Nr.: FIN-04-2011

Flexible Dynamic Software Updates of Java Applications: Tool Support and Case Study

M. Pukall, C. Kaestner, W. Cazzola, S. Goetz, A. Grebhahn, R. Schroeter, G. Saake

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-04-2011

Flexible Dynamic Software Updates of Java Applications: Tool Support and Case Study

M. Pukall, C. Kaestner, W. Cazzola, S. Goetz, A. Grebhahn,
R. Schroeter, G. Saake

Arbeitsgruppe Datenbanken

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Mario Pukall
Postfach 4120
39016 Magdeburg
E-Mail: mario.pukall@iti.cs.uni-magdeburg.de

http://www.cs.uni-magdeburg.de/Technical_reports.html

Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 30.03.2011

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

Flexible Dynamic Software Updates of Java Applications: Tool Support and Case Study

Mario Pukall[1], Christian Kaestner[2], Walter Cazzola[3], Sebastian Goetz[4], Alexander Grebhahn, Reimar Schroeter[1], and Gunter Saake[1]

[1] University of Magdeburg
mario.pukall@iti.cs.uni-magdeburg.de
{alexander.grebhahn, reimar.schroeter}@st.ovgu.de
saake@ovgu.de
[2] Philips-University Marburg
kaestner@informatik.uni-marburg.de
[3] University of Milano
cazzola@dico.unimi.it
[4] TU Dresden
sebastian.goetz@acm.org

Abstract. Software is changed frequently during its life cycle. New requirements come and bugs must be fixed. To update an application it usually must be stopped, patched, and restarted. This causes time periods of unavailability which is always a problem for highly available applications. Even for the development of complex applications restarts to test new program parts can be time consuming and annoying. Thus, we aim at dynamic software updates to update programs at runtime. There is a large body of research on dynamic software updates, but so far, existing approaches have shortcomings either in terms of flexibility or performance. In addition, some of them depend on specific runtime environments and dictate the program's architecture. We present *JavAdaptor*, the first runtime update approach based on Java that (a) offers flexible dynamic software updates, (b) is platform independent, (c) introduces only minimal performance overhead, and (d) does not dictate the program architecture. JavAdaptor combines schema changing class replacements by class renaming and caller updates with Java HotSwap using containers and proxies. It runs on top of all major standard Java virtual machines. We evaluate our approach's applicability and performance in a nontrivial case study and compare it to existing dynamic software update approaches.

1 Introduction

Once a program goes live and works in productive mode its development is not completed. It has to be changed because of bugs and new requirements. In order to maintain a program, it usually must be stopped, patched, and restarted. This downtime is always a problem for applications that must be highly available. But, also for the development of complex applications restarts to test the new

program parts can be time consuming and annoying. This is also true for end-user desktop applications that have to be restarted because patches must be applied [4]; end users prefer update approaches that do not interrupt their tasks. For that reasons, we aim at *dynamic software updates* (DSU), i.e., program updates at runtime.

Even though dynamic languages like Smalltalk, Python, or Ruby natively support runtime program changes, we address Java for several reasons. First, Java is a programming language commonly used to implement highly available applications. Examples are *Apache Tomcat*, *Java DB*, or *JBoss Application Server*. Second, in most fields of application Java programs execute faster than programs based on dynamic languages [11]. Thus, developers often prefer Java over dynamic languages in time-critical scenarios. Amongst others, one reason for the better performance is that Java is a statically typed language. Unfortunately, compilation prevents Java and other statically typed languages such as C or C++ from natively offering powerful instruments for runtime program updates.

Literature suggests a wide range of DSU approaches for Java (see related work in Section 6). The *flexibility* of an approach can be determined by answering the following three questions: Are unanticipated changes allowed (i.e., can we apply requirements for which the running program was not prepared)? Can already loaded classes (including their schema) be changed, and is the program state kept beyond the update? Other quality criteria for a DSU approach are the caused *performance* overhead, the influence on the *program architecture* and the *platform independency*. We believe that it is impossible to prepare an application for all potential upcoming requirements. Furthermore, only offering modifications of not previously executed program parts while disregarding the executed parts (e.g., already loaded classes) restricts the application of program changes. In addition, state loss and major performance overhead are unacceptable in many scenarios as well. Next, we argue that DSU approaches should not dictate the program's architecture, i.e., they should be capable of being integrated into the program's natural architecture (different application domains might require different architectures). Last but not least, runtime update approaches should not force the customer to use a specific platform for program execution, e.g., to use a Windows based Java Virtual Machine even though the customer only runs Linux based machines. For all these reasons, we aim at (a) flexible, (b) platform independent, and (c) performant runtime update approaches that (d) do not affect the program's natural architecture. However, we do not (yet) aim at a solution that fully ensures *consistency* regarding runtime semantics (which, to our best knowledge, is not supported by any existing DSU approach which is applicable in real world scenarios). In other words, *our goal is to provide Java with the same runtime update capabilities known from dynamic languages.*

Researchers spent a lot of time to overcome Java's shortcomings regarding runtime program adaptation. Approaches like *Javassist* [6,7] and *BCEL* [8] allow to apply some unanticipated changes, but only to program parts that have not been executed yet. In contrast, *Steamloom* [17], *Reflex* [41], *PROSE* [29], *DUSC*

	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, Superinterface, Subinterface, 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. Language Constructs of Java 1.6 [13].

[31], *AspectWerkz* [3], *Wool* [36], or *JAsCo* [43] allow unanticipated changes even of executed program parts; however, Steamloom, Reflex, PROSE, AspectWerkz, Wool, and JAsCo do not support class schema changing runtime updates. Although DUSC allows class schema changes the program loses its state. Another dynamic software update approach is *JRebel* [19] which puts abstraction layers between the executed code and the JVM. It enables class schema changes except from modifications of the inheritance hierarchy. Kim presents in [21] a DSU approach based on proxies which, similar to JRebel, only enables schema changes that not affect the inheritance hierarchy.

We present *JavAdaptor*, the first (to our best knowledge) dynamic software update approach that fulfills all our quality criteria postulated above: it is flexible, platform independent, performant, and it does not affect the architecture of the program to be updated. To meet the criteria, we utilize Java HotSwap in an innovative way and combine it with class replacement mechanisms. Technically, we update all classes with a changed schema via class replacements and update their callers with the help of Java HotSwap. The key concepts of our solution are class renamings (to replace classes) and containers respectively proxies (to avoid caller class replacements). Furthermore, we contribute a discussion of desired properties for DSU approaches and a detailed survey off related approaches and their trade offs. Last but not least, we demonstrate the applicability of our approach in a nontrivial real world scenario and show that the performance drops are minimal.

2 Motivating Example

Program maintenance is not a trivial task, which usually affects many parts of a program. Depending on the requirements, it ranges from single statement modifications to complex structural modifications, i.e., it might affect all language constructs of Java as listed in Table 1.

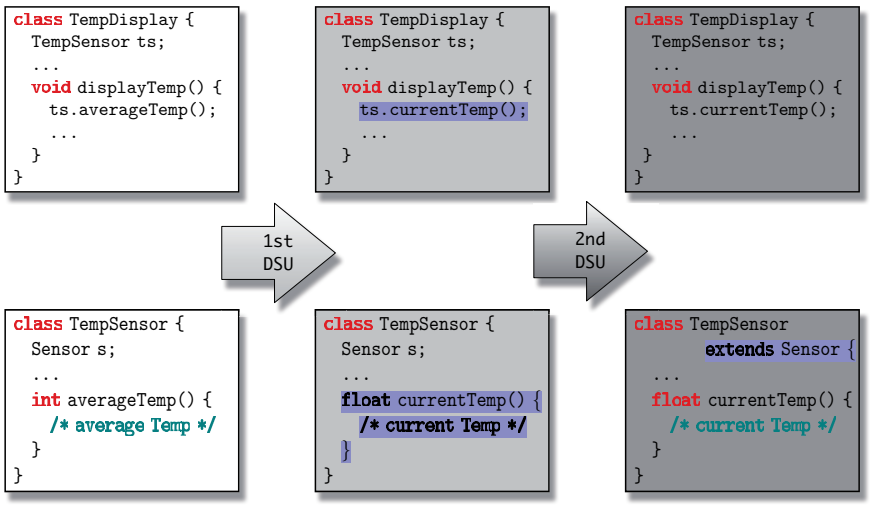

Fig. 1. Weather station.

