Towards Unanticipated Runtime Adaptation of Java Applications

Mario Pukall, Christian Kästner, and Gunter Saake
University of Magdeburg
39106 Magdeburg, Germany
{mario.pukall, kaestner, saake}@iti.cs.uni-magdeburg.de

Abstract

Modifying an application usually means to stop the application, apply the changes, and start the application again. That means, the application is not available for at least a short time period. This is not acceptable for highly available applications. One reasonable approach which faces the problem of unavailability is to change highly available applications at runtime. To allow extensive runtime adaptation the application must be enabled for unanticipated changes even of already executed program parts. This is due to the fact that it is not predictable what changes become necessary and when they have to be applied. Since Java is commonly used for developing highly available applications, we discuss its shortcomings and opportunities regarding unanticipated runtime adaptation. We present an approach based on Java HotSwap and object wrapping which overcomes the identified shortcomings and evaluate it in a case study.

1. Introduction

When thinking about software development and software maintenance there is just one thing which is predictable - alteration. Nearly all applications with a life cycle longer than just a few days and an average degree of complexity have to be adapted sooner or later. Well known reasons for program adaptation are new requirements and incorrect program code. To change a program, it is usually necessary to stop the program, apply a patch and to start it again. Furthermore, some systems need a considerable startup time, or time until they run with desired performance, e.g., because caches first need to be filled. This means that for each adaptation the application is not available for at least a short time period. This conflicts with applications which have to be highly available, e.g., security applications, web applications, and banking systems. Therefore, in this paper, we aim at modifying highly available applications at runtime without any downtime.

Literature suggests many different ways to change applications at runtime (see survey in [28]). They can be distinguished by the point of time changes are applied and by the degree in which changes are anticipated. One alternative is to change a program at load-time, i.e., before the program part (e.g., a class) to be changed is deployed. Another alternative is to change a program during deploy-time, i.e., when the program part to be changed is already loaded. Furthermore, anticipated program adaptation means to prepare the program for requirements which are expected to become relevant in the future. Unanticipated program adaptation describes the ability to adapt a running program according unpredictable requirements. We believe that a program cannot be prepared for all future requirements. For that reason, our approach targets at deploy-time adaptation that can also apply unanticipated adaptation.

Despite the fact that dynamic languages like Smalltalk, Python or Ruby directly support unanticipated runtime program adaptation up to deploy-time, we discuss Java in this paper, because Java is a mainstream programming language commonly used in highly available applications, e.g., *Apache Tomcat*[1], *Java DB*[2], *JBoss Application Server*[3], or *SmallSQL*[4]. Furthermore, compared to dynamic languages Java's speed of execution is better in most fields of application [13]. Unfortunately, runtime program adaptation in statically typed languages such as Java is a challenging issue [6]. Statically typed languages do not natively offer enough powerful instruments for runtime adaptation. For example, Java does not allow dynamic schema changes of already loaded classes (Java's Reflection API offers only read access), which is actually the precondition for extensive unanticipated runtime adaptation of Java applications.

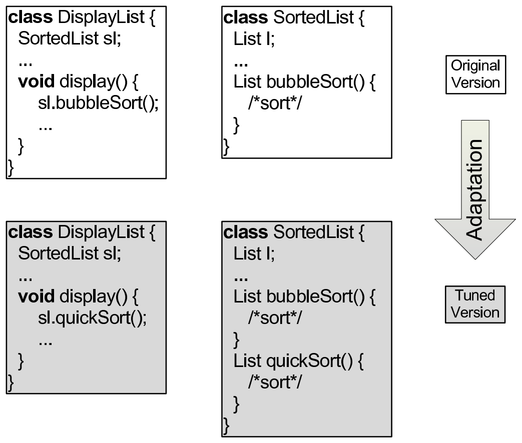
In recent work, some approaches for runtime adaptation in Java have been suggested. First, there are several approaches that allow unanticipated modifications only until load-time but not at deploy-time, e.g., *Javassist* [9, 8]

[1]http://tomcat.apache.org/index.html
[2]http://developers.sun.com/javadb/
[3]http://www.jboss.org/jbossas/
[4]http://www.smallsql.de/

or *BCEL*[5]. Second, there are approaches that allow anticipated changes, e.g., object wrapping [19, 3] or static aspect-oriented programming [17]. Finally, *PROSE* [26, 25], *DUSC* [27], or *AspectWerkz* [5] allow unanticipated adaptation up to deploy-time. However, in terms of PROSE the application must the run in a modified Java virtual machine (JVM) which is not possible in many environments. DUSC does not allow of keeping the program state while changing the schema of classes. Unanticipated runtime adaptation in AspectWerkz is restricted to class schema keeping program changes. None of these approaches allows to adapt (1) stateful Java programs at (2) deploy-time while (3) running in a standard JVM, even in an (4) unanticipated way also including (5) class schema changes.

