions/removals of public or protected methods impossible. The only way to change the public interface of a class with DUSC is to remove the old class version and introduce the up-to-date class version as an entirely new class, which results in state losses (i.e, all instances of the old class version would be deleted).

JRebel

The next proxy-based DSU approach is *JRebel* [Kab11]. It is a Java agent which is loaded into the JVM at program start. Once loaded, it scans the program's class path for changed classes and, if found, loads the up-to-date classes with a new version into the JVM. JRebel allows the developer to add or remove fields and methods and could be integrated with many different program architectures.

Even if beneficial in many update scenarios, JRebel fails when it comes to updates that require inheritance hierarchy changes, which is due to the usage of proxies. Moreover, JRebel introduces performance penalties of up to 60 percent.

FastSwap

Another proxy-based dynamic software update approach is *FastSwap* [Ora11a], which is integral part of Oracle's WebLogic Server[1] (since WebLogic Server version 10.3). FastSwap supports additions and removals of fields and methods and preserves the state of existing objects.

Similar to our approach, FastSwap keeps the original class loader and reloads changed classes with new versions, which enables the approach for fine-grained updates. However, FastSwap does not support inheritance hierarchy changes and thus offers not the desired level of flexibility. In addition, the WebLogic Server which provides FastSwap only executes component-based applications.

OSGi

The *Open Services Gateway initiative* (OSGi) offers a framework which uses customized class loaders to dynamically update component-based Java applications [All11]. The framework consists of five layers which are of different importance for dynamic software updates.

[1] `http://www.oracle.com/technetwork/middleware/weblogic/overview/index.html`

The layers crucial for DSU are the module layer (defines the modularization model), the life cycle layer (defines how the components are deployed, updated, or undeployed), and the service layer (constitutes the developer's entry point for the development and deployment of components).

OSGi serves all kinds of class schema changes, i.e., it allows the user to update virtually all parts of a running program in an unanticipated way. However, one issue of the framework is that component updates cause state losses, which is a big constraint regarding our flexibility criterion. Next, OSGi causes architecture dependencies, because it requires the applications that run on top of it to be refactored into components. Furthermore, even small changes require to replace whole components which may be inefficient.

Javeleon

As the OSGi framework and FastSwap show, components are a good basis for the development of dynamic software update approaches, which is something the developers of *Javeleon* [Gre10b] realized, too. Javeleon builds up on the NetBeans framework[2] and extends it with flexible dynamic software update capabilities. The core of Javeleon constitutes the In-Place Proxification mechanism, which basically delegates calls of outdated methods to their up-to-date counterparts.

Javeleon offers the whole bandwidth of possible class schema changes including inheritance hierarchy updates. It even preserves the program state. Furthermore, Gregersen et al. claim in [GJ11] that Javeleon comes with a moderate performance overhead of only 15 %. But, Javeleon needs components to act and thus introduces architecture dependencies. Furthermore, even minimal program changes require to replace whole components. That is, Javeleon offers only coarse-grained updates.

In October 2011, the developers of Javeleon came out with a separate tool version, which offers fine-grained dynamic software updates for standalone applications. However, we could not find any information on whether both tool versions base on the same concepts and thus could be integrated into one single tool tackling the architecture-dependency issue. So far, developers have to use the one or the other tool version depending on whether they want to update component-based applications or standalone applications. That is, the architecture dependency problem remains. In addition, we used the standalone version of Javeleon to replay our HyperSQL case study presented in Section 5.1.1 and measured runtime performance penalties of approximately 80 percent.

[2] http://netbeans.org/features/platform/index.html

Dynamic Update Transactions

Zhang and Huang propose *Dynamic Update Transactions* (*DUT*) [ZH07], which make use of customized class loaders. They use special update classes to process updates and load every class scheduled for an update with a new customized class loader instance. Method and field invocations are processed through reflective calls. The state transfer between outdated and up-to-date objects is performed by state mapping methods which must be applied to every application class before program start.

The major issue the approach suffers from is performance degeneration, which is caused by the fact that every method and field must be accessed using Java's reflection API. Cazzola [Caz04] found out that even simple reflective method invocations slow down method invocations with a factor of up to 6.5 compared to direct method invocations. More complex reflective calls might cause even higher performance penalties.

Iguana/J

Iguana/J[3] is another framework which offers dynamic software updates for Java applications [RC02]. It was developed by Barry Redmond and bases on the language independent Iguana reflective programming model [Gow97]. The basic principle of Iguana/J is to use so called meta objects to enhance standard Java objects with new functionality. Central elements of the approach are special meta classes and meta protocols. The meta classes contain the update and the meta protocols are responsible for grouping and managing the meta classes.