The weather station program depicted in Figure 1 examplifies that even simple program changes can affect many parts of a program. The weather station program consists of 2 classes. One class (`TempSensor`) measures the air temperature while the other class (`TempDisplay`) is responsible for displaying the temparature. 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 nonstop 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, method `averageTemp` of class `TempSensor` must be replaced by method `currentTemp` which requires to change the class schema. Second, in order to execute the new algorithm method `displayTemp` of class `TempDisplay` must be reimplemented. Short time after applying the new 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, statement `extends Sensor` has to be applied to class `TempSensor`. Additionally, member `s` of original class `TempSensor`

has to be removed because superclass `Sensor` let it become useless. However, changing the program code is only the first step toward an updated application. In addition, all objects that exist in the program must be also updated to let them access the new program parts as well as to keep the program state.

Even if the required program changes seem to be simple, they affect many different parts of the program (i.e., points 1-6 of Table 1). Therefore, we search for a new mechanism in Java that allows to change every part of a program at runtime without anticipating the changes.

3 The Java Virtual Machine

In order to understand what is provided or possible in Java and what challenges remain regarding runtime adaptation, it is necessary to understand the standard design of Java's runtime environment – the Java virtual machine (JVM)[45]. As shown in Figure 2, a Java program is stored in the *heap*, in the *method area*, as well as on the *stacks* of the JVM. Within the heap the runtime data of all class instances are stored [25]. In case a new class instance has to be created, the JVM explicitly allocates heap memory for the instance, whereas the *garbage collector* cleans the heap from data bound to class instances no longer used by the program. Unlike the heap, the method area stores all class (type) specific data such as runtime constant pool, static field information and method data, and the code of methods (including constructors) [25]. The stacks contain the currently executed program parts.

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 changes such as depicted in Figure 1 which also include method replacements 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. In order to disallow the developer class schema changing updates, the JVM enforces a strict class loading concept. To load a class, the JVM requests the following basic class loaders (in this order): (a) the *bootstrap* class loader (root class loader – loads system classes), (b) the *extension* class loader (loads classes of the extension library), and (c) the *application* class loader (loads classes from classpath). The first class loader in this hierarchy that is able to load the requested class will be finally bounded to this class, i.e., none of the other class loaders is allowed to load or reload this class. The only way (beside customized class loaders that we will discuss in later sections) to reload (update) a class with a changed schema is to unload the old class version, which is only possible 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.

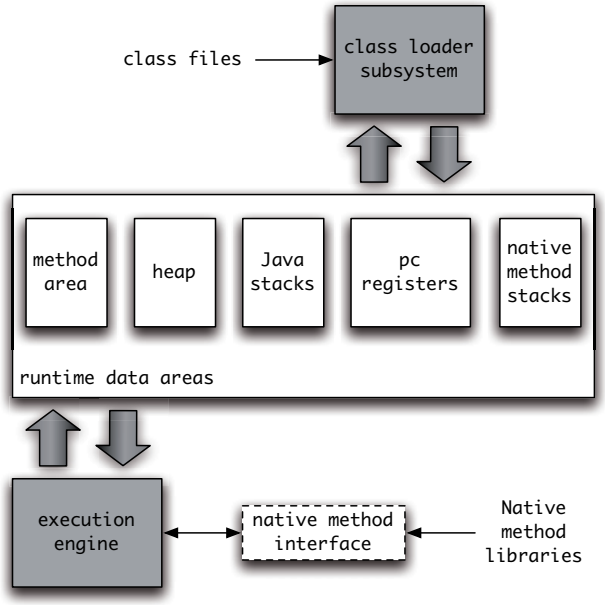

Fig. 2. Program representation – HotSpot JVM [45].

Java HotSwap. Despite the insufficient native runtime adaptation support of the JVM there is one feature that provides some simple runtime update capabilities – called *Java HotSwap* [9]. It is provided by the Java Virtual Machine Tool Interface (JVMTI) [39] and allows to replace the body of a method (which partly covers points 4 - 6 of Table 1) while the program is running. Even if HotSwap is not a standard feature, it is implemented by all major Java virtual machines commonly used in production, i.e., the HotSpot JVM, the JRockit JVM, and IBM's JVM.

The class data restructuring via Java HotSwap consists of the following steps: First, an updated version of a class is loaded into the JVM. It contains the new method bodies. Second, it is checked if old and new class version share the same class schema. Third, the references to the constant pool, method array, and method objects of the old class are successively (in the given order) redirected to their (up-to-date) counterparts within the updated class. After this is done, all corresponding method calls refer to the redefined methods. Unfortunately, Java HotSwap (and other features of JVMTI) neither allows to swap the complete class data nor removing or adding methods, i.e., it does not allow class schema changes.

4 Dynamic Software Updates via JavAdaptor

Having described the shortcomings of Java's runtime environment, i.e., the JVM, regarding flexible runtime program updates, we present *JavAdaptor* which overcomes the limitations of the Java VM and adds flexible DSU to Java while not causing platform dependencies, architecture dependencies, and significant performance drops. It combines Java HotSwap and class replacements, which are implemented via containers and proxies.

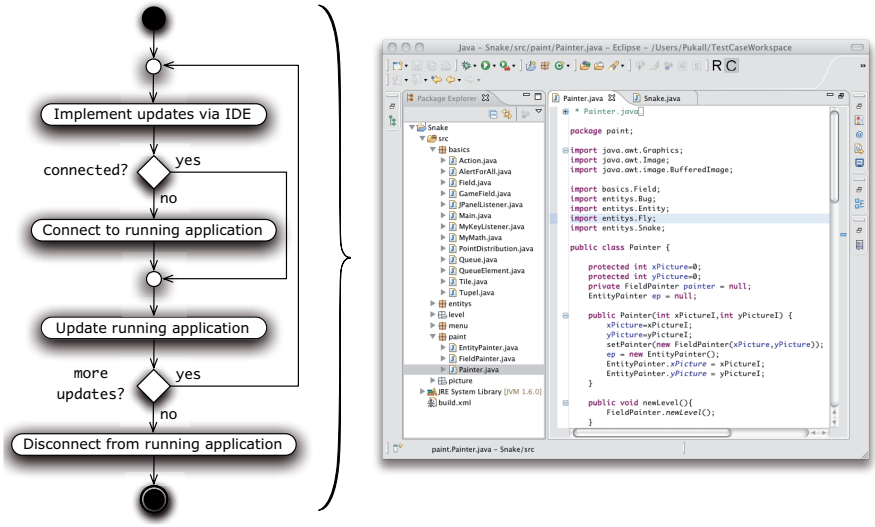

Fig. 3. Update process.

4.1 Tool Description and Demonstration

Before we describe the concepts of our DSU approach, let us illustrate the general architecture and update process of JavAdaptor.

Tool Description. Figure 3 describes JavAdaptor from the developers point of view. The current implementation of our tool comes as a plug-in which smoothly integrates into the *Eclipse 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 static software development process, i.e., the developer implements the required functions using the Eclipse IDE and compiles the sources. This ensures type-safety because of the static type checking done by the compiler.

When the developer decides to update the running application, JavAdaptor establishes a connection to the JVM executing the application (see Figure 4).

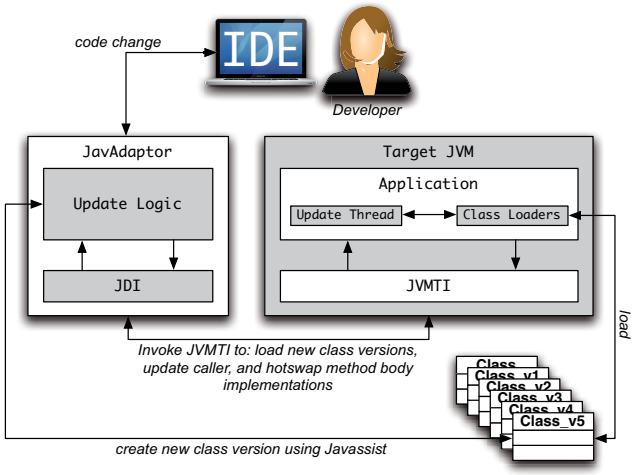

Fig. 4. Dynamic software update architecture.

In more detail, it connects to the JVM's *Java Virtual Machine Tool Interface* (JVMTI) which is used to control the JVM [39] (accessible from outside the JVM through the *Java Debug Interface* which is part of the *Java Platform Debugger Architecture* [40]). Once the update process is triggered, JavAdaptor prepares the classes changed within Eclipse so that they can be applied to the running application. The required bytecode modifications are performed by Javassist.[5] 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. After the update, JavAdaptor disconnects from the application. The described process can be repeated as often as required.

Tool Demonstration. Because abstract descriptions on the usage of tools are sometimes hard to understand and do not reflect the reality well, we created a tool demonstration showing JavAdaptor in action. Concretely, we used JavAdaptor to update the well-known arcade game *Snake* at runtime. The update consists of 4 different steps which each add new functions to the (at startup) very basic game. It required to redefine existing methods, to add new methods and fields, and even to update inheritance hierarchies. That is, the demonstration covers all kinds of updates essential to flexibly update running applications. For more information about our tool demo see [34]. The corresponding demo video is available on YouTube.[6]

[5] http://www.csg.is.titech.ac.jp/~chiba/javassist/
[6] http://www.youtube.com/watch?v=jZm0hvlhC-E

In the following, we describe the basic mechanisms how JavAdaptor changes applications running in the target JVM, namely class replacements using containers and proxies.

