Technical Report

Nr.: FIN-014-2009

Flexible Runtime Program Adaptations in Java - A Comparison

M. Pukall, C. Kästner, S. Götz, W. Cazzola, G. Saake

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-014-2009

Flexible Runtime Program Adaptations in Java - A Comparison

M. Pukall, C. Kästner, S. Götz, W. Cazzola, G. Saake

Arbeitsgruppe Datenbanken

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: pukall@ovgu.de

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

Auflage: 52

Redaktionsschluss: 27.10.2009

Herstellung: Dezernat Allgemeine Angelegenheiten,
Sachgebiet Reproduktion

Bezug: Universitätsbibliothek/Hochschulschriften- und
Tauschstelle

Flexible Runtime Program Adaptations in Java - A Comparison

Mario Pukall, Christian Kästner
University of Magdeburg
{pukall, ckaestne}@ovgu.de

Sebastian Götz
TU Dresden
sebastian.goetz@jexam.de

Walter Cazzola
University of Milano
cazzola@dico.unimi.it

Gunter Saake
University of Magdeburg
saake@ovgu.de

Abstract

Software development is an ongoing process which does not end when the first version of an application is released. Bugs must be fixed and requirements evolve. Maintaining an application usually means to stop the application, apply the required changes, and start the application again. This downtime is not acceptable for applications that must be available 24 hours a day, 7 days a week. On the other hand even for end-user desktop applications, restarts to apply patches can be an annoying user experience. For that reason we investigate how to maintain applications at runtime. However, due to the fact that it is not predictable what changes become necessary and when they have to be applied the application must be enabled for unanticipated changes even of already executed program parts. In previous work we proposed a solution for Java, since Java is commonly used for developing 24/7 applications. Unfortunately, this solution came with some limitations. Therefore, we present a novel runtime maintenance approach based on class replacements and mediators which overcomes these limitations and allows unanticipated changes of applications that run on top of a standard Java virtual machine.

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. Erlikh [1] and Moad [2] estimate the costs to maintain a program at 90 percent of the overall development costs. Time periods of unavailability additionally increase the maintenance expenses. This is particularly true for applications that must be available 24 hours a day, 7 days a week. On the other hand even for end-user desktop applications, restarts to apply patches can be an annoying user experience [3]. Users do not prefer maintenance approaches that require to interrupt user tasks. For that reasons we aim at approaches that allow to maintain applications at runtime.

Even though dynamic languages like Smalltalk, Python or Ruby natively support runtime program changes, we choose Java as language for several reasons. First, Java is a programming language commonly used to implement highly available applications. Examples are *Apache Tomcat*[1], *Java DB*[2], or *JBoss Application Server*[3]. Second, in most fields of application Java programs execute faster than programs based on dynamic languages [4]. Amongst others, this is due to the fact that Java is a (pre-)compiled language. Unfortunately, compilation prevents Java and languages like C and C++ from natively offering powerful instruments for runtime program adaptation.

Literature suggests a wide range of runtime program adaptation approaches (see related work in Section 8). The usability of an approach can be determined by answering the following two questions: (a) are unanticipated changes allowed (i.e., the application of requirements for what the running program was not prepared), and (b) can already executed program parts be changed? We believe that it is impossible to prepare an application for all upcoming requirements. Furthermore, only offering modifications of not previously executed program parts (e.g., not yet loaded classes) while disregarding the executed parts (e.g., already loaded classes) restricts the application of program changes. For that reason, our approach aims at unanticipated runtime program changes that also affect executed program parts.

Researchers spent a lot of time to overcome Java's shortcomings regarding runtime program adaptation. Approaches like *Javassist* [5], [6] and *BCEL* [7] allow to apply some unanticipated changes, but only to not yet executed program parts. In contrast *PROSE* [8], *DUSC* [9], *AspectWerkz* [10], *Wool* [11], or *JAsCo* [12] allow unanticipated changes even of executed program parts. However, PROSE, AspectWerkz, Wool, and JAsCo do not enable class schema affecting runtime adaptations. Although DUSC allows class schema changes the program loses its state. In prior work, we developed an approach based on Java HotSwap and object wrapping that overcomes the limitations of the referred approaches while supporting a wide range of program changes [13]. Unfortunately, the previous approach comes with some drawbacks because we applied certain workarounds to enable unanticipated changes. This reduces the approachs benefit and applicability as we will unveil. Amongst others, it violates *encapsulation*, introduces the *self-problem*, and

1. http://tomcat.apache.org/index.html
2. http://developers.sun.com/javadb/
3. http://www.jboss.org/jbossas/

	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 [15].

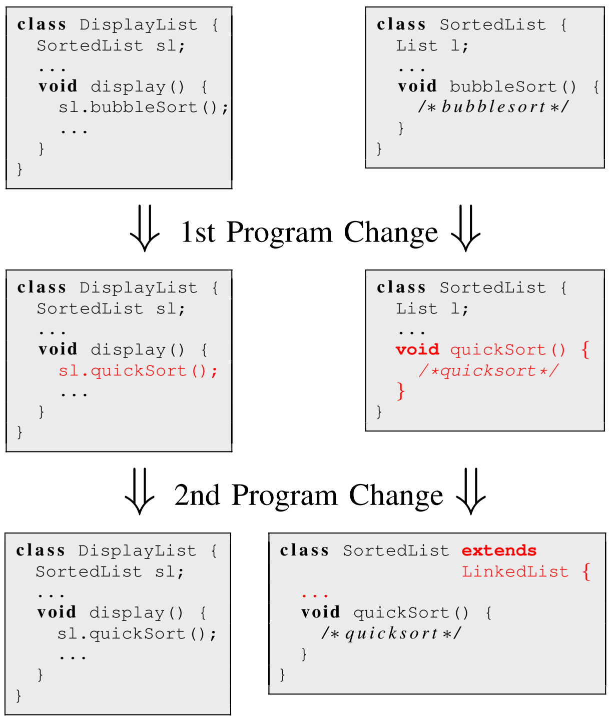
decreases the programs reliability due to frequent type casts.

This paper presents a completely reworked and enhanced version of our previous runtime adaptation approach. It enables Java applications for unanticipated changes at runtime even of already executed program parts – as only known from dynamic languages. We review the shortcomings of our previous approach and explain how we overcome them by combining class replacement techniques and mediators with functions provided by the *Java Virtual Machine Tool Interface* (as part of *Java's Platform Debugger Architecture* [14]) which allow to inspect and to control an application running on top of Sun's HotSpot VM.