The framework is highly flexible, but it significantly slows down program execution, which is due to the reflective programming model Iguana/J bases on. The authors state that the reflective programming model slows down object creations and method invocations by a factor of about 25.

UpgradeJ

UpgradeJ is an extension to the Java language, which allows the developer to plan and define dynamic and type-safe class upgrades [BPN08]. The language extension supports three different kinds of upgrades: a) new class upgrades, b) revision upgrades, and c) evolution upgrades. The new class upgrades add new classes to the running program. Like Java HotSwap, revision upgrades offer class schema keeping updates, i.e., they allow the developer to redefine the method bodies of already loaded classes. The most powerful upgrade form is the evolution upgrade. It enables program updates, which add fields or methods to the running program.

[3] `http://www.compeng.dit.ie/staff/bredmond/iguanaj/`

As previously mentioned, adding fields or methods using UpgradeJ is no problem. But, the update approach prevents removals of fields and methods, which affects the approach's flexibility. Even Tempero et al. [TBNP08] found out that UpgradeJ could not be applied to all update scenarios they considered.

7.1.2 JVM Patches

As mentioned in Section 2.4, the JVM disallows the developer to reload a class whose schema has changed and thus forbids flexible dynamic software updates. Therefore, researchers suggest virtual machine patches that enable to reload classes with changed schemas. In this section, we will discuss different approaches which build up on JVM modifications.

DCEVM

Würthinger's *Dynamic Code Evolution VM* (DCEVM) [Wür11, WBA$^+$10] builds up on the work of Dmitriev [Dmi01a] and provides the OpenJDK HotSpot VM with extended dynamic update capabilities. The patch allows the developer to add or remove methods and fields. It even permits inheritance hierarchy updates. Central part of the DCEVM is a modified garbage collector, which exchanges outdated class versions with their up-to-date counterparts, maps the state, and redirects all existing references to the new class versions and their instances.

The DECVM does not only fulfill our flexibility criterion. It additionally, offers fine-grained DSU and causes virtually no performance penalties (tested against our HyperSQL case study). Furthermore, the DECVM is not restricted to specific program architectures. However, as all JVM patches, it relies on a specific JVM implementation (i.e. the HotSpot JVM) and thus causes platform dependencies.

JVolve

Subramanian et al. present *JVolve* [SHM09], an alternative JVM patch also aiming at dynamic software updates. JVolve extends the Jikes RVM[4] (Jikes Research Virtual Machine), which is a testbed for alternative JVM implementations. Similar to the DCEVM, JVolve consists of a modified garbage collector that enables to load new versions of an already loaded class. During class reloading, JVolve generates special transformer classes, which transfer the state from old to new class versions. The patch allows the developer to add and remove fields and methods.

[4] http://jikesrvm.org/

JVolve was developed with performance in mind. That is, updates do not slow down the updated application. The problem with JVolve is, that it does not support inheritance hierarchy updates. Moreover, it relies on the Jikes RVM and thus causes platform dependencies.

JDrums

The *Java Distributed Run-Time Update Management System* (JDrums) [RA00] is another DSU approach, which belongs to the class of JVM patches. It decouples classes and objects from each other through indirection layers. Because of the indirections, outdated class versions could be easily replaced by their up-to-date counterparts. The state migration between old and new class (including their instances) is accomplished by so called conversion classes.

One major drawback of JDrums is its poor performance, which is caused by the fact that JDrums disables the JIT compiler and lets programs run in interpreted mode. In addition, JDrums does not map inherited state (i.e., the state that belongs to superclasses) and thus comes short regarding our flexibility criterion. As with every other so far considered JVM patch, JDrums, on the one hand, offers fine-grained dynamic updates and, on the other hand, causes platform dependencies.

DVM

The next JVM patch was proposed by Malabarba et al. [MPG$^+$00]. It is called *Dynamic Virtual Machine* (DVM) and modifies Oracle's HotSpot VM (version 1.2) for Solaris. The patch basically consists of two parts. The first part is the dynamic class loader, which (different from the standard class loaders) permits to replace already loaded classes. The second part of the patch builds the program logic that maps the state between old and new class versions (including their instances) and resolves/modifies the dependencies between updated classes and their callers.

DVM serves all possible kinds of program updates. That is, the patch even permits to change inheritance hierarchies. Furthermore, DVM enables fine-grained runtime updates and does not require specific program architectures to act. Unfortunately, the patch prevents method inlinings and disables the JIT (i.e., programs run in interpreted mode only), which heavily affects the program execution speed. On top of that, DVM relies on a specific version of Oracle's HotSpot VM and thus causes platform dependencies.

7.2 DSU Across Different Languages

Research regarding dynamic software updates does not only target Java and its runtime environment. In fact, DSU capabilities are in great demand across many different languages.