4.2 Class Reloading

As stated in Section 3, the JVM disallows updating an already loaded class when the update alters the class schema. In order to circumvent these restrictions we perform class replacements (updates) through *class renaming*. As exemplified in Figure 5, 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.

Fig. 5. Class renaming.

Listing 1.1 sketches how class loading based on class renaming is implemented in JavAdaptor. The renamed and updated class (here class `TempSensor_v2` from our motivating example depicted in Figure 1) is created by our adaptation tool (using the source level API of Javassist to manipulate the bytecode in Lines 7-8). In the next step, the adaptation tool invokes method `loadClass` (Line 12) of class `UpdateHelper` which resides in the update thread added to the target application on application start. By invoking `loadClass` within the target application, the new class version is loaded by the same class loader that loaded the original class (Listing 1.2, Line 20), which ensures that our DSU approach is compatible with any application employing multiple class loaders (e.g., component based applications).

4.3 Caller Side Updates

As demonstrated above, our class reloading mechanism allows us to load a new version of an already loaded class even if the class schema has changed. However, the mechanism only triggers the loading of the updated class. To let the class become part of program execution, all references to the original class have to be changed to point to the new class version. For the sake of clarity, we will name the classes which hold references to classes to be reloaded (updated) *callers* and the classes subject to updates *callees*. In addition, the terms *caller side* and *callee side* cover the class itself as well as all its instances.

When it comes to short-lived objects, such as local variable `local` of class `TempDisplay` (Figure 6), only method body redefinitions are required to refer

Listing 1.1. JavAdaptor – class reloading.

```
1  class ClassUpdateLoader {
2    VirtualMachine targetJVM;
3    ...
4    void replaceClass(String oldClassName) {
5      if(isOldClassLoaded) {
6        ...
7        CtClass c = classpool.getCtClass(oldClassName);
8        c.replaceClassName (oldClassName, newClassName);
9        ...
10       ReferenceType refT = targetJVM.classesByName("UpdateHelper").get(0);
11       ObjectReference uHelper = refT.instances(0).get(0);
12       uHelper.invokeMethod(t, loadClass, args[newClassName], options);
13     }
14   }
15 }
```

Listing 1.2. Target VM – class reloading.

```
16 class UpdateHelper extends Thread{
17   ClassLoader origClassLoader;
18   ...
19   void loadClass(String newClassName) {
20     origClassLoader.loadClass(newClassName);
21   }
22 }
```

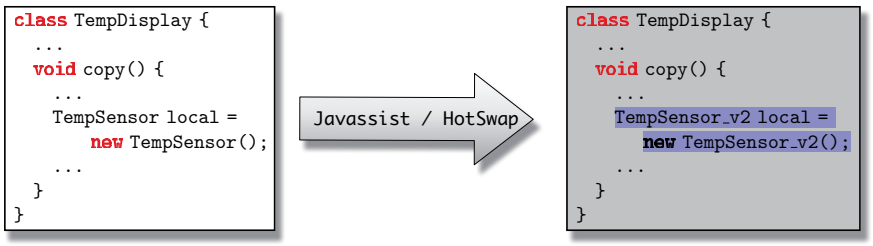

Fig. 6. Caller side updates in case of short-lived objects.

to the new class version. This is because with each method execution the local variables are newly created. Thus, after redefining a method, such as depicted in Figure 6, the local variables created during method execution will be of type of the updated class (here of class `TempSensor_v2`). Those updates can be easily located and applied using the source level API of Javassist and Java HotSwap.

A snippet of the corresponding update code is depicted in Listing 1.3. For each application class, JavAdaptor checks whether the class references the class to be updated. Technically, all classes referenced by the caller side are requested using Javassist method `getRefClasses` (Line 4). If references to short-lived objects of type of the old callee class (here of class `TempSensor`) are found (Line 9), they are redirected method by method to the updated class (Lines 10-17). After this is done, the updated caller method is redefined using Java HotSwap (Line 19).

Listing 1.3. JavAdaptor – caller update in case of short-lived objects.

```
1  class CallerUpdateShortLived {
2    ...
3    void detectAndUpdateCaller(CtClass caller) {
4      Collection col = caller.getRefClasses();
5      Iterator colIterator = col.iterator();
6      ...
7      while(colIterator.hasNext) {
8        CtClass callee = (CtClass) caller.getNext();
9        if(callee.getName.compareTo(oldClassName) == 0) {
10         CtMethod[] methods = caller.getDeclaredMethods();
11         ClassMap classMap = new ClassMap();
12         classMap.put("oldClassName", "newClassName");
13         ...
14         for(int i = 0; i < methods.length; i++) {
15           ...
16           methods[i].setBody(bodyCopy, classMap);
17         }
18         ...
19         targetVM.redefineClasses(callerClass);
20         ...
21       }
22     }
23   }
24 }
```

Different from references to short-lived objects, 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 must be handled in a different way. Those updates require four steps: (1) *caller detection*, (2) *instantiation of the updated callee class*, (3) *callee side state mapping*, and (4) *reference updates*.

Listing 1.4. JavAdaptor – caller detection.

```
1  class CallerUpdateLongLived {
2    ...
3    List<ClassObjectReference> detectCallers() {
4      ReferenceType refL = targetJVM.classesByName(oldClassName);
5      List<ObjectReference> oRefL = refL.instances(0);
6      ...
7      oRefL.get(i).getReferringObjects(0);
8      ...
9
10   }
11 }
```

Caller Detection. In order to replace the references to instances of the original callee class by instances of the new callee class version (as required for class `TempSensor` from our motivating example) we have to detect all callers and their instances that refer to long-lived objects of the original callee class. The JVMTI supports this operation. A snippet of the caller detection implementation is de-

picted in Listing 1.4. First, the class object of the old callee class is retrieved from the target JVM (Line 4). This object is used to get all instances of the old callee class (Line 5) via reflection. Again, using the instances all callers are retrieved (Line 7). This includes even callers whose global fields are of type of a super class the old class extends, which is possible because the function requests the objects runtime type and not the static type. In addition, JavAdaptor searches all application classes for class and instance fields of type of the old callee class (using Javassist method `getRefClasses`). This is necessary in order to detect even caller classes which are not yet loaded, instantiated, or whose instances do not refer to the callee side because the corresponding class or instance fields are not yet initialized.

Callee Class Instantiation. In the next update step, JavAdaptor creates for each instance of the original callee class an instance of the new class version (here of class `TempSensor_v2` from our motivation). The new instances will be used later on to replace the instances of the old class and, thus, to update the caller side (i.e., class `TempDisplay`).

Listing 1.5. JavAdaptor – instantiation.

```
1 class UpdateInstantiation {
2   ClassObjectReference updateHelper;
3   ...
4   void createInstance(String newClassName) {
5     ...
6     updateHelper.invokeMethod(t, newInst, args[newClassName], options);
7   }
8 }
```

Listing 1.6. Target VM – instantiation.

```
9  class UpdateHelper extends Thread {
10   Unsafe unsafe;
11   ClassLoader applClassLoader;
12   ...
13   Object newInst(String cName) {
14     Class c = Class.forName(cName, false, applClassLoader);
15     return unsafe.allocateInstance(c);
16   }
17 }
```

Again, the instantiation is triggered by our adaptation tool. The corresponding code is depicted in Listing 1.5. Method `createInstance` of our update tool takes as argument the name of the new class version and invokes method `newInst` of class `UpdateHelper` in the target application which creates an instance of the new class. Listing 1.6 shows a code snippet of method `newInst` of the helper class at application side. Via method `forName` we retrieve the class object of the updated class (Line 14). Then we call method `allocateInstance` of class `sun.misc.Unsafe` which performs the instantiation. The reason why we use `sun.misc.Unsafe` instead of method `newInstance` of class `Class` for instantiation is that it prevents us from initializing the objects twice, i.e., it would require

to initialize the objects when they are created and again when they get the state from their outdated counterparts, which would be inefficient.

Listing 1.7. Target VM – callee state mapping.

```
1 class UpdateHelper extends Thread {
2   ...
3   void mapState(Object oldObj, Object newObj) {
4     ...
5     newObj.setValue(newField, oldObj.getValue(oldField));
6     ...
7   }
8 }
```

Callee Side State Mapping. Having finished the instantiation step, JavAdaptor has to map the state from old to corresponding new instances. In our example, this means to map the state from instances of old class `TempSensor` to instances of class update `TempSensor_v2`. Due to the simplicity of one-to-one mappings (mappings of values from fields that exist in both class versions) and mappings where either fields are removed or added they can be executed automatically. However, for more complex (indefinite) mappings, e.g., mappings where the type of a field differs between old and new class but the field name remains the same, a mapping function must be manually defined by the user. Listing 1.7 sketches how our adaptation tool implements one-to-one mappings.