2. Motivating Example

Program maintenance is not a trivial task that 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.

Figure 1 examplifies that even simple program changes can affect many parts of a program. The program depicted in Figure 1 consists of 2 classes. One class (`SortedList`) stores and sorts lists while the other class (`DisplayList`) is responsible for displaying them. Considering a maintenance task, the actual sorting algorithm (*Bubblesort*) must be replaced by a faster one (*e.g., Quicksort*). For some reason (e.g. long start up times because cashes have to be filled, no backup available, better user experience, etc.) 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 `bubbleSort()` of class `SortedList` must be replaced by method `quickSort()` which implements the *Quicksort* algorithm. Second, in order to execute the Quicksort algorithm method `display()` of class `DisplayList` must be reimplemented. Short time after applying the QuickSort algorithm it was also decided to let `SortedList` inherit from class `LinkedList` in order to add new functions to `SortedList` while avoiding to implement them again. Therefore, statement `extends LinkedList` has to be applied to class `SortedList`. Additionally, member `l` of original class `SortedList` has to be removed because superclass `LinkedList` let it become useless.


```
class DisplayList {
  SortedList sl;
  ...
  void display() {
    sl.bubbleSort();
    ...
  }
}
```
```
class SortedList {
  List l;
  ...
  void bubbleSort() {
    /*bubblesort*/
  }
}
```

```
class DisplayList {
  SortedList sl;
  ...
  void display() {
    sl.quickSort();
    ...
  }
}
```
```
class SortedList {
  List l;
  ...
  void quickSort() {
    /*quicksort*/
  }
}
```

```
class DisplayList {
  SortedList sl;
  ...
  void display() {
    sl.quickSort();
    ...
  }
}
```
```
class SortedList extends
                 LinkedList {
  ...
  void quickSort() {
    /*quicksort*/
  }
}
```

Figure 1. Unanticipated Runtime Adaptation.

Even if the required program changes seem to be simple they affect wide 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 in an unanticipated way.

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 internals of Java's runtime environment – the Java virtual machine (JVM). A Java program is stored in the *heap* and in the *method area* of the JVM (as well as on the stack). Within the heap the runtime data of all class instances are stored [16]. 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).

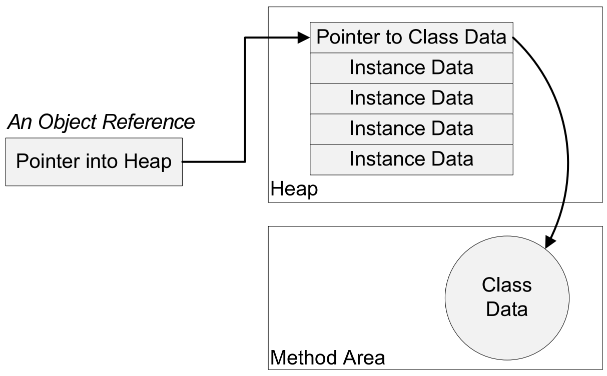


Figure 2. Program Representation in Java HotSpot [17].

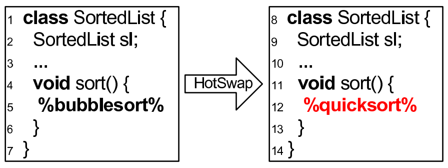
Changing a program during its execution in the JVM requires to modify the data within the heap and the method area. For instance, program changes such as depicted in Figure 1 which also include method replacements require to change the data of a class. In general, they require to modify the class schema. Unfortunately, the JVM does not permit class schema changes, because class schema changes may let the data on the heap and the class data stored in the method area become inconsistent while the JVM does not provide functions to synchronize them.

Java HotSwap. Despite the insufficient native runtime adaptation support of the JVM there is a feature provided by the Java HotSpot VM – called *Java HotSwap* [18], [19]. It allows to replace the body of a method (which partly covers points 4 - 6 of Table 1) while the program is running. The virtual machine permits this operation because it does not cause inconsistencies.

The class data restructuring via Java HotSwap consists of the following steps: First, an updated version of the class to be changed 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 as well as every other function 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. Object Wrapping Approach

Having described why solely using Java HotSwap for runtime adaptation concerns is not sufficient we present two approaches we developed to overcome the mentioned limitations. The first solution based on object wrappings and interfaces was presented in [13]. It allows many runtime program changes but has some drawbacks which restrict its application in real world programs. Before we present our reworked and improved runtime adaptation approach, we revisit the previous solution and discuss the shortcomings which forces us to develop the new approach.

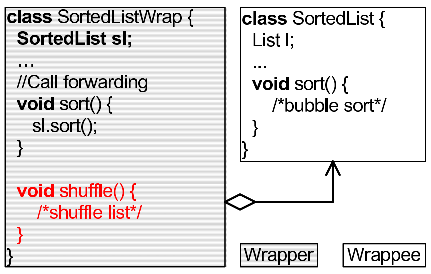


```
class SortedList {
  SortedList sl;
  ...
  void sort() {
   %bubblesort%
  }
}
```

```
class SortedList {
  SortedList sl;
  ...
  void sort() {
   %quicksort%
  }
}
```

Figure 3. Class Schema Keeping Updates.

Schema Preserving Runtime Adaptations. One reason for changing a running program is to replace method bodies which partly covers points 4 - 6 of Table 1. Such an example is depicted in Figure 3, where the actual sorting algorithm (Bubblesort) of class `SortedList` is replaced by the faster Quicksort algorithm. Such replacements only require to reimplement the body of a method (here of method `sort()`) which does not affect the class schema. We achieve such method body reimplementations using the standard functionality of Java HotSwap as described above.



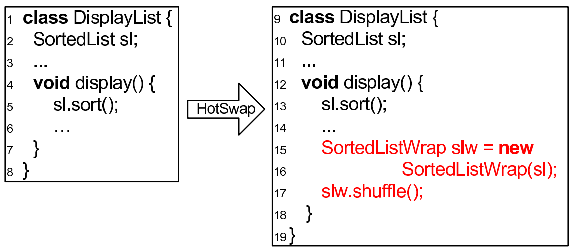
```
class SortedListWrap {
  SortedList sl;
  ...
  //Call forwarding
  void sort() {
    sl.sort();
  }

  void shuffle() {
    /*shuffle list*/
  }
}
```

```
class SortedList {
  List l;
  ...
  void sort() {
    /*bubble sort*/
  }
}
```

Figure 4. Class Schema Affecting Updates.