Within this section, we briefly discuss how dynamic software updates are supported in languages different from Java.

As we already mentioned in Section 2.2, dynamic type systems provide best conditions for runtime program updates. In fact, dynamically typed languages such as SmallTalk [GR83], CLOS [Ste90, GWB91], or Python [Bea09] natively support all possible kinds of dynamic software updates. Ruby [TFH05] only forbids inheritance hierarchy changes. All that update flexibility comes free of platform and architecture dependencies and at a fine level of update granularity. However, normally Java programs execute faster than the same programs based on dynamic languages. According to Fulgham and Gouy [FG11], Java is 10 times faster than SmallTalk and approximately 41 times faster than Python or Ruby. Thus, Java may be the better choice in performance critical application scenarios.

Whereas, dynamic languages come with build in support for dynamic software updates, statically typed languages C and C++ do not. Neamtiu et al. developed *Ginseng* [NHSO06] to provide C with dynamic update capabilities. Ginseng builds up on function indirections and type wrappers. It allows the developer to add and remove fields and methods, but introduces performance penalties of up to 32 % compared to the same non-updatable program.

Another DSU approach for the C programming language is proposed by Makris et al. [Mak09, MB09]. It is called *UpStare* and, similar to Ginseng, uses pointer indirections to enable runtime program updates. UpStare supports additions and removals of fields and methods. Makris et al. evaluated how UpStare affects the system's execution speed and measured performance penalties of at least 30 percent.

Hjálmtÿsson and Gray propose *Dynamic C++ Classes* [HG98] to overcome the limitations regarding dynamic software updates in C++. The approach uses proxies and enables additions and removals of private fields and methods. Additions or removals of public members are not possible, because the proxy interfaces are immutable. Moreover, Hjálmtÿsson's and Gray's solution causes performance penalties of up to 4872 % regarding object creation. Method invocations come with an overhead of up to 611 percent.

Dynamically Evolvable C++ Classes [SKNCN06] is another DSU approach for the C++ language. Central to this approach are smart pointers (which could be redirected to new class versions) and state transformation functions. Dynamically Evolvable C++ Classes permit additions and removals of fields and methods. Inheritance hierarchy updates are not supported. Object creation times in the presence of this approach are increased by up to 80 percent. Furthermore, method invocations using smart pointers cause significantly longer access times than direct method invocations.

Edit-and-Continue[5] is a feature of Microsoft's .NET runtime environment, i.e., the Common Language Runtime (CLR). Edit-and-Continue offers the same DSU capabilities as Java HotSwap, i.e., it allows developers to redefine method bodies. Its usage was origi-

[5] http://msdn.microsoft.com/en-us/library/bcew296c%28v=vs.80%29.aspx

	DSU Approach	Flexibility	Performance	Plat. Indep.	Arch. Indep.	Fine Gran.
	JavAdaptor	●	●	●	●	●
Language Level	***Object Wrapping* [PKS08]**	○	○	●	●	●
	***Kim et al.* [Kim09]**	○	●	●	●	●
	***DUT* [ZH07]**	●	○	●	●	●
	***Iguana/J* [RC02]**	●	○	●	●	●
	***UpgradeJ* [BPN08]**	○	–	●	●	●
	***DUSC* [ORH02]**	○	●	●	●	●
	***JRebel* [Kab11]**	○	○	●	●	●
	***OSGi* [All11]**	○	●	●	○	○
	***FastSwap* [Ora11a]**	○	●	●	○	●
	***Javeleon (NetBeans)* [Gre10b]**	●	●	●	○	○
	Javeleon (Standalone)	●	○	●	○	●
JVM Patches	***DECVM* [Wür11]**	●	●	○	●	●
	***JVolve* [SHM09]**	○	●	○	●	●
	***JDrums* [RA00]**	○	○	○	●	●
	***DVM* [MPG$^+$00]**	●	○	○	●	●
Other Languages	***SmallTalk* [GR83]**	●	○	●	●	●
	***CLOS* [Ste90, GWB91]**	●	–	●	●	●
	***Python* [Bea09]**	●	○	●	●	●
	***Ruby* [TFH05]**	○	○	●	●	●
	***Ginseng* [NHSO06]**	●	○	●	●	●
	***UpStare* [Mak09, MB09]**	●	○	●	●	●
	***Dyn. C++ Classes* [HG98]**	○	○	●	●	●
	***Dynamically Evolvable C++ Classes* [SKNCN06]**	○	○	●	●	●
	***Edit-and-Continue* [EF05]**	○	●	○	●	●

Table 7.1: Comparison: JavAdaptor vs. related work (based on our evaluation criteria). Meaning of the symbols: ● = criterion fulfilled, ○ = criterion not fulfilled, – = no information available.

nally restricted to the .NET version of C++. In 2005 Eaddy and Feiner proposed a general Edit-and-Continue approach [EF05], which enables method body redefinitions across all programming languages provided by the .NET platform. However, even if meanwhile language independent, Edit-and-Continue only works properly on Windows platforms.