Reference Updates. Finally, once for each instance of the original callee class an instance of the new class version has been created and initialized with the state of its outdated counterpart, JavAdaptor updates the caller side. That is, all instances of the original callee class (such as class `TempSensor` from our motivation) have to be replaced by the instances of the new callee class (here class `TempSensor_v2`). Unfortunately, updated and outdated callee class are not type compatible, thus, objects of the updated class cannot be assigned to fields of type of the outdated class (such as required to update field `ts` of caller class `TempDisplay`).

Containers. To solve the type incompatibility problem, we use containers whose usage is exemplified in Figure 7. Before program start, JavAdaptor prepares the program for the container approach, i.e., it adds field `cont` (Line 11) to each class in the program using Javassist. The container field does not affect program execution as long as no callee of the caller class has to be replaced. To replace a callee instance referenced by the caller class, the program has to be changed as depicted in the right part of Figure 7. First, JavAdaptor creates a container class (via Javassist) used to store instances of the new callee class. Second, our tool assigns a newly created callee instance to an instance of the container. The container instance is then assigned to field `cont` within the caller

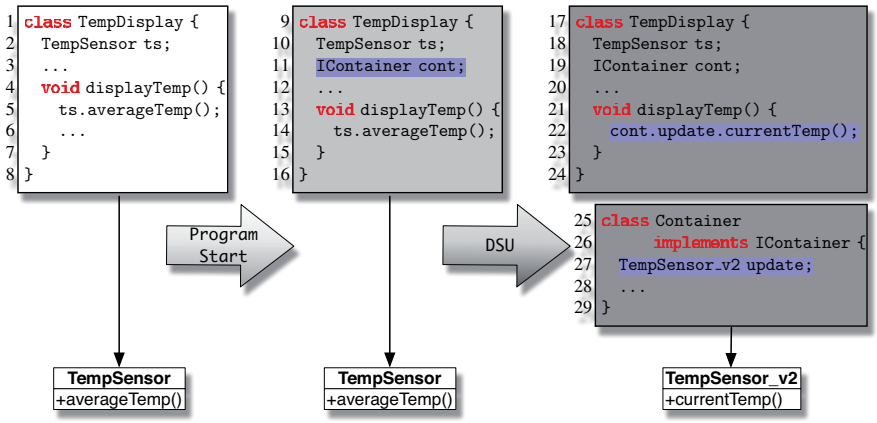

Fig. 7. Containers.

class. Third, the tool redirects all accesses of the old callee instance to the updated callee instance located in the container (Line 22), i.e., the tool redefines all method bodies (using Javassist) in which the old callee instance is accessed and swaps the resulting method bodies via HotSwap.

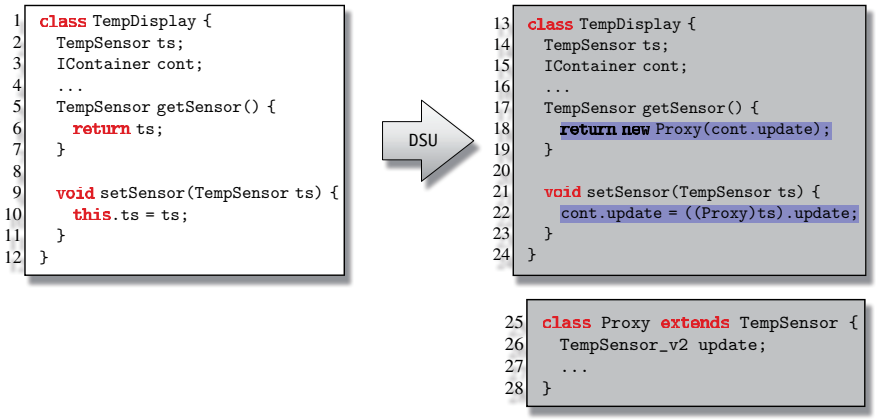

Fig. 8. Proxies.

Proxies. The basic container approach described in Figure 7 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 8, 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 8). 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`. Technically, we use method *allocateInstance* from class `sun.misc.Unsafe` for the proxy instantiation, because it allows us to create proxy instances even if the proxies super class has no default constructor. 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 8 (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`.

Listing 1.8. Bytecode modifications proxy: return.

```
1  TempSensor getSensor() {
2     0 aload_0
3     1 astore_1
4     2 aconst_null
5     3 astore_2
6     4 aload_1
7     5 getfield #15 <TempDisplay.fieldContainer1265725244704>
8     8 checkcast #17 <TempDisplay_Cont_1>
9    11 getfield #21 <TempDisplay_Cont_1.ts>
10   14 astore_2
11   15 aload_2
12   16 invokestatic #27 <TempSensor_Proxy_1.newInst>
13   19 checkcast #29 <TempSensor>
14   22 areturn
15 }
```

Proxy Bytecode Modifications. Up to this point, most of the required bytecode modifications described above could be processed using the source level

API of Javassist which makes bytecode modifications easy to handle. However, the modifications required to apply proxies exceed the power of Javassist's source level API. The source level API cannot terminate the type of local variables referenced through the method's *local variable table*. Because parameters are stored in local variables by default, it is not possible to apply the code to unwrap them using the source level API. The same problem occurs when locally stored objects that have to be returned must be wrapped by a proxy. For that reasons, we manage the application of proxies manually, i.e., with the bytecode level API of Javassist.

Listing 1.9. Bytecode modifications proxy: parameters.

```
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 }
```

Listing 1.8 shows the bytecode modifications (here of method `getSensor` of example class `TempDisplay`) required to wrap returned objects. First, we call method `newInst` (Line 12) of the Proxy class which takes as parameter an object of the updated callee class (here of class `TempSensor_v2`), wraps the object 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).

How to modify the bytecode in order to unwrap proxy based parameters (here of method `setSensor` of example class `TempDisplay`) is depicted in Listing 1.9 (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 updated class 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).

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 2. This example only consists of one single class update and the corresponding caller side update. However, JavAdaptor does not only

allow the developer to update a single class but multiple classes in one step, which is essential to 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.

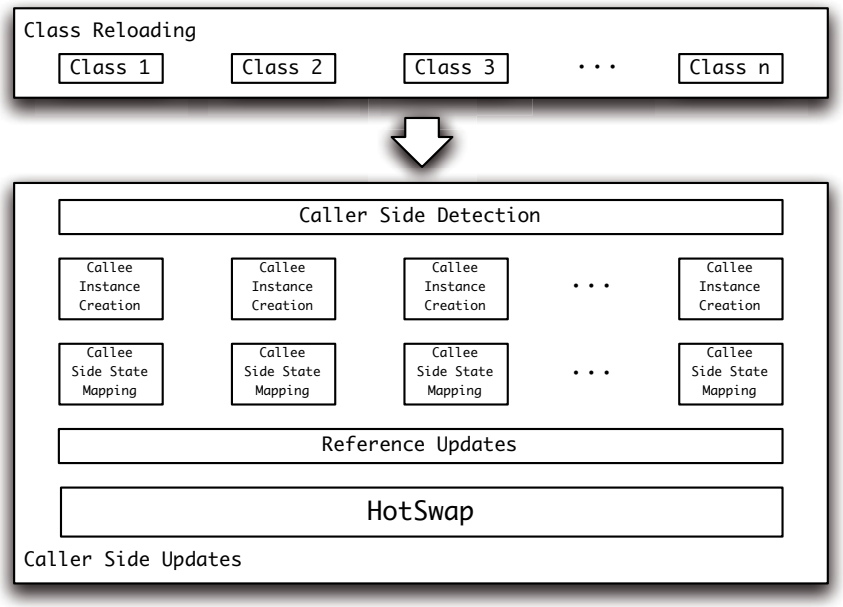

Fig. 9. Concurrent multiple class updates.

Figure 9 sketches how JavAdaptor handles concurrent updates of multiple classes. At first, JavAdaptor reloads all classes with changed schemas (as described in Section 4.2). Afterwards, it identifies all classes (callers) with references to the classes to be reloaded (see Section 4.3). 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 4.3 and 4.3). If this is done, JavAdaptor updates all references conform to the workflow described in Section 4.3. 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.

In summary, 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 remain untouched which minimizes the influence of the update on the running program.

5 Evaluation

Our goal was to develop an update approach which allows running Java applications to be updated in every possible way (a feature only known from dynamic languages). In addition, the approach should not introduce performance drops.