Schema Altering Runtime Adaptations. In many cases new requirements imply more complex program changes then provided by Java HotSwap (i.e., all points of Table 1). This is exemplary shown in Figure 4, where new requirements make it necessary to add new methods (here `shuffle()`) to the program. We use object wrapping to apply new elements to the program. As depicted in Figure 4 wrapper `SortedListWrap` applies the new functionality (method `shuffle()`) to the program whereas all other methods are delegated to the original class to preserve the original behavior.



```
1  class DisplayList {
2    SortedList sl;
3    ...
4    void display() {
5      sl.sort();
6      ...
7    }
8  }
```

```
9  class DisplayList {
10   SortedList sl;
11   ...
12   void display() {
13     sl.sort();
14     ...
15     SortedListWrap slw = new
16                 SortedListWrap(sl);
17     slw.shuffle();
18   }
19 }
```

Figure 5. Caller Update.

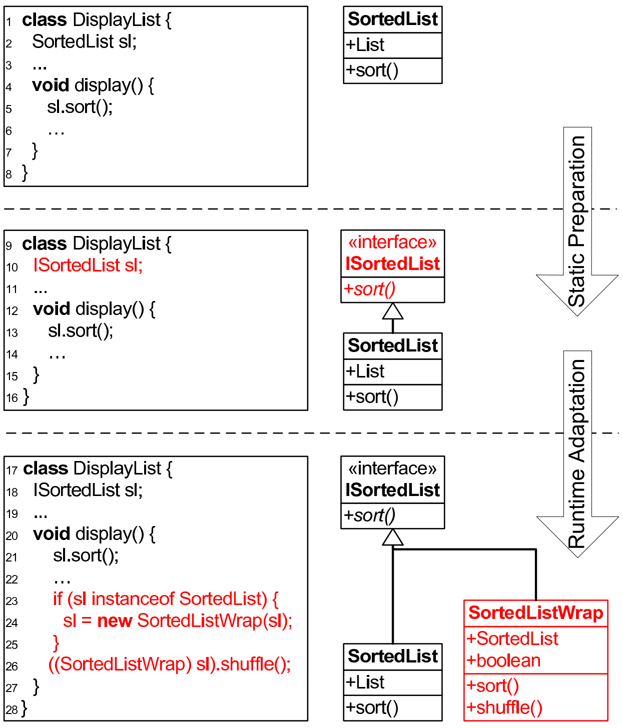
To apply and invoke the new or modified functionality provided by the wrapper all instances of the changed class have to be updated. To use additional functions caller method `display()` has to be reimplemented using Java HotSwap (see Figure 5). The reimplementation consists of two parts. First, an instance of wrapper `SortedListWrap` is created (line 15-16). Second, the new functionality is called (line 17).

We note that the mechanism described above only allows local wrappings. In order to also allow global wrappings we combine our approach with interfaces (see Figure 6). The static type of callee `sl` of class `DisplayList` is changed (before program start) to interface type `ISortedList` which enables `sl` for late binding. Thus, it is possible to assign objects different from type `SortedList` to `sl` during runtime. The application of program changes itself also bases on method reimplementations using Java HotSwap and works as depicted down to the left of Figure 6. First, an instance of class SortedListWrap is created (line 24). It takes the original value of callee `sl` (an instance of `SortedList`) as input. Second, the wrapper instance is assigned to callee `sl`, i.e., the runtime type of `sl` switches from `SortedList` to `SortedListWrap` (line 24). To call methods not declared in interface `ISortedList` `sl` must be casted to the actual runtime type (line 26).

4.1. Shortcomings

In [13], we demonstrated the practicability of our previous runtime adaptation approach through a case study. Thereby, we observed different problems which result from object wrappings and interfaces. Some of the problems we solved through workarounds (which required boilerplate code) while others remained open issues. Unfortunately, the workarounds and open problems reduce the applicability of our runtime adaptation approach.

Problems – Object Wrapping. Wrapping an object means to combine new and old program parts. This requires to allow read and write access to all fields of the old



```
1  class DisplayList {
2    SortedList sl;
3    ...
4    void display() {
5      sl.sort();
6      ...
7    }
8  }
```

```
9  class DisplayList {
10   ISortedList sl;
11   ...
12   void display() {
13     sl.sort();
14     ...
15   }
16 }
```

```
17 class DisplayList {
18   ISortedList sl;
19   ...
20   void display() {
21     sl.sort();
22     ...
23     if (sl instanceof SortedList) {
24       sl = new SortedListWrap(sl);
25     }
26     ((SortedListWrap) sl).shuffle();
27   }
28 }
```

Figure 6. Global Wrappings.

program part namely the object to be wrapped. To meet the requirements all fields have to be public. In other words, the object wrapping approach on the one hand requires to violate encapsulation which bypasses the scoping concept and security guarantees of Java and on the other hand necessitates to expensively change the field modifiers (which must be done manually in our previous runtime adaptation approach).

Next problem resulting from object wrappings is the *self-problem*. It describes situations which require to call the wrapper from within the wrappee, i.e., situations which require delegation [20]. Since Java does not support delegation object wrappings fail in such situations.

Another challenge of object wrappings is function removement. In terms of highly available applications it is probable that the application has to be changed more than once. In this regard it might be the case that already wrapped objects have to be further enhanced by introducing additional wrappings. As denoted in Figure 7 this results in a wrapping construct with strong dependencies inside, i.e., each wrapper expects its wrappee to be of specific type. Due to the dependencies our previous approach is unable to fit program changes which necessitate to remove functionality introduced by a wrapper from within the middle of the wrapping chain.

Additionally, complex wrappings such as depicted in Figure 7 cause performance penalties [21]. This is because

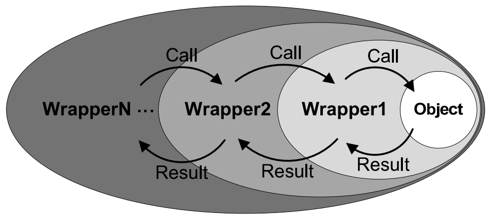


Figure 7. Wrapping Chain.

of the indirections they introduce.

Problems – Interface Mechanism. As previously described we achieve updates of callee instances (also of global ones) via late binding respectively interfaces. Even if this approach lets the underlying application become widely modifiable it comes along with major shortcomings.