7.3 Summary

Within this chapter, we gave an overview of other work done in our field of research and compared the related work with JAVADAPTOR on the basis of our evaluation criteria, i.e., update flexibility, performance, platform independence, architecture independence, and update granularity. We mainly focussed on DSU approaches for Java, because JAVADAPTOR targets Java, too. However, we even considered DSU for languages different from Java, i.e., C, C++, SmallTalk, Python, Ruby, and CLOS. An overview of the related work including their strengths and weaknesses regarding our evaluation criteria can be found in Table 7.1.

What we have learned from our studies, different from JAVADAPTOR, none of the related DSU approaches fulfills all evaluation criterions, which underpins the novelty, originality, and benefits of our solution.

8 Summary and Concluding Remarks

With this chapter, we aim to recall and sum up the central parts and contributions of the presented work and draw conclusions.

8.1 Summary

Chapter 2. In Chapter 2, we provided background information on the field of research, i.e., dynamic software updates. We introduced basic terms and definitions and detailed why dynamic software updates on the basis of statically typed languages are challenging. In addition, we described the internals of Java's runtime environment (i.e., the Java Virtual Machine) and detailed how it supports and restricts dynamic software updates.

Chapter 3. Within Chapter 3, we presented the conceptual core of JAVADAPTOR. We described how JAVADAPTOR integrates with the runtime environment of Java, i.e., with the Java Platform Debugger Architecture of the JVM. In addition, we detailed how our tool combines class reloadings, Java HotSwap, containers, and proxies to hit the targeted high level of update flexibility. We further gave explanations on JAVADAPTOR's state mapping mechanisms. Having described the core concepts of JAVADAPTOR, we exemplified on the basis of inheritance hierarchy changes and nested class updates why the concepts cover even complex update scenarios.

Chapter 4. Chapter 4 details the implementation of JAVADAPTOR. The main purposes of revealing the implementational details of JAVADAPTOR were to help developers to reproduce our solution and to impart the knowledge required to apply our approach to languages different from Java. We first described JAVADAPTOR from the users point of view. Then, we went through every part of JAVADAPTOR's workflow and detailed its implementation including possible pitfalls.

Chapter 5. Our vision was the development of a highly flexible, performant, platform independent, architecture independent, and fine-grained approach for dynamic software updates. In Chapter 5, we evaluated whether JAVADAPTOR meets the stated goals. We demonstrated the flexibility and fine update granularity of our tool on the basis of three different case studies, i.e., HyperSQL, Snake, and the Refactorings case study. Furthermore, we measured the runtime performance. In a nutshell, solely our proxy concept to avoid schema changes of referring classes introduces slight performance penalties. The runtime performance remains the same even after many updates. We additionally evaluated the update speed of our current JAVADAPTOR version and found that it could be improved, but is sufficient in many different scenarios. Moreover, we successfully applied JAVADAPTOR to all publicly available standard JVM's and thus were able to demonstrate the tools platform independence. Next, we gave reasons for why JAVADAPTOR is architecture independent. Even if not central part of our contribution, we last but not least measured the memory consumption of JAVADAPTOR after many updates and found the footprint acceptable.

Chapter 6. In Chapter 6, we summarized ongoing and future work to improve JAVADAPTOR. The chapter consists of three different parts. Within the first part of Chapter 6, we detailed how we are going to improve the update speed of JAVADAPTOR and presented first benchmark results showing significant improvements in this regard. The second part of the chapter consists of descriptions on tool enhancements, which aim at consistent program updates. We detailed how we are going to support thread-safe updates, prevent potential state losses, handle binary-incompatible updates, and provide full reflection support. Within part three of Chapter 6, we discussed long-term objectives such as our plans to apply JAVADAPTOR to languages different from Java.

Chapter 7. Within Chapter 7, we discussed the related work and reviewed it with respect to our evaluation criteria. Unlike JAVADAPTOR, none of the discussed approaches fulfills all evaluation criterions, which underpins the novelty, originality, and benefits of our solution.

8.2 Contributions

The contributions of the presented work are manifold. Primarily, we aimed at the development of concepts that fulfill the five criterions justified in Chapter 1, i.e., at concepts for highly flexible, performant, platform independent, architecture independent, and fine-grained dynamic software updates for Java. Besides, we contributed a tool which implements our concepts. Using the tool, we were able to demonstrated the practicability of our solution, and could substantiate the fulfillment of the thesis goals. In the following, we list our contributions in detail.