In order to check whether JavAdaptor meets the goals, we applied it to a nontrivial case study. To simulate a real world scenario which requires flexible runtime updates, we proceeded as follows. We chose a reasonable application to update, which was *HyperSQL*[7] (HSQLDB) amongst others used by *OpenOffice* (we chose HSQLDB because it is a database management system for which runtime adaptation promises benefits of no-downtime, it is entirely written in Java, and an open source application whose source code is available for the latest program version and earlier versions). We started version 1.8.0.9 of it downloaded from the HSQLDB website and applied all changes to evolve it to the next version 1.8.0.10 without shutting down the application. After program start, we ran the open source database benchmark *PolePosition*[8] in order to generate and query some data which ensured that HSQLDB was fully activated and deployed.

5.1 Dynamic Software Updates

The new version of HSQLDB (released 9 month after version 1.8.0.9 came out) 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. The updates affected many language constructs (points 1-7 of Table 1). In case of 21 out of 33 classes the changes did not affect the class schema, i.e., the changes could be applied by our tool solely using Java HotSwap. Apart from that, 12 classes were affected by schema-changing

[7] http://hsqldb.org/
[8] http://polepos.sourceforge.net/

Replaced Class kind of Update	Caller Updates Short-lived Obj. (places)	 Container (places)	 Proxy (places)
FontDialogSwing structural	8 (9×)	0 (-)	0 (-)
HsqlDatabaseProperties functional	11 (98×)	2 (25×)	11 (23×)
LockFile functional	1 (9×)	10 (5×)	11 (47)
LockFile$HeartbeatRunner functional	2 (2×)	0 (-)	0 (-)
Logger structural	22 (93)	3 (93×)	3 (4)
NIOLockFile changed inheritance hierachy	0 (-)	0 (-)	0 (-)
ScriptReaderZipped functional	3 (3×)	0 (-)	0 (-)
SimpleLog structural	9 (105×)	3 (27×)	0 (-)
Token structural	5 (671×)	0 (-)	0 (-)
Trace structural	80 (1306×)	0 (-)	0 (-)
Transfer structural	4 (6×)	0 (-)	0 (-)
View functional	3 (37×)	3 (13×)	3 (16×)

Table 2. Required class replacements.

program modifications. JavAdaptor replaced them using class replacements. The state mappings that came along with the replacements span one-to-one mappings, added, and removed fields, i.e., they were automated by JavAdaptor. Table 2 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. Inheritance hierarchy updates as well as updates that involve nested classes are supported by JavAdaptor. However, Table 2 also provides 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 places 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 %) updates because of references to short-lived callee objects (via Java HotSwap) were required to update the callers. 21 caller classes (10.7 %) had to be updated through containers. 28 caller class updates (14.2 %) required proxies.

In order to verify that HSQLDB was still correctly working (in a consistent state) after the update, we reran the PolePosition benchmark. In the result, HSQLDB 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*[9] into the running application and checked what classes/methods were

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

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.

5.2 Performance

Having demonstrated JavAdaptor's ability to update complex real world applications, it is time to take a look at the performance penalties induced by our tool, i.e., the execution speed of the changed program parts.

To measure the performance penalties of the program updates we proceeded as follows. We ran the PolePosition benchmark (mentioned above) immediately after runtime updating HSQLDB to version 1.8.0.10 and compared the results with the benchmark results of HSQLDB version 1.8.0.10 not updated at runtime. We could not measure any statistically significant difference, i.e, the benchmark results of the HSQLDB instance updated at runtime were as good as the results of the HSQLDB instance not updated at runtime. In other words, the runtime updates performed by us did not affect the performance of HSQLDB in a measurable way.

However, even if we did not measure performance penalties because of our runtime update approach in a real world scenario, we assumed that our approach does not come entirely without performance overhead in some borderline cases. To get evidence about this assumption, we additionally implemented a micro 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., updated callees).

Type		**Original** ns	**Callee Update** ns	**cons. Callee Update** ns	**Caller Update** ns
invoke Method void m(Callee)	load 0×	0 (+0,0137)	31 (±0,1089)	30 (±0,1150)	0 (+0,0078)
	load 1×	70 (±0,0507)	86 (±0,1605)	85 (±0,1531)	62 (±0,0493)
	load 2×	140 (±0,0762)	145 (±0,1580)	146 (±0,1543)	119 (±0,0605)
invoke Method Callee m()	load 0×	0 (+0,0019)	30 (±0,1140)	31 (±0,1201)	0 (+0,0020)
	load 1×	69 (±0,0493)	84 (±0,1206)	85 (±0,1158)	58 (±0,0562)
	load 2×	139 (±0,0620)	143 (±0,1141)	142 (±0,1072)	117 (±0,0615)
invoke Method Callee m(Callee)	load 0×	0 (+0,0059)	62 (±0,1609)	63 (±0,1598)	0 (+0,0062)
	load 1×	70 (±0,0537)	99 (±0,1444)	99 (±0,1396)	62 (±0,0431)
	load 2×	140 (±0,0781)	158 (±0,1704)	159 (±0,1765)	120 (±0,0830)

Table 3. Performance overhead when using proxies.

To get reliable results, we ran thousand samples of one million invocations of all major invocation types and for each calculated the average access time

in nanoseconds. For containers and local updates, no statistically significant performance overhead was measurable (calculated through a one-way analysis of variance), i.e., programs updated using containers and local updates 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.

In Section 4.3, we described the need for proxies to avoid implicit caller replacements in case the callee appears to be an argument of a caller method, a returned object, or both. To measure the proxy performance with our micro benchmark, we again ran thousand samples of one million method invocations and calculated the average access time. The results of our benchmark (average and, in brackets, the confidence interval with a probability value of 95%) are shown in Table 3: proxies induce slight constant overhead compared to the original program (between 30 and 63 nanoseconds per method call). In order to get to know how the results scale, we put some workload on the methods and let them compute $\sin x$ one time respectively two times. As shown in Table 3, even small workload as introduced by the equation consumes clearly more computing power than proxy instrumentation. Furthermore, updating the caller via class replacement recovers the original performance of the program (see Table 3).

All in all, the results of our case study and the micro benchmark confirm that runtime program changes by JavAdaptor produce only minimal performance overhead. Only proxies produce a measurable overhead. Caller updates through local changes and containers do not cause measurable performance drops.

5.3 Update Speed

Even if the contributions of our current JavAdaptor implementation are others than applying updates the fastest way, we evaluated how well JavAdaptor performs in this regard. That is, we measured the time JavAdaptor pauses the application during the update process in order to avoid program inconsistencies. Our measurements base on two different programs representing different application scenarios.

At first, we measured the time required to update our HSQLDB case study under different conditions. With our first test, we measured the time period required to update HSQLDB with an empty database (i.e., without any data object stored), which was 1070 milliseconds. In further tests, we ran the PolePosition benchmark creating thousands, ten thousands, and hundred thousands of data objects before the update. The corresponding update times ranged from 1249 milliseconds (thousands of data objects), over 1658 milliseconds (ten thousands of data objects), to 5363 milliseconds (hundred thousands of data objects), which seems to be not outstanding fast but sufficient in many scenarios. By contrast, as shown in Table 4, restarts and reinitializations of HSQLDB (e.g., filling caches, reloading data objects, creating views, creating users, etc.) as we simulated them using PolePosition took more time. However, as shown in Table 4 column 5, what may be a bottleneck of our current JavAdaptor implementation is method `getReferringObjects`, which execution times notably increase

the more objects are present in the JVM, even if the number of objects to be updated remains unchanged (we will discuss in later sections how to solve this problem and moreover how to avoid time periods of unavailability during the update at all).

Pole Position Config.	**Restart**	**DSU with JavAdaptor**		
		Overall (ratio to restart)	Mapping, HotSwap, Reference updates (ratio to overall)	getReferringObjects (ratio to overall)
Original	6632	1249 (18,83 %)	1147 (91,83 %)	102 (8,17 %)
Original 10x	16563	1658 (10,01 %)	1335 (80,52 %)	323 (19,48 %)
Original 100x	86664	5363 (6,19 %)	2814 (52,47 %)	2549 (47,53 %)

Table 4. Times of unavailability: restart vs. JavAdaptor (in milliseconds).

The other application for which we measured the update times was the *Snake* demo we briefly described in Section 4.1 and presented in [34]. Compared to the update of HSQLDB, 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 on YouTube[10] suggests, the update times are rather short ranging 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 (even if currently other DSU approaches such as presented in [46] and [15] may offer shorter update times). However, high speed updates were not yet in our scope. Therefore, our current JavAdaptor implementation is not optimized for them. But of course, optimizations to the update speed are subject to future versions of JavAdaptor.