In order to prepare an application for the interface mechanism it has to be changed extensively. First, all classes have to be forced to implement unique interfaces. Second, all class fields have to be of type of the interface its classes implement. Third, all final class fields must be changed to be modifiable. Since, all changes have to be done manually in our previous runtime adaptation approach preparation takes much effort and is error-prone.

Next problem is program reliability. Due to the fact that interfaces have a fixed set of public methods they declare, they cannot be used to invoke protected methods as well as methods introduced by the wrapper. Invocation of such methods requires to cast the relevant object into the wrappers type which is unsafe and reduces performance. Additionally, the cast must be done for each such call within the program which (especially in terms of program wide used methods) dramatically decreases the programs reliability.

Just like object wrappings interfaces also complicate function removements. Whenever a new wrapper must be created it has to implement the complete set of interface methods, i.e., it is impossible to remove a method from the program that is part of interface definition.

Challenges	Object Wrapping	Interface
Encapsulation Violation	yes	no
Self-Problem	yes	no
Reliability Problem	no	yes
Function Removement	yes	yes
Performance Penalties	yes	yes
Preparation Effort	yes	yes

Table 2. Summary – Shortcomings.

5. Runtime Adaptation via Class Replacements and Mediators

The lesson we have learned from our case study presented in [13] is that our runtime adaptation approach based on object wrappings and interfaces can be applied to real world applications. Nevertheless, due to the constraints described in Section 4, the application of program changes is not straightforward. It has open issues and requires lots of workarounds (see Table 2). For that reason, we developed a completely reworked runtime adaptation approach which comes without the described limitations. It still uses Java HotSwap, but class replacements instead of wrappers and mediator on caller side instead of interfaces.

5.1. Runtime Adaptation Architecture

As we described in Section 3 the JVM does not support class updates that change the class schema. Therefore, we developed our own runtime adaptation tool which combines class replacements and mediators to achieve flexible program updates also including class schema changes. It comes as a plug-in which smoothly integrates into the *Eclipse IDE*[4].

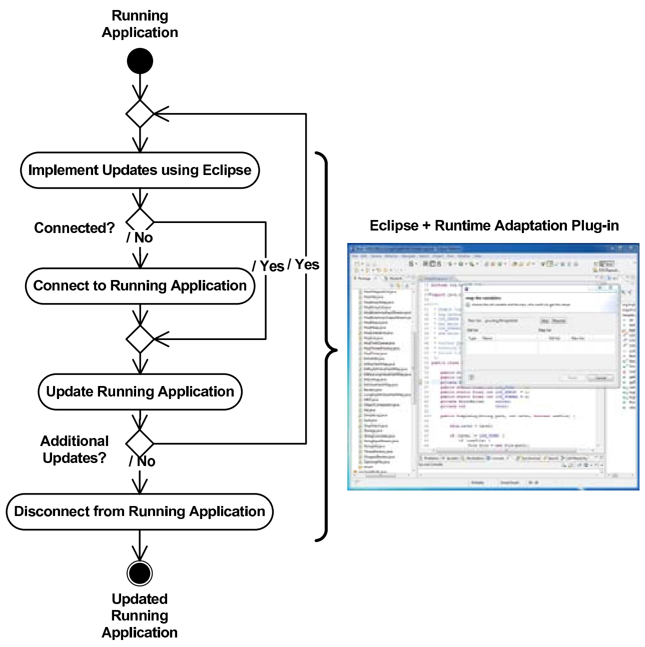

Running Application
Implement Updates using Eclipse
Connected?
/ No
/ Yes
/ Yes
Connect to Running Application
Update Running Application
Additional Updates?
/ No
Disconnect from Running Application
Updated Running Application
Eclipse + Runtime Adaptation Plug-in

Figure 8. Update Process.

Figure 8 describes the usage of our tool from the developers point of view. The implementation of the required program updates is conform to the usual static software development process, i.e., the developer implements the required functions using the eclipse IDE and compiles the sources. This results in sound bytecode because of the static type checking done by the compiler. When the developer decides to update the running application, our tool establishes a connection to the JVM executing the application (see Figure 9). In more detail it connects to the

4. http://www.eclipse.org/

JVM's *Java Virtual Machine Tool Interface* (JVMTI) which is used to control the JVM [22] (accessible from outside the JVM through the *Java Debug Interface* (JDI)). Once the connection is successfully established our runtime adaptation tool prepares the bytecode so that it can be applied to the running application. After the update our tool disconnects from the application. The described process can be repeated as often as required.

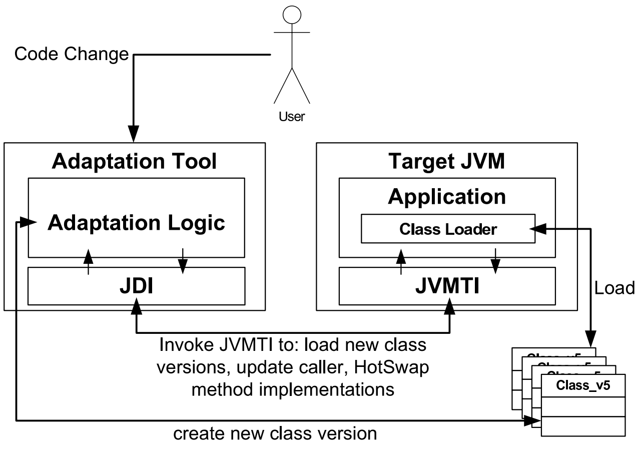


Figure 9. Runtime Adaptation Architecture.

In the following we describe the basic mechanisms how our adaptation tool changes applications running in the target JVM, namely class replacements and mediators.

5.2. Class Replacement

The only way to change the schema of a loaded class is to replace the old class version by a new one and update all callers of the outdated class.

Unfortunately, replacing a class is a challenging task which requires deep knowledge about the class loading concept Java enforces. The concept is based on hierarchically ordered class loaders. 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 key problem that makes class replacements difficult is that it is not possible to unload a class until its class loader is dereferenced. To dereference a class loader all its classes (even the unchanged ones) have to be dereferenced which in case of Java's basic class loaders is equivalent to an application stop.

Beside all restrictions, we found two ways to enable runtime adaptations based on class replacements. Here, we focus on loading a new class version into the running program. How the new class version becomes part of program execution is discussed later.