Goals at Design Level. We developed novel concepts that provide Java with highly flexible, performant, platform independent, architecture independent, and fine-grained dynamic update capabilities:

- We hit the targeted high level of *update flexibility* through combining schema changing class reloadings with caller-side updates on the basis of Java HotSwap, containers, and proxies.
- JAVADAPTOR fulfills the *performance* criterion, because we designed the containers and proxies in such a way that they introduce only few indirections and thus execute fast.
- We achieved *platform independence*, because we established most of our concepts at language level. JAVADAPTOR only requires Java HotSwap to be provided by the targeted JVM, which is fulfilled by all major standard JVMs.
- We fulfilled the *architecture independence* criterion, because neither our container nor our proxy approach requires specific program architectures to act.
- Our approach performs *fine-grained* program updates. It only changes the program parts (i.e., the classes) scheduled for an update. In case of schema changing updates of superclasses, we must replace all subclasses in addition.

Tool Support. We fully implemented the presented concepts of JAVADAPTOR:

- We show that the described concepts are *implementable*.
- We provide our implementation as an *easy to use* eclipse plugin. Doing so, developers can take advantage of unrestricted dynamic software updates for Java applications.
- We *detailed our implementation*. The description may help developers to reproduce our solution and imparts the knowledge required to apply the solution to languages different from Java.

Case Studies. We applied JAVADAPTOR to three different non-trivial case studies:

- With HyperSQL and Snake, we demonstrated the *practicability* of JAVADAPTOR by two complementary real-world examples. The HyperSQL case study confirms the usefulness of JAVADAPTOR with respect to maintenance tasks, while the Snake case study reveals the tool's benefits during development.
- We showed the *general applicability* of JAVADAPTOR with the help of our Refactoring case study.

Empirically Goals. We substantiated that JAVADAPTOR fulfills the thesis goals:

- We demonstrated the *flexibility* of JAVADAPTOR on the basis of our HyperSQL, Snake, and Refactoring case study. Because the case studies cover many different kinds of dynamic software updates, chances are high that JAVADAPTOR performs well when it must update applications different from the ones subject to our studies.
- We confirmed that JAVADAPTOR is *performant*. We dynamically updated HyperSQL and could not measure any performance overhead. We further created micro benchmarks and measured the performance of our containers and proxies. To sum up, containers do not decrease program performance, while proxies cause only slight performance penalties. Moreover, we measured the update speed of two different JAVADAPTOR versions, i.e., with and without lazy state mappings. We found that the version based on lazy state mappings updates programs quickly, while the update speed of the other version is not outstanding high but sufficient in many cases.
- We demonstrated JAVADAPTOR's *platform independence*. We successfully applied JAVADAPTOR to all major standard JVMs without any modification.
- We sketched why JAVADAPTOR is *architecture independent*. On the one hand, we successfully updated programs with different architectures. On the other hand, we gave reasons why our container and proxy approaches do not rely on specific program architectures.
- We confirmed that JAVADAPTOR enables *fine-grained* dynamic software updates on the basis of our case studies. In all cases, the tool solely updated the program parts that must be really updated, while it kept the remaining program parts untouched.

8.3 Conclusion and Outlook

Dynamic software updates are a often requested approach to update applications while improving the user experience and avoiding down times. Furthermore, DSU support software developers during development, because they do not need to restart their applications to test the changed program parts. Even if in great demand, particularly statically typed languages lack extensive dynamic update capabilities. Therefore, there is a large body of research on dynamic software updates for statically typed languages. But so far, existing approaches have shortcomings either in terms of update flexibility, performance, platform independence, architecture independence, and/or update granularity. With JAVADAPTOR, we have shown that dynamic software updates must not come with the mentioned restrictions. Conceptually, JAVADAPTOR combines schema changing class replacements with reference updates based on Java HotSwap, containers, and proxies. We detailed the concepts of JavAdaptor and their implementation. In addition, we demonstrated the practicability of JavAdaptor within different non-trivial case studies and substantiated that the tool fulfills the thesis goals.

With JavAdaptor, we improved the state of the art regarding dynamic software updates. Nevertheless, there is still space for enhancements and optimizations. As we detailed in Chapter 6, our short-term objectives are improvements regarding the update speed, improved deadlock and state-loss prevention, and full support for binary-incompatible updates and reflective calls. We further aim to apply our solution to languages different from Java. However, our big goal is to provide a JavAdaptor version, which fully ensures program consistency, is useful in practice, and reliably applies updates and thus once again improves the state of the art. All in all, we feel confident that the here presented work builds a strong base to meet the big goal.

Bibliography

[All11] The OSGi Alliance. OSGi Service Platform Core Specification, December 2011. `http://www.osgi.org/Download/File?url=/download/r4v42/r4.core.pdf`.