6 Related Work and Comparison

In this section we provide an overview of recent work to overcome Java's limitations regarding dynamic software updates. For better comparability and because of the broad range of related work ranging from theoretical to practical solutions, we focus on practice-oriented approaches which, like JavAdaptor, can be directly applied in real world scenarios. We group the related work into three groups based on their main strategies: *Customized Java Virtual Machines*, *Customized*

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

Class Loaders, and *Wrappers*. For each group we discuss the general mechanism and some representative approaches.

In addition, we evaluate the quality of JavAdaptor and of the related work based on the criteria given in Section 1. That is, we analyze an approach's *flexibility*, *platform independency*, *performance* and its influence on the *program architecture*. We chose the criteria because they let us describe the differences between the approaches. For instance, considering consistent program update support would be irrelevant, because only approaches, such as presented in [44] and [22], address program consistency theoretically, but they are not yet available as practical tool. Furthermore, the criteria align with our goals presented in Section 1. We derived the criterion *flexibility* from the fact that static software development allows the developer to change a program in any way, no matter *when* and *where* the changes must be applied. Runtime update approaches should provide the same flexibility in order to cover all update scenarios. We further choose *platform independency* because platform independence is one of the reasons for the success of Java, i.e., DSU approaches should not cause dependencies to specific JVM implementations. In Section 1, we argued that Java's *performance* in terms of program execution speed is better than the performance of dynamic languages, which natively provide flexible runtime updates. 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. 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). Finally, we pick up the *program architecture* criterion because in software development there is no such thing like "one architecture fits all scenarios". As already mentioned in Section 1, different scenarios require different architectures. Thus, DSU approaches should not restrict the usage of different architectures. However, different criteria might be of different importance to different stakeholders. For instance, users might emphasize *flexibility* whereas administrators might attach great importance to *platform independence*. That is, in order to satisfy the stakeholders, a DSU approach must fulfill all mentioned criteria.

6.1 Customized Java Virtual Machines

As mentioned in Section 3, 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. For instance, Malabarba et al. [27] add dynamic class loaders to their *Dynamic Virtual Machine* (DVM) for this purpose. *JDrums* [35] is a JVM that uses handles to decouple classes and objects from each other in order to reload classes. The *Jvolve* VM [38] decouples classes using meta-objects that can be easily changed to refer to updated classes. In addition to Java HotSwap, which allows the developer to redefine methods bodies of already loaded classes, Dmitriev [9] patched the Hotspot JVM in such way that it

supports even class schema changes. Unfortunately, unlike Java HotSwap, this feature never made it into a standard JVM.

Flexibility. All in all, customized Java virtual machines perform well when it comes to flexibility. They allow unanticipated changes of virtually all parts of a program. Furthermore, they all provide mechanisms to keep the program state beyond the update. Customized JVMs provide this flexibility because the update mechanism is implemented within the JVM itself and not at application level which otherwise would complicate or prevent flexible updates.

Platform Independency. Even if virtual machine customization seems to be the most natural way to enhance Java's runtime update capabilities (because it does not require to operate at application level to apply the update approach), different problems arise from it. First of all, there is a standard which precisely defines the functionality and structure of a JVM [25]. Changing the standard in order to add dynamic software updates is difficult because it would require to change all existing JVM implementations. Thus, there are only slight chances that DSU becomes a standard. However, as long as DSU is not part of the JVM specification it must be added via patches. One problem with JVM patches is that they base on a specific JVM implementation and might not be applicable to other JVMs. In addition, each new release of the JVM must be patched again. This might be difficult (eventually impossible) in case the JVM implementation has largely changed in the new JVM version. Last but not least, companies rather prefer standard (certified) JVMs over customized ones to run their applications in productive mode. This is why dynamic software update approaches are needed that operate on top of different standard virtual machines.

Performance. First of all we point out that it is virtually impossible to exactly measure and compare the performance of the referred approaches. Some JVMs are not available for download and those that are available do (partly) support only outdated Java versions (e.g., JDRUMS only executes programs based on Java version 1.2). Thus, we were not able to benchmark them and get meaningful benchmark results. Instead, we searched the literature for information regarding the performance. We found that the four patched JVMs significantly differ in terms of performance (see [38] and [46]). *DVM* [27] executes programs in interpreted mode only, which is commonly known to be slow. *JDrums* [35] aims at lazy updates and uses transformer functions to migrate the state from old objects to their updated counterparts which introduces noticeable constant performance overhead. *Jvolve* [38] immediately updates applications, i.e., it applies the updates in one step and thus avoids considerable performance penalties. Würthinger et al. present in [46] a new and improved version of Dmitriev's JVM patch [9] that comes without any performance overhead.

Program Architecture. As previously described, JVM customization aims at integrating the update mechanisms with the JVM which makes changes to the application architecture unnecessary.

6.2 Customized Class Loaders

As mentioned above, the basic idea of JVM patches is to enhance the JVM with capabilities to reload and thus update classes. In addition to the basic class loaders required to load and run a program, the class loading capabilities of a program can be extended even at application level by customized class loaders [23], which is common technique to load updated versions of already loaded classes or components. For instance, the *OSGi Service Platform* [1] or Oracles *FastSwap* [30] utilize customized class loaders to update components. *Javeleon* [15] allows to flexibly update *NetBeans* based applications and thus uses customized class loaders, too. Zhang and Huang [47] presented *Dynamic Update Transactions* (*DUT*) which also make use of customized class loaders.

Flexibility. Customized class loaders serve the flexibility required to largely update running programs, i.e., they allow to update virtually all parts of a running program in an unanticipated way while preserving the program state. This is true for *Javeleon* [15] and also for *Dynamic Update Transactions* (*DUT*) [47]. In case of the *OSGi Service Platform* [1] the state of a bundle is lost when it is refreshed, though.

Platform Independency. Because customization of class loaders is a standard feature in Java, it can be applied to all standard Java runtime environments. *Javeleon* additionally requires *NetBeans* components for execution.

Performance. One issue with customized class loaders is that they reduce the application performance when applied to JVMs older than version 1.6. This is, because old and updated program parts are loaded by different class loaders which requires poor performing reflection-based version-barrier crossings. Cazzola [5] found out that even simple reflective method invocations (as required for crossing the version barrier) slow down method invocations with a factor of up to 6.5 compared to direct method invocations. More complex version-barrier crossings might cause even higher performance penalties. However, with Java 1.6 this situation relaxed because the related JVM is able to optimize reflective calls.

Program Architecture. Generally, the application of customized class loaders largely affects the application architecture. More precisely, customized class loaders dictate how an application must be designed and thus disallow alternative (tailor-made) designs. *DUT* requires methods that maintain the updates to be present in each class. *Javeleon*, *FastSwap* as well as the *OSGi Service Platform* require the applications to run on top of their infrastructure to be refactored into components (if not already done). This does not only alter the application architecture, it might be also inefficient because even small changes require to replace whole components.

6.3 Wrappers

Another frequently used approach to provide Java with enhanced runtime update capabilities are wrappers (also known as decorators [12]). Wrappers aim at wrapping old program parts in order to update them [33][32][42].

To apply the updates introduced by a wrapper, all clients (callers) of the changed program parts must be updated, too. That is, all references to the original callee must be redirected to the corresponding wrapper instance that wraps the callee. To update the caller side, Gamma et al. [12] suggest that wrapper and wrappee extend the same superclass or (even more flexible) implement the same interface. The application of the wrapping can be either statically predefined before program start or triggered at runtime using method body redefinitions based on Java HotSwap (as we did it in [33]). However, the big conceptual drawback compared to JavAdaptor, JVM patches, and customized class loaders is that wrappers never really update (reload) classes but put them in a new context from which several limitations (particularly regarding our criteria) arise.

Flexibility. Wrappers do not provide the same flexibility as JavAdaptor, customized JVMs and customized class loaders do. *Lasagne* [42] and *JAC* [32] only allow anticipated runtime program updates, because the wrappings must be predefined before application start. Nevertheless, wrappers can be also used in an unanticipated way, as we demonstrated in prior work[33]. The big issue is that conceptually wrappers cannot remove fields or methods defined in classes they wrap.

Platform Independency. The wrapper approach is a well-known design pattern [12], which is fully implemented at application level and thus does not require specific platforms to act. However, to enlarge its flexibility it must be combined with techniques which allow to (re-)define wrappings at runtime.

Performance. There is one point with wrappers that cause significant performance penalties: indirections due to object wrappings. In [14], we measured the performance penalties caused by long wrapping chains, which raise by up to 50% compared to the same program without wrappers.