Customized Class Loader. In addition to the basic class loaders required to load and run a program, the class loading capabilities of a program can be extended by using customized class loaders [23]. This is exemplified in Figure 10 where method `loadClass()` instantiates the customized class loader `MyClassLoader` (line 4) and then loads a class using Java's Reflection API (line 5).

```
1  ...
2  static Class<?> loadClass(String path)
3                                       throws Exception {
4      ClassLoader cl = new MyClassLoader( path );
5      return cl.loadClass("ExampleClass");
6
7  }
```

Figure 10. Target JVM – Customized Class Loader.

Loading a new class version can be done by again calling method `loadClass()`. To reload a class in an unanticipated way the code shown in Figure 10 must be added to the program before start. Additionally, the program must run inside our runtime adaptation architecture. Before class reloading the adaptation tool creates the new class version and triggers the target JVM to execute method `loadClass()`.

Class Renaming. As we described above, each time a class has to be reloaded using customized class loaders a new class loader instance must be created. As a result of frequent class updates the application would be polluted by class loader instances that slow down the application and consume memory space. For that reason we consider another class replacement strategy – class renaming. As exemplified in Figure 11, 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. Additionally, in case of the application class loader, instances of the new class version can be created by the `new` operator instead of using Java's reflection API which causes performance penalties. Because of the advantages mentioned above, we use class renaming to load new class versions.

Figure 12 sketches how class loading based on class renaming is implemented in our runtime adaptation architecture. The renamed and updated class is created by our adaptation tool (based on user input). In the next step the adaptation tool retrieves the class loader bounded to the original class from the target JVM (line 7 - 8) and then invokes method `loadClass()` of this class loader to load the updated class (line 10).

```
class ExampleClass { ... }
```

⇓ Replacement ⇓

```
class ExampleClass_v2 { ... }
```

Figure 11. AT – Class Renaming.

```
class ClassUpdateLoader {
  VirtualMachine targetJVM;
  ...
  void reloadClass(String path) {
    ReferenceType oldClass =
            targetJVM.classesByName("ExampleClass");
    ClassLoaderReference cl =
                           oldClass.classloader();
    ...
    cl.invokeMethod(t, loadClass, args, options);
  }
}
```

Figure 12. AT – Class Loading.

5.3. Caller Update through Mediators

As demonstrated above our class reloading mechanism allows to load a new version of an already loaded class even if the class schema has changed. However, the mechanism only permits the updated class to be present within the running program but not its execution. To let the class become part of program execution all callers of the original class have to be changed so that they call instances of the new class version. This requires four steps: (1) caller detection, (2) instantiation of the new class version, (3) state mapping, and (4) callee substitution. Different from our previous runtime adaptation approach our adaptation tool automates these steps as far as possible. Additionally, we now use the mediator pattern [21] to substitute the callees which overcomes the shortcomings of the previously used interface mechanism as we will show.

```
1   class ObjectReferenceDetector {
2     VirtualMachine targetJVM;
3     ...
4     List<ClassObjectReference> detectCallers() {
5       ReferenceType refL =
6               targetJVM.classesByName("ExampleClass");
7       List<ObjectReference> oRefL = refL.instances();
8       ...
9       oRefL.get(i).referringObjects(l);
10      ...
11    }
12  }
```

Figure 13. AT – Caller Detection.

Caller Detection. In order to replace all instances of the original class by instances of the new class version we have to detect all caller of the original class. The JVMTI implementation of Sun's HotSpot VM supports this operation. A snipped of the corresponding code executed by our adaptation tool is depicted in Figure 13. First, the class object of the old class is retrieved from the target JVM (lines 5 - 6). This object is used to get all instances of the old class (line 7) via reflection. Again, using the instances all callers are retrieved (line 9).

Class Update Instantiation. In the next program adaptation step, for each instance of the original class an instance of the updated class has to be created. The created instances will be used later on to replace the instances of the old class and thus to update the callers.

```
1  class UpdateInstantiation {
2    ...
3    void createInstance(ClassObjectReference cRef) {
4      cRef.invokeMethod(t, constructor, args, options);
5    }
6  }
```

Figure 14. AT – Class Instantiation.

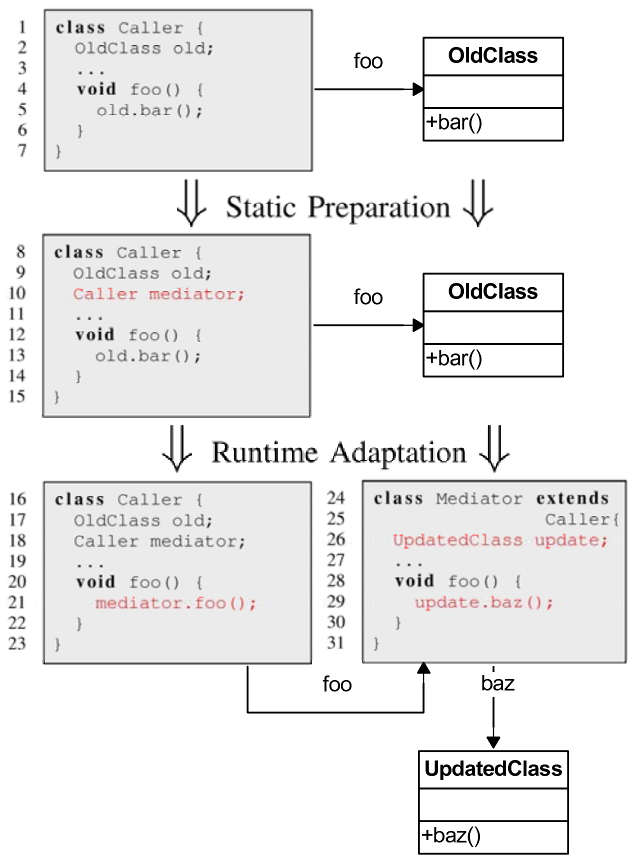
The instantiation is triggered by our adaptation tool. The corresponding code is depicted in Figure 14. Method `createInstance()` takes as argument a class object of the new class version which is used to invoke the default constructor of the new class version within the target JVM (line 4).

```
class StateMapper {
  ...
  void mapState(ObjectReference oldObj,
                        ObjectReference newObj) {
    ...
    newObj.setValue(newField,
                       oldObj.getValue(oldField));
    ...
  }
}
```

Figure 15. AT – State Mapping.