[BAM$^+$09] R. Bazzi, A., K. Makris, P. Nayeri, and J. Shen. Dynamic Software Updates: the State Mapping Problem. In *Proceedings of the International Workshop on Hot Topics in Software Upgrades*, pages 7:1–7:2. ACM, 2009.

[BCG07] L. Bettini, S. Capecchi, and E. Giachino. Featherweight Wrap Java. In *Proceedings of the Symposium on Applied computing*, pages 1094 – 1100. ACM, 2007.

[Bea09] D.M. Beazley. *Python Essential Reference*. Addison-Wesley, 4. edition, 2009.

[Bia06] R. P. Bialek. *Dynamic Updates of Existing Java Applications*. PhD thesis, University of Copenhagen, 2006.

[BMZ$^+$05] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a Taxonomy of Software Change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, 2005.

[BPN08] G. Bierman, M. Parkinson, and J. Noble. UpgradeJ: Incremental Typechecking for Class Upgrades. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 235 – 259. Springer, 2008.

[BW00] M. Büchi and W. Weck. Generic Wrappers. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 201 – 225. Springer, 2000.

[Caz04] W. Cazzola. SmartReflection: Efficient Introspection in Java. *Journal of Object Technology*, 3(11):117–132, 2004.

[Chi00] S. Chiba. Load-Time Structural Reflection in Java. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 313–336. Springer, 2000.

[CN03] S. Chiba and M. Nishizawa. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 364 – 376. Springer, 2003.

[DJ06] D. Dig and R. Johnson. How do APIs Evolve? A Story of Refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18:83–107, 2006.

[Dmi01a] M. Dmitriev. *Safe Class and Data Evolution in Large and Long-Lived Java Applications*. PhD thesis, University of Glasgow, 2001.

[Dmi01b] M. Dmitriev. Towards flexible and safe Technology for Runtime Evolution of Java Language Applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution*, pages 1 – 7, 2001.

[EF05] M. Eaddy and S. Feiner. Multi-Language Edit-and-Continue for the Masses. Technical Report CUCS-015-05, Columbia University, 2005. `http://www.cs.columbia.edu/techreports/cucs-015-05.pdf`.

[EVDB05] P. Ebraert, Y. Vandewoude, T. D'Hondt, and Y. Berbers. Pitfalls in Unanticipated Dynamic Software Evolution. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 41–49. University of Magdeburg, 2005.

[Fab76] R. S. Fabry. How to Design a System in which Modules can be hanged on the Fly. In *Proceedings of the International Conference on Software Engineering*, pages 470–476. IEEE, 1976.

[FG11] B. Fulgham and I. Gouy. The Computer Language Benchmarks Game, December 2011. `http://shootout.alioth.debian.org/`.

[Fow06] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2006.

[GBE07] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 57–76. ACM, 2007.

[GBI$^+$10] D. Garlan, F. Bachmann, J. Ivers, J. Stafford, L. Bass, P. Clements, and P. Merson. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2010.

[GHJV04] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. Addison-Wesley, 2004.

[GJ11] A. R. Gregersen and B. N. Jørgensen. Run-time Phenomena in Dynamic Software Updating: Causes and Effects. In *Proceedings of the Workshop on Principles of Software Evolution and ERCIM Workshop on Software Evolution*, pages 6–15. ACM, 2011.

[GJB96] D. Gupta, P. Jalote, and G. Barua. A Formal Framework for On-line Software Version Change. *IEEE Transactions on Software Engineering*, 22(2):120–131, 1996.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition)*. Addison-Wesley, 2005.

[GK06] R. Griffith and G. Kaiser. A Runtime Adaptation Framework for Native C and Bytecode Applications. In *Proceedings of the International Conference on Autonomic Computing*, pages 93–104. IEEE, 2006.

[Gow97] B. Gowing. *A Reflective Programming Model and Language for Dynamically Modifying Compiled Software*. PhD thesis, University of Dublin, 1997.

[GP09] S. Götz and M. Pukall. On Performance of Delegation in Java. In *Proceedings of the International Workshop on Hot Topics in Software Upgrades*, pages 1–6. ACM, 2009.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, 1983.

[Gre10a] A. Grebhahn. JavAdaptor - vererbungshierarchiebeeinflussende Programmänderungen zur Laufzeit. Bachelor thesis, University of Magdeburg, 2010.

[Gre10b] A. R. Gregersen. *Extending Netbeans with Dynamic Update of Active Modules*. PhD thesis, University of Southern Denmark, 2010.