Program Architecture. The principle drawback of wrappers is that an application must be completely refactored in order to prepare it for wrapper based dynamic software updates. If the developer aims at avoiding poor performing reflective field accesses, she has to allow read and write access to all fields of the old program part namely the object to be wrapped. Furthermore, all classes have to be forced to implement unique interfaces. In addition, all fields have to be of the type of the interface their classes implement. That is, similar to customized class loaders, the wrapper approach dictates the design of an application and, thus, restricts user-defined application designs. In addition, the forced design has serious drawbacks because it violates encapsulation and causes the self-problem [24]. Another problem with the design of several wrapper approaches is decreased reliability due to frequent type casts.

6.4 JavAdaptor

So far, all considered approaches have their strengths and weaknesses regarding the given criteria, i.e., no approach fulfills them all. But, as we described above, their is a need for approaches that cover all criteria. In the following we compare JavAdaptor with the previously described approaches and discuss whether

	DSU Approach	Flexibility	Platform Independency	Performance	Appl. Architecture
	JavAdaptor	●	●	●	●
JVM	*Jvolve*	●	○	●	●
JVM	*HotSpot*	●	○	●	●
JVM	*JDrums*	●	○	○	●
JVM	*DVM*	●	○	○	●
CCL	*Javeleon*	●	●	◑	○
CCL	*DUT*	●	●	◑	○
CCL	*FastSwap*	◑	●	◑	○
CCL	*OSGi*	○	●	◑	○
Wrapper	*[33]*	◑	●	○	○
Wrapper	*JAC*	○	●	○	○
Wrapper	*Lasagne*	○	●	○	○

Table 5. Overview comparison.

JavAdaptor fulfills all criteria or not. An overview of the comparison results can be found in Table 5.

Flexibility. As demonstrated in Section 5, the flexibility of our runtime update approach JavAdaptor is as good as the flexibility that could be achieved by patched JVMs and customized class loaders. More precisely, it is on a par with *Jvolve*, *JDrums*, *DVM*, the HotSpot VM patch of Dmitriev and Würthinger [9][46], *Javeleon*, and *DUT*.

Platform Independency. When it comes to platform independence, JavAdaptor clearly outperforms many competitors. Without any JVM patches it runs on top of all standard JVMs that provide Java HotSwap, which amongst others is a standard feature in the HotSpot VM, the JRockit VM, and IBM's JVM. Furthermore, it does not require any library or framework to act.

Performance. Another strength of JavAdaptor is its performance. As our benchmark results in Section 2 show, container based updates come along without performance penalties and proxy based updates only cause slight performance drops. JavAdaptor neither requires performance-dropping JVM patches nor reflection-based version-barrier crossings (which may be slow particularly on older JVMs) caused by customized class loaders. It also does not depend on a component framework, such as *Javeleon*, *FastSwap* or *OSGi*, whose execution causes additional performance overhead. Furthermore, JavAdaptor causes no wrapping chains and thus comes without the related performance issues.

Application Architecture. Unlike customized JVMs, JavAdaptor requires to add a container field to each class. However, the container field is transparent to the user and can be easily integrated with the application to be updated without any changes to the architecture. By contrast, customized class loaders particularly in conjunction with component frameworks dictate the application design and, thus, render alternative (tailor-made) application designs impossible.

This is also true for wrappers where the forced application design additionally causes serious drawbacks.

7 Open Issues

In Section 4 we described the basic concepts of our DSU approach and evaluated its practicability under real world conditions in Section 5. Even if the results of our evaluation confirm the practicability and potential of JavAdaptor, there is still space for improvements. In this section we summarize work in progress to improve JavAdaptor.

Fig. 10. Active method problem.

Active methods. An issue the JavAdaptor implementation described in this paper not yet addresses is the handling of methods active on the stack scheduled for an update. Figure 10 illustrates the problem of updating active methods. Supposed we update our small weather station program (i.e., reload callee class `TempSensor` and update caller class `TempDisplay`) at a point in time where method `displayTemp` of class `TempDisplay` is active on the stack, method `displayTemp` will continue to execute the old code. Thus, it will still call the

outdated `TempSensor` and its instances (such as depicted down to the right of Figure 10) until it is popped from the stack.[11] The problem is, that after the state mapping between the outdated and updated instances of class `TempSensor`, method `displayTemp` may continue to alter the state of an outdated instance of type `TempSensor` and not of the corresponding up-to-date instance which results in state losses on the updated callee side.

One solution for this problem would be to postpone the update until no method to be updated 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 until it comes true. In order to apply updates immediately, even in the presence of active methods, we have to find another strategy. Fowler argues in [10] that, compared to direct field accesses, getter and setter methods allow us to flexibly manage field accesses. Figure 11 shows how getter and setter methods can be used to face the active method problem described above. The getters and setters act as indirection layers between caller and field. If we now apply the update, we can redefine the getter (see Lines 48 – 51) and setter (Lines 53 – 55) methods in such a way that they redirect field accesses from within active methods to the corresponding up-to-date instances of the reloaded class (here of class `TempSensor_v2` stored in the container.

In order to be able to update every class and to handle related active methods, those getter and setter methods have to be created for all class and instance fields of all classes including the system classes of Java. However, before we started to experiment with this solution we thought that the system wide usage of getter and setter methods would probably cause significant performance penalties which of course would be at odds with our claims. But, contrary to expectations, first benchmark results show that this is virtually not the case because of the excellent optimization capabilities of the JVM and its just-in-time compiler. In addition, other DSU approaches such as Kim's proxy based DSU approach [21] and Javeleon [15], which use system wide getter and setter methods for similar purposes as we intend to do, show that those kinds of indirections must not cause significant performance drops.

Reflection support. However, we not only focus on support for dynamic software updates in the presence of active methods. 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. Currently, we are figuring out the applicability of two different approaches to solve this issue. One

[11] We could of course pop those methods from the stack by hand, which is supported by the JVM. But, the JVM would not allow us to push the updated method back on the stack.

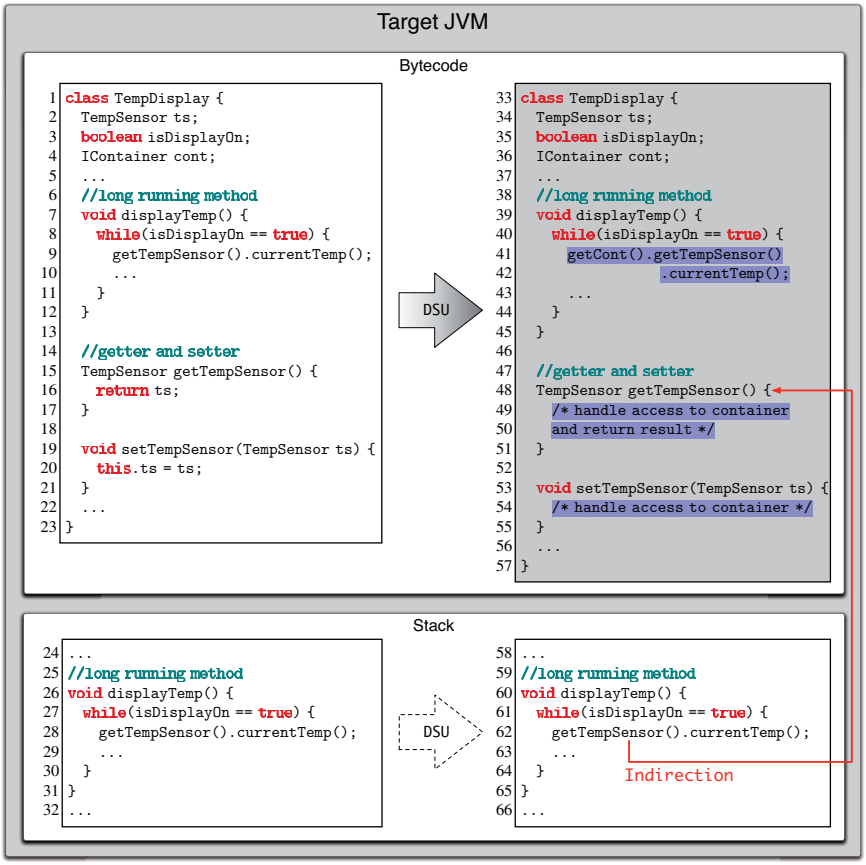

Fig. 11. Active method handling.

solution for this problem is to detect all existing reflective objects that refer to the old class and replace them with corresponding objects referring the latest class version. Another strategy would again deploy getter and setter methods to ensure that each reflective call will be redirected to the latest class version.

Update speed. In Section 5.3 we evaluated to what degree JavAdaptor cuts down the times of service unavailability of our example program HSQLDB compared to updates through program restarts. We found that JavAdaptor outperforms the restart strategy (based on program reinitialization through PolePosition) in terms of service availability. Nevertheless, we also observed that method `getReferringObjects` (to identify all callers of the objects to be updated) may inadequately increase the time required to update a program under certain conditions. One solution for this problem might be to not pause the

application and update the caller site 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. Once again, getter and setter methods can help us out. Using Java HotSwap we could redefine them in such way that they do not only redirect field accesses but also update the related objects on demand. This solution not only renders the usage of method `getReferringObjects` unnecessary but avoids time periods of unavailability during the update at all.