State Mapping. Having finished the instantiation step the state has to be mapped pairwise from old to new instance. The mapping is processed by our adaptation tool. Due to the simplicity of one to one mappings (mappings of values from fields that exist in both class versions) our adaptation tool executes them automatically. We also support 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. However, the mapping function must be manually defined by the user. The function is then used by our adaptation tool to automatically map the state pairwise from old to new instances. Figure 15 sketches how our adaptation tool implements one to one mappings.

Callee Substitution. Once for each instance of the original class an instance of the new class version has been created and initialized with the state of its outdated counterpart all caller have to be updated. That is, all instances



```
class Caller {
  OldClass old;
  ...
  void foo() {
    old.bar();
  }
}
```

```
class Caller {
  OldClass old;
  Caller mediator;
  ...
  void foo() {
    old.bar();
  }
}
```

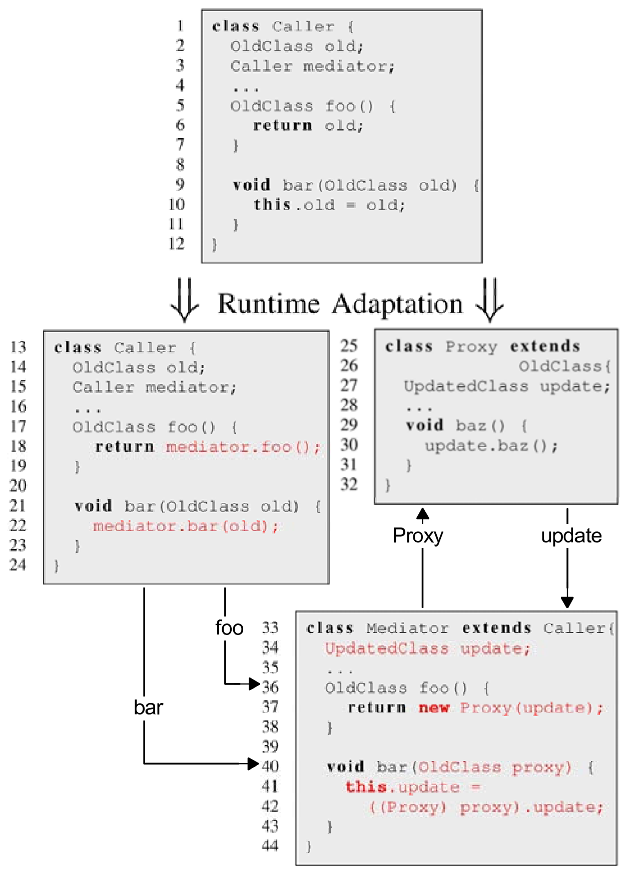
```
class Caller {
  OldClass old;
  Caller mediator;
  ...
  void foo() {
    mediator.foo();
  }
}
```

```
class Mediator extends
                  Caller{
  UpdatedClass update;
  ...
  void foo() {
    update.baz();
  }
}
```

Figure 16. Caller Update through Mediator Pattern [21].

of the original class have to be replaced by the instances of the new class. In contrast to our previous approach we now use mediators to update the callers. Figure 16 exemplifies the usage of mediators. Before program start our adaptation tool prepares the program for the mediator approach, i.e., it adds field `mediator` to each program class. The mediator field does not affect program execution as long as no callee of the caller class has to be replaced. To replace a callee of the caller class the program has to be changed as depicted in the lower part of Figure 16. First, our runtime adaptation tool creates a mediator class and copies all method and field specific code (except of the constructors and the field that must be updated) from the caller class to this class. Second, the tool adds a new field from type of the updated class to the mediator class. Third, our runtime adaptation tool updates all method calls within the mediator according the functionality introduced by the updated class (in Figure 16 the mediator calls method `baz()` instead of method `bar()`). After creating the mediator class the tool assigns (scheduled by the user) an instance of the mediator to field `mediator` within the caller class (which is possible because the mediator class extends the caller class). Last but not least, our tool triggers the target JVM to redefine all public methods of the caller class so that they delegate to their counterparts within the mediator class. This is done using Java HotSwap.

Replacing Return Types and Parameter Types with Proxies. Even if the basic mediator approach described in Figure 16 is sufficient in many cases, it fails when the caller class to be updated contains methods which parameters and returned objects are of type of the old callee class (such as shown in Figure 17, line 6 and 9). One workaround would be to replace the caller class as well. However, 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 runtime adaptation approach become inefficient.



```
1  class Caller {
2    OldClass old;
3    Caller mediator;
4    ...
5    OldClass foo() {
6      return old;
7    }
8
9    void bar(OldClass old) {
10     this.old = old;
11   }
12 }
```

```
13 class Caller {
14   OldClass old;
15   Caller mediator;
16   ...
17   OldClass foo() {
18     return mediator.foo();
19   }
20
21   void bar(OldClass old) {
22     mediator.bar(old);
23   }
24 }
```

```
25 class Proxy extends
26                  OldClass{
27   UpdatedClass update;
28   ...
29   void baz() {
30     update.baz();
31   }
32 }
```

```
33 class Mediator extends Caller{
34   UpdatedClass update;
35   ...
36   OldClass foo() {
37     return new Proxy(update);
38   }
39
40   void bar(OldClass proxy) {
41     this.update =
42       ((Proxy) proxy).update;
43   }
44 }
```

Figure 17. Extended Mediator.

In order to avoid cascading class replacements, we extend the basic mediator approach with proxies (see Figure 17). Caller updates work in the same manner as described above. Only difference is that in addition to the mediator class a proxy class is generated. The proxy manages accesses to updated callees from outside the caller and ensures return as well as input type compatibility when calling methods of the caller class which require and/or return objects of type of the old callee class. The usage of proxies is exemplified on the basis of method `foo()` of class `Caller` which returns an instance of callee class `OldClass` (line 6). After replacing callee class `OldClass` by class `UpdatedClass`, method `foo()` has to return an instance of class `UpdatedClass` which is not possible because `OldClass` and `UpdatedClass` are not type compatible. To achieve type compatibility we assign the instance of `UpdatedClass` to an instance of class `Proxy` (line 37). Since the proxy extends class `OldClass` and also implements the methods of class `UpdatedClass` it can be returned by method `foo()` and also used by the receiver to operate on the `UpdatedClass` instance. How to propagate instances of the updated class back to the caller (more precisely to the mediator) is exemplarily shown in Figure 17 (line 41-42). In order to access and use the received instance of class `UpdatedClass`, the proxy must be cast to type `Proxy`.