[GSA04] J. Gustavsson, T. Staijen, and U. Assmann. Runtime Evolution as an Aspect. In *Proceedings of the Workshop on Foundations of Unanticipated Software Evolution*, pages 1–10. Elsevier, 2004.

[GWB91] R. P. Gabriel, J. L. White, and D. G. Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. *Communications of the ACM*, 34:29–38, 1991.

[HG98] G. Hjálmtẏsson and R. Gray. Dynamic C++ Classes – A Lightweight Mechanism to Update Code in a Running Program. In *Proceedings of the USENIX Annual Technical Conference*, pages 65–76. USENIX Association, 1998.

[HN05] M. Hicks and S. Nettles. Dynamic Software Updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049–1096, 2005.

[HS04] J. Hunt and M. Sitaraman. Enhancements: Enabling Flexible Feature and Implementation Selection. In *Proceedings of the International Conference on Software Reuse*, pages 86 – 100. Springer, 2004.

[HSHF11] C. M. Hayden, E K. Smith, M. Hicks, and J. S. Foster. State Transfer for Clear and Efficient Runtime Upgrades. In *Proceedings of the Workshop on Hot Topics in Software Upgrades*, pages 1–6. IEEE, 2011.

[Kab11] J. Kabanov. JRebel Tool Demo. *Electronic Notes in Theoretical Computer Science*, 264:51–57, 2011.

[KB02] F. Karablieh and R. A. Bazzi. Heterogeneous Checkpointing for Multithreaded Applications. In *Proceedings of the Symposium on Reliable Distributed Systems*, pages 140–149. IEEE, 2002.

[Kim09] D. K. Kim. *Applying Dynamic Software Updates to Computationally-Intensive Applications*. PhD thesis, Virginia Polytechnic Institute and State University, 2009.

[KM90] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293 –1306, 1990.

[Kni99] G. Kniesel. Type-Safe Delegation for Run-Time Component Adaptation. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 351–366. Springer, 1999.

[KTR11] D. K. Kim, E. Tilevich, and C. J. Ribbens. Dynamic software updates for parallel high performance applications. *Concurrency and Computation: Practice and Experience*, 23:415–434, 2011.

[LB98] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 36 – 44. ACM, 1998.

[Lia99] S. Liang. *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999.

[Lie86] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 214–223. ACM, 1986.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification – Second Edition*. Prentice Hall, 1999.

[Mak09] K. Makris. *Whole-Program Dynamic Software Updating*. PhD thesis, Arizona State University, 2009.

[MB09] K. Makris and R. A. Bazzi. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14. USENIX Association, 2009.

[MBJ06] Y. Murarka, U. Bellur, and R. K. Joshi. Safety Analysis for Dynamic Update of Object Oriented Programs. In *Proceedings of the Asia Pacific Software Engineering Conference*, pages 225–232. IEEE, 2006.

[MLH$^+$09] B. Morin, T. Ledoux, M.B. Hassine, F. Chauvel, O. Barais, and J.-M. Jezequel. Unifying Runtime Adaptation and Design Evolution. In *Proceedings of the International Conference on Computer and Information Technology*, pages 104–109. IEEE, 2009.

[MPG$^+$00] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime Support for Type-safe dynamic Java Classes. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 337 – 361. Springer, 2000.

[NHSO06] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical Dynamic Software Updating for C. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 72–83. ACM, 2006.

[OJ90] W. F. Opdyke and R. E. Johnson. Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. In *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications*, pages 145 – 161. ACM, 1990.

[Ora11a] Oracle. BEA WebLogic Server Using FastSwap to Minimize Redeployment, December 2011. http://download.oracle.com/docs/cd/E13222_01/wls/essex/TechPreview/pdf/FastSwap.pdf.

[Ora11b] Oracle. Java Platform Debugger Architecture, December 2011. http://download.oracle.com/javase/6/docs/technotes/guides/jpda/.

[Ora11c] Oracle. Java Virtual Machine Tool Interface Version 1.2, December 2011. http://download.oracle.com/javase/6/docs/platform/jvmti/jvmti.html.

[ORH02] A. Orso, A. Rao, and M. Harrold. A Technique for Dynamic Updating of Java Software. In *Proceedings of the International Conference on Software Maintenance*, pages 649–658. IEEE, 2002.

[Ori04] M. Oriol. *An Approach to the Dynamic Evolution of Software Systems*. PhD thesis, University of Geneva, 2004.

[PDFS01] R. Pawlak, L. Duchien, G. Florin, and L. Seinturier. Dynamic Wrappers: Handling the Composition Issue with JAC. In *Proceedings of the Conference on Technology of Object-Oriented Languages and Systems*, pages 56–65. IEEE, 2001.

[PG06] S. C. Previtali and T. R. Gross. Dynamic Updating of Software Systems Based on Aspects. In *Proceedings of the International Conference on Software Maintenance*, pages 83–92. IEEE, 2006.