Consistency. However, so far we discussed solutions for issues already solved by other DSU approaches such as Kim's proxy based DSU approach [21] and Javeleon [15]. What remains an open question to the research community is, how to fully ensure program consistency beyond dynamic updates. Gupta et al. state in [16] that the consistency problem is undecidable. Nevertheless, a lot of related work exists facing the problem (see [44][22][37][18][28][2][20][26]). But, to our best knowledge, all approaches either provide approximated solutions only or are not applicable in real world scenarios (e.g., due to the lack of tool support, etc.). Thus, our big goal with JavAdaptor is to ensure consistency while bridging the gap between theory and practice.

All in all, we are optimistic to solve the issues with active methods, reflective calls, and the update speed, soon. Furthermore, preliminary results of experiments with prototypes suggest that the issues can be solved without compromising the contributions of JavAdaptor claimed in this paper, i.e., its flexibility, performance, architecture independence, and platform independence. 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 [21] and [15]) facing similar problems. However, what is still missing and what we intend to provide with JavAdaptor in the long run is a DSU approach which on the one side is useful in practice and on the other side ensures program consistency.

8 Conclusion

Dynamic software updates are a often requested approach to update applications while improving the user experience and prevent down times. Furthermore, DSU supports the software developers because they do not need to restart their applications to test the changed program parts.

However, different from dynamic languages, native DSU support for Java is severely limited. Thus, approaches are needed that overcome Java's limitations regarding dynamic software updates. In Section 1 and 6, we argue that a DSU approach should provide *flexible* runtime program updates without serious *performance* drops. Additionally, it should be *platform independent* and should not dictate the *program architecture*. With JavAdaptor, we overcome Java's limited

runtime update support and add the runtime update capabilities known from dynamic languages to Java. Furthermore, JavAdaptor is (to our best knowledge) the first approach that fulfills all proposed quality criteria: it is flexible, runs on every major (unmodified) JVM, performs well, and does not dictate the architecture of the program. Conceptually, it combines schema changing class replacements with class renaming and caller updates based on Java HotSwap with the help of containers and proxies.

With a case study, we have demonstrated that JavAdaptor fits runtime updates of real world applications executed under real world conditions. Nevertheless, there is still space for improvements. Currently we are working on the integration of the improvements to JavAdaptor described in Section 7, which tackle some issues of the current JavAdaptor implementation. However, in the long run, we will focus on the development of solutions to be integrated into JavAdaptor that fully ensure the program consistency in the presence of runtime updates, which is still not possible with any existing DSU approach applicable in practice.

9 Acknowledgements

We would like to thank Shigeru Chiba for providing the invaluable bytecode modification tool Javassist. Furthermore, we thank Janet Feigenspan for calculating the statistical significance of our benchmark results. Mario Pukall's work is part of the RAMSES project[12] which is funded by DFG (Project SA 465/31-2). Kästner's work is supported in part by the European Union (ERC grant ScalPL #203099).

References

1. The OSGi Alliance. OSGi Service Platform Core Specification, 2009. http://www.osgi.org/Download/File?url=/download/r4v42/r4.core.pdf.
2. Rida A. Bazzi, Kristis Makris, Peyman Nayeri, and Jun Shen. Dynamic Software Updates: the State Mapping Problem. In *HotSWUp '09: Proceedings of the Second International Workshop on Hot Topics in Software Upgrades*, pages 1–2, New York, NY, USA, 2009. ACM.
3. J. Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 1–2, 2004.
4. G. Bracha. Objects as Software Services, 2005. Invited talk at the International Conference on Object-Oriented Programming, Systems, Languages, and Applications.
5. Walter Cazzola. SmartReflection: Efficient Introspection in Java. *Journal of Object Technology*, 3(11):117–132, 2004.
6. 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, 2003.

[12] http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/ramses/index.htm

7. Shigeru Chiba. Load-Time Structural Reflection in Java. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 313–336, 2000.
8. Markus Dahm. Byte Code Engineering. In *Java-Informations-Tage*, pages 1 – 11. Springer-Verlag, 1999.
9. M. Dmitriev. *Safe Class and Data Evolution in Large and Long-Lived Java Applications*. PhD thesis, University of Glasgow, 2001.
10. Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2006.
11. B. Fulgham and I. Gouy. The Computer Language Benchmarks Game, 2010. http://shootout.alioth.debian.org/.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable object-oriented Software*. Addison-Wesley, 1995.
13. J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition)*. Addison-Wesley Professional, 2005.
14. Sebastian Götz and Mario Pukall. On Performance of Delegation in Java. In *Proceedings of the International Workshop on Hot Topics in Software Upgrades*, pages 1–6, 2009.
15. Allan R. Gregersen and Bo N. Jørgensen. Dynamic Update of Java applications - Balancing Change Flexibility vs Programming Transparency. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(2):81–112, 2009.
16. Deepak Gupta, Pankaj Jalote, and Gautam Barua. A Formal Framework for Online Software Version Change. *IEEE Trans. Softw. Eng.*, 22:120–131, 1996.
17. Michael Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. PhD thesis, Software Technology Group, Darmstadt University of Technology, 2006.
18. Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, 2005.
19. J. Kabanov. JRebel Tool Demo. In *Proceedings of the Workshop on Bytecode Semantics*, pages 1–6, 2010.
20. Feras Karablieh and Rida A. Bazzi. Heterogeneous Checkpointing for Multithreaded Applications. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 140–149. IEEE Computer Society, 2002.
21. Dong Kwan Kim. *Applying Dynamic Software Updates to Computationally-Intensive Applications*. PhD thesis, Virginia Polytechnic Institute and State University, 2009.
22. J. Kramer and J. Magee. The evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293 –1306, 1990.
23. 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, 1998.
24. 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, 1986.
25. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification – Second Edition*. Prentice Hall, 1999.
26. Kristis Makris. *Whole-Program Dynamic Software Updating*. PhD thesis, Arizona State University, 2009.
27. Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime Support for type-safe dynamic Java Classes. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 337 – 361, 2000.

28. Yogesh Murarka, Umesh Bellur, and Rushikesh K. Joshi. Safety Analysis for Dynamic Update of Object Oriented Programs. In *APSEC '06: Proceedings of the XIII Asia Pacific Software Engineering Conference*, pages 225–232, Washington, DC, USA, 2006. IEEE Computer Society.
29. A. Nicoara, G. Alonso, and T. Roscoe. Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. In *Proceedings of the EuroSys Conference*, pages 233–246, 2008.
30. Oracle. BEA WebLogic Server Using FastSwap to Minimize Redeployment. Technical report, 2006. http://download.oracle.com/docs/cd/E13222_01/wls/essex/TechPreview/pdf/FastSwap.pdf.
31. 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, 2002.
32. Renaud Pawlak, Laurence Duchien, Gerard Florin, and Lionel 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, 2001.
33. 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, 2008.
34. Mario Pukall, Alexander Grebhahn, Reimar Schröter, Christian Kästner, Walter Cazzola, and Sebastian Götz. JavAdaptor: Unrestricted Dynamic Software Updates for Java. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1 – 3, 2011. to appear.
35. Tobias Ritzau and Jesper Andersson. Dynamic deployment of java applications. In *Java for Embedded Systems Workshop*, pages 1–9, 2000.
36. Y. Sato, S. Chiba, and M. Tatsubori. A Selective, Just-in-Time Aspect Weaver. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 189 – 208, 2003.
37. Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. *Mutatis Mutandis*: Safe and Flexible Dynamic Software Updating. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, pages 183–194, January 2005.
38. Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic Software Updates: A VM-Centric Approach. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 1–12, June 2009.
39. Sun Microsystems. Java Virtual Machine Tool Interface Version 1.1. Technical report, 2006. http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html.
40. Sun Microsystems. Java Platform Debugger Architecture. Technical report, 2010. http://java.sun.com/javase/6/docs/technotes/guides/jpda/index.html.
41. Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: spatial and temporal selection of reification. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 27–46, 2003.
42. 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, 2001.
43. W. Vanderperren and D. Suvee. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In *Proceedings of the AOSD Workshop on Dynamic Aspects*, pages 120–134, 2004.

44. 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.
45. B. Venners. *Inside the Java 2 Virtual Machine*. Computing McGraw-Hill., 2000.
46. 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, 2010.
47. Shi Zhang and LinPeng Huang. Type-Safe Dynamic Update Transaction. In *Proceedings of the Computer Software and Applications Conference*, pages 335–340, 2007.