6. Comparison

As described in Section 4.1, our previous runtime adaptation approach, more precisely the mechanisms we used in it (object wrapping and interfaces), has several drawbacks. In this section, we want to answer the question whether class replacements and mediators struggle with the same drawbacks and/or cause new ones.

Encapsulation Violation. In our original runtime adaptation approach we used object wrappings in order to achieve class schema changes. Unfortunately, object wrappings violated encapsulation. Unlike object wrappings, class replacements not only simulate class schema changes, but really change the schema of a class, i.e., modified code and class fields as well as instance fields reside in the same piece of code (i.e, a class). For that reason class replacements allow field modifiers to remain untouched.

Self-Problem. As we described in Section 4.1 object wrappings cause the self-problem. The reason why class replacements do not cause this problem is the same as for avoiding encapsulation violation. It comes from the fact that class replacements provide real class schema changes which make delegation unnecessary.

Reliability Problem. Reliability is one of the most important concerns of software maintenance. The usage of interfaces requires type casts in order to invoke methods not declared within the corresponding interface. Frequent type casts decrease program reliability. Through mediators (that substitute interfaces in our new runtime adaptation approach) an updated object is explicitly applied to the program, i.e., the objects runtime type is the same as its static type. That allows not only direct invocations of newly added methods but also static type checking which improves program reliability.

Function Removement. Removing functions with the previous runtime adaptation approach is difficult for two reasons. On the one hand it is difficult because of the strong dependencies inside wrappings. On the other hand this is because of the interface mechanism which restricts to remove methods that are part of interface definitions. Naturally, class replacements cannot cause dependencies such as object wrappings do (because of the absence of the corresponding inter object relations) and thus do not complicate function removements. Additionally, mediators do not require to determine the set of methods that have to be implemented by a class, i.e., every method can be removed.

Performance Penalties. There were two critical points regarding performance in our previous runtime adaptation approach: additional indirections due to object wrappings and frequent usage of interfaces. Unlike object wrappings class replacements do not cause additional indirections. No matter how many times a class has to be replaced. Mediators do not base on interfaces and thus come without the performance overhead of them. They only introduce one additional indirection and no chains of indirections which would significantly decrease performance.

Preparation Effort. In order to prepare an application for our previous runtime adaptation approach it must be completely refactored. The new runtime adaptation approach only requires to add an additional instance field (placeholder for a mediator) to each class before program start. In other words, we significantly reduced boilerplate code and thus preparation effort.

State Mapping Effort. Beside the advantages of class replacements there is one point of criticism: state mapping effort. Whereas object wrappings do not require one to one mappings (since the state of the wrappee is kept) class replacements do. This is because all instances of the old class are fully replaced by instances of the new class version. Fortunately, our adaptation tool automates one to one mappings which moderates the mapping overhead. However, more complex (indefinite) state mappings must be handled by both approaches.

Automation. Our previous runtime adaptation approach required to process all adaptation steps manually. The adaptation tool we implemented for our new runtime adaptation approach reduces the effort of runtime program changes because it automates the following adaptation steps: creation of new class versions (based on user input), instantiation of new classes, state mappings (one to one), and caller detection. Callee substitutions are also automatically processed but must be triggered by the user.

7. Open Issues

Even if the presented runtime adaptation approach solves almost all problems the old approach came with (see Table

Challenges	Wrapping Interface	Class Replacement Mediator
Encapsulation Violation	yes	no
Self-Problem	yes	no
Reliability Problem	high	decreased
Function Removement	yes	no
Performance Penalties	high	low
Preparation Effort	high	low
State Mapping Effort	low	increased
Automation	no	semi-automatic

Table 3. Comparision.

3), there is still space for improvements. Most important point is to ensure program consistency beyond the program change. Actually, we schedule program changes manually, i.e., we manually identify the points within the program on which it is safe to change the program. Unfortunately, this requires deep knowledge about the program and its control flow. Currently, we accumulate this knowledge through source code reviews and (manual) stack analysis. Unfortunately, the manual processing of program changes is a complex and error-prone task. Therefore, we consider to adopt approaches for consistent program changes from Makris and Bazzi [24], and Subramanian et al. [25].

8. Related Work

In recent work, various approaches for runtime adaptations in Java have been suggested.

First to say is that Java HotSwap as we use it in our runtime adaptation approach was also developed with respect to unanticipated runtime program adaptation (note that runtime debugging is just a special kind of runtime program adaptation). However, various approaches exist which solely use Java HotSwap for unanticipated runtime adaptation concerns. For example, *AspectWerkz* [10], [26], *Wool* [11], *PROSE* [8], [27], and *JAsCo* [12] utilize Java HotSwap in order to apply aspects [28] to the running application. What these approaches have in common with all other runtime adaptation approaches solely using Java Hotswap is the absence of functions which aim on class schema changes.

Like Java HotSwap object wrapping is also subject of numerous runtime adaptation approaches. Hunt and Sitaraman describe in [29] an object wrapping approach which also allows to stepwise extend the interface of the wrapping using dynamic proxies. Kniesel presents in [30] an object wrapping approach which adds type-safe delegation to Java. Büchi and Weck [31] introduce *Generic Wrappers* which solve the problem of wrapper transparency. Bettini et al. [32], [33] present *Featherweight wrap Java*, an extension for Java like languages which allows type-safe object wrapping. What all mentioned approaches are missing is the ability to apply the wrapping itself to the running application in an unanticipated way.

Gregersen and Jørgensen presented in [34] a component based runtime adaptation approach which enables unanticipated changes also including class schema changes. However, the current implementation requires *Java NetBeans* and cannot be applied to standalone applications. Additionally, it aims at hidding the update process (update transparency) and thus denies state mappings which require user input.

Short after we presented our previous runtime adaptation approach in [35], Kim and Tilevich presented a similar approach based on Java HotSwap and Proxies [36]. It allows to add new methods and fields to a running program. However, this approach does not allow to remove methods defined in the proxy and to change inheritance relationships. Additionally, it uses Helper classes to change the program which causes the self-problem.