[PGS$^+$11] M. Pukall, A. Grebhahn, R. Schröter, C. Kästner, W. Cazzola, and S. Götz. JavAdaptor: Unrestricted Dynamic Software Updates for Java. In *Proceedings of the International Conference on Software Engineering*, pages 989–991. ACM, 2011.

[PKC$^+$12] M. Pukall, C. Kästner, W. Cazzola, S. Götz, A. Grebhahn, R. Schröter, and G. Saake. JavAdaptor – Flexible Runtime Updates of Java Applications. *Software: Practice and Experience*, 2012. early view.

[PKS08] M. Pukall, C. Kästner, and G. Saake. Towards Unanticipated Runtime Adaptation of Java Applications. In *Proceedings of the Asia-Pacific Software Engineering Conference*, pages 85–92. IEEE, 2008.

[Puk08] M. Pukall. Object Roles and Runtime Adaptation in Java. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 33 – 37. University of Madeburg, 2008.

[RA00] T. Ritzau and J. Andersson. Dynamic Deployment of Java Applications. In *Proceedings of Java for Embedded Systems Workshop*, pages 1–9, 2000.

[RC02] B. Redmond and V. Cahill. Supporting Unanticipated Dynamic Adaptation of Application Behaviour. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 205–230. Springer, 2002.

[Sch10] R. Schröter. JavAdaptor - Programmänderungen zur Laufzeit in Bezug auf this Referenzen und Verschachtelte Klassen in Java. Bachelor thesis, University of Magdeburg, 2010.

[SHB⁺05] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and Flexible Dynamic Software Updating. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 183–194. ACM, 2005.

[SHM09] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic Software Updates: A VM-Centric Approach. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 1–12. ACM, 2009.

[SKNCN06] J. Stanek, S. Kothari, T. N. Nguyen, and C. Cruz-Neira. Online Software Maintenance for Mission-Critical Systems. In *Proceedings of the International Conference on Software Maintenance*, pages 93–103. IEEE, 2006.

[SPT04] A. Di Stefano, G. Pappalardo, and E. Tramontana. An Infrastructure for Runtime Evolution of Software Systems. In *Proceedings of the Symposium on Computers and Communications*, pages 1129–1135. IEEE, 2004.

[SSH⁺05] J. Shen, X. Sun, G. Huang, W. Jiao, Y. Sun, and H. Mei. Towards a Unified Formal Model for Supporting Mechanisms of Dynamic Component Update. In *Proceedings of the European Software Engineering Conference held jointly with the International Symposium on Foundations of Software Engineering*, pages 80–89. ACM, 2005.

[Ste90] G. L. Steele. *Common LISP: The Language*. Digital Press, 2. edition, 1990.

[TBNP08] E. Tempero, G. Bierman, J. Noble, and M. Parkinson. From Java to UpgradeJ: an empirical study. In *Proceedings of the International Workshop on Hot Topics in Software Upgrades*, pages 1–5. ACM, 2008.

[TFH05] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2. edition, 2005.

[TSCI01] M. Tatsubori, T Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 236–255. Springer, 2001.

[TVJ+01] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. Nørregaard Jørgensen. Dynamic and Selective Combination of Extensions in Component-Based Applications. In *Proceedings of the International Conference on Software Engineering*, pages 233–242. IEEE, 2001.

[VEBD05] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Influence of type systems on dynamic software evolution. Technical Report CW 415, Katholieke Universiteit Leuven, 2005. `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW415.pdf`.

[VEBD07] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *IEEE Transactions on Software Engineering*, 33(12):856 –868, 2007.

[Ven00] B. Venners. *Inside the Java 2 Virtual Machine*. Computing McGraw-Hill., 2000.

[WBA+10] T. Würthinger, W. Binder, Danilo Ansaloni, P. Moret, and H. Mössenböck. Improving Aspect-Oriented Programming with Dynamic Code Evolution in an Enhanced Java Virtual Machine. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 25–29. ACM, 2010.

[Wür11] T. Würthinger. *Dynamic Code Evolution for Java*. PhD thesis, Johannes Kepler University Linz, 2011.

[WWS10] T. Würthinger, C. Wimmer, and L. Stadler. Dynamic Code Evolution for Java. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 10–19. ACM, 2010.

[ZH06] S. Zhang and L. Huang. Formalizing Class Dynamic Software Updating. In *Proceedings of the International Conference on Quality Software*, pages 403–409. IEEE, 2006.

[ZH07] S. Zhang and L. Huang. Type-Safe Dynamic Update Transaction. In *Proceedings of the Computer Software and Applications Conference*, pages 335–340. IEEE, 2007.