In our new runtime adaptation approach we use class replacements to modify a program. Liang and Bracha describe in [23] how customized class loader can be used to reload a class. However, reloading classes with changed class schema was out of their scope. Malabarba et al. [37] suggest type-safe dynamic class replacements also including class schema changes. Unfortunately, they have to use a modified Java virtual machine. JavaRebel[5] is a commercial tool used to quickly develop Java programs. It allows to change a class (including the schema) and to reload it into the running program. But, it does not allow to change inheritance relationships.

9. Conclusion

Runtime program adaptation is a reasonable approach to maintain applications while improving the user experience and avoiding time periods of unavailability. In order to change at runtime all program parts that appear in a program (see Table 1 points 1 - 13) it must be enabled for unanticipated changes even of already executed program parts. This is due to the fact that the kind of change becoming necessary and the time of its application cannot be foreseen when the program starts.

Our previous runtime adaptation approach was the first solution that served the requirements described above. It also allowed program execution on top of a standard Java virtual machine, i.e., Sun's Hotspot VM. Unfortunately, the previous approach has several limitations which reduce its applicability. They were caused by the mechanisms used, e.g., object wrappings and interfaces.

The new runtime adaptation approach overcomes these limitation while barely introducing new ones (see Table 3). It substitutes object wrappings for class replacements and interfaces for mediators. Additionally, it automates runtime program adaptation.

5. http://www.zeroturnaround.com/javarebel/

Like in the previous runtime adaptation approach program changes must be still scheduled manually in order to keep the program consistent. This is a complex and error-prone task that requires deep knowledge about the program and its control flow. In further work we plan to extend our runtime adaptation approach so that it automatically schedules program changes while keeping the program consistent.

10. Acknowledgements

Mario Pukall's work is part of the RAMSES project[6] which is funded by DFG (Project SA 465/31-2).

References

[1] L. Erlikh, "Leveraging Legacy System Dollars for E-Business," *IT Professional*, 2000.

[2] J. Moad, "Maintaining the Competitive Edge," *DATAMATION*, 1990.

[3] G. Bracha, "Objects as Software Services," 2005, Invited talk at the International Conference on Object-Oriented Programming, Systems, Languages, and Applications.

[4] B. Fulgham and I. Gouy, "The Computer Language Benchmarks Game," http://shootout.alioth.debian.org/.

[5] S. Chiba and M. Nishizawa, "An Easy-to-Use Toolkit for Efficient Java Bytecode Translators," in *Proceedings of the second International Conference on Generative Programming and Component Engineering*, 2003.

[6] S. Chiba, "Load-Time Structural Reflection in Java," *Lecture Notes in Computer Science*, 2000.

[7] M. Dahm, "Byte Code Engineering," in *Java-Informations-Tage*. Springer-Verlag, 1999.

[8] A. Nicoara, G. Alonso, and T. Roscoe, "Controlled, systematic, and efficient code replacement for running java programs," in *Proceedings of the EuroSys Conference*, 2008.

[9] A. Orso, A. Rao, and M. Harrold, "A Technique for Dynamic Updating of Java Software," in *Proceedings of the International Conference on Software Maintenance*, 2002.

[10] 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*, 2004.

[11] 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*, 2003.

[12] W. Vanderperren and D. Suvee, "Optimizing JAsCo dynamic AOP through HotSwap and Jutta," in *Proceedings of the 1st AOSD Workshop on Dynamic Aspects*, 2004.

[13] M. Pukall, C. Kästner, and G. Saake, "Towards Unanticipated Runtime Adaptation of Java Applications," in *Proceedings of the 15th Asia-Pacific Software Engineering Conference*. IEEE Computer Society, 2008.

[14] "Java Platform Debugger Architecture," Sun Microsystems, Tech. Rep., http://java.sun.com/javase/6/docs/technotes/guides/jpda/index.html.

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

6. http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/ramses/

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

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

[18] 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*, 2001.

[19] ——, "Safe Class and Data Evolution in Large and Long-Lived Java Applications," Ph.D. dissertation, University of Glasgow, 2001. [Online]. Available: citeseer.ist.psu.edu/dmitriev01safe.html

[20] 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*, 1986.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997.

[22] "Java Virtual Machine Tool Interface Version 1.1," Sun Microsystems, Tech. Rep., 2006, http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html.

[23] S. Liang and G. Bracha, "Dynamic Class Loading in the Java Virtual Machine," in *Proceedings of the 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1998.

[24] K. Makris and R. Bazzi, "Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction," Department Of Computer Science And Engineering, Arizona State University, Tech. Rep., 2008, TR-08-007.

[25] 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*, Jun. 2009.

[26] J. Bonér, "AspectWerkz – dynamic AOP for Java," *Invited talk at 3rd International Conference on Aspect-Oriented Software Development*, 2004.

[27] A. Nicoara and G. Alonso, "Dynamic AOP with PROSE," in *Proceedings of the CAiSE'2005 Workshop on Adaptive and Self-Managing Enterprise Applications*, 2005.

[28] G. Kiczales, J. Lamping, A. Mendhekar, C. L. C. Maeda, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the European Conference on Object-Oriented Programming*, 1997.

[29] J. Hunt and M. Sitaraman, "Enhancements: Enabling Flexible Feature and Implementation Selection," in *Proceedings of the International Conference on Software Reuse*, 2004.

[30] G. Kniesel, "Type-Safe Delegation for Run-Time Component Adaptation," in *Proceedings of the European Conference on Object-Oriented Programming*, 1999.

[31] M. Büchi and W. Weck, "Generic Wrappers," in *Proceedings of the European Conference on Object-Oriented Programming*, 2000.

[32] L. Bettini, S. Capecchi, and E. Giachino, "Featherweight wrap Java," in *Proceedings of the ACM symposium on Applied computing*, 2007.

[33] L. Bettini, S. Capecchi, and B. Venneri, "Extending Java to dynamic Object Behaviors," in *Proceedings of the ETAPS Workshop on Object-Oriented Developments*, 2003.

[34] A. R. Gregersen and B. N. Jørgensen, "Dynamic update of java applications - balancing change flexibility vs programming transparency," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 2, 2009.

[35] M. Pukall, “Object Roles and Runtime Adaptation in Java,” in *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2008.

[36] D. Kim and E. Tilevich, “Overcoming JVM HotSwap Constraints via Binary Rewriting,” in *Proceedings of the Workshop on Hot Topics in Software Upgrades*, 2008.

[37] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes, “Runtime Support for Type-Safe Dynamic Java Classes,” in *Proceedings of the 14th European Conference on Object-Oriented Programming*, 2000.