In this paper we present an approach which enables stateful Java applications for unanticipated runtime adaptation even during deploy-time – as only known from dynamic languages. It works with the Java HotSpot virtual machine which is the standard virtual machine of Sun's current Java 2 platform. We overcome the limitations of previous approaches by combining object wrapping techniques with modifications of the runtime code using Java HotSwap technology, that has been officially introduced in *Java Virtual Machine Debug Interface* of Sun's Java 2 Standard Development Kit version 1.4[6] [15, 16]. We also demonstrate the applicability of our approach in a non-trivial case study.

2. Motivating Example



```
class DisplayList {
  SortedList sl;
  ...
  void display() {
    sl.bubbleSort();
    ...
  }
}
```

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

```
class DisplayList {
  SortedList sl;
  ...
  void display() {
    sl.quickSort();
    ...
  }
}
```

```
class SortedList {
  List l;
  ...
  List bubbleSort() {
    /*sort*/
  }
  List quickSort() {
    /*sort*/
  }
}
```

Figure 1. Unanticipated adaptation.

Unanticipated runtime adaptation is a non trivial operation that usually affects different parts of a program. Figure 1 illustrates a program which manages and displays sorted lists. The original version of the program (upper part of Figure 1) uses the *BubbleSort* algorithm to sort the list content. While the program is running, the requirements change. Due to execution speed penalties when sorting large lists the *BubbleSort* algorithm must be replaced by a faster sorting algorithm, e.g., the *QuickSort* algorithm. In order to satisfy the new requirements, the program must be adapted. In this example class *SortedList* is enhanced by method *quickSort()* which implements the *QuickSort* algorithm. Additionally, method *display()* of class *DisplayList* uses *quickSort()* instead of *bubbleSort()* to faster display the list content.

Assuming that the list content of *SortedList* is part of a banking transaction, stopping the program in order to apply the necessary changes will result in aborting the transaction. For that reason the program has to be changed at runtime.

3. Runtime Program Adaptation in Java

To understand how runtime adaptation is possible or restricted in Java it is first necessary to know how the Java virtual machine works. In Java's virtual machine a program is represented in the *heap* as well as in the *method area* (as shown in Figure 2). The heap is the memory area which stores the runtime data of all class instances [23]. Just like the method area it is shared by all program threads. The Java virtual machine explicitly allocates heap memory for new class instances while the *garbage collector* cleans the heap memory from class instances which are not longer referenced. The method area stores all class (type) specific data such as runtime constant pool, field and method data, and the code for methods and constructors [23].

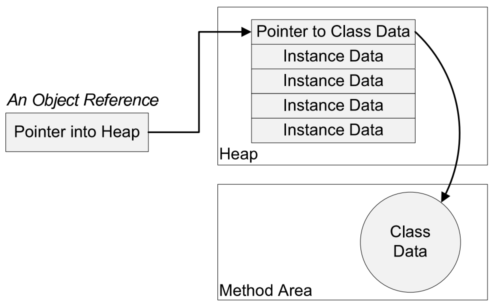


Figure 2. Object representation in the Java HotSpot virtual machine [32].

In order to change a program running in the Java virtual machine the data stored in the heap and in the method area have to be modified. For example adding a method to a class requires to change the class data, more precisely it requires to change the class schema of the class. Unfortunately, the Java virtual machine does not allow class schema changes. This is due to the fact that class schema changes require to

[5]http://jakarta.apache.org/bcel/

[6]Since Sun Java Development Kit version 6 replaced by the *Java Virtual Machine Tool Interface*.

synchronize the heap data and the class data. The virtual machine does not provide functions for such synchronizations.

One approach for runtime program adaptation which also enables class schema changes is class replacement. It aims at replacing the class data while also replacing all related class instances. The applicability of this approach depends on two facts: object to object interconnection and object to class interconnection. The Java virtual machine specification does not strictly specify these interconnections [23]. Therefore two different implementations are possible. The first implementation is shown in Figure 2. An object refers to other objects respectively to its class directly, i.e., there is no indirection in object to object interconnection and object to class interconnection. Figure 3 illustrates the second implementation which was used by, e.g., early versions of Sun's Java virtual machine [24]. Here, the interconnections are implemented via object handles. Thus, an object is never directly connected with other objects respectively with its class. Using the later implementation runtime program adaptation via class replacement can be easily achieved by switching the pointers of the object handle to the updated elements. Unfortunately, the level of indirection caused by the object handles result in performance deficits. Due to this fact Java virtual machines that are shipped for productive usage such as Sun's HotSpot virtual machine implement the first approach [1, 24]. Class replacement based on such virtual machines is difficult. One class replacement strategy is to unload the class, to redefine the class, and to reload the redefined class. However, in order to determine the Java virtual machine to unload a class all objects of the class including the corresponding class loader have to be located and de-referenced (note that only user defined class loaders can be de-referenced). Additionally, the objects have to be recreated according the new class definition. Another class replacement strategy is to create a new class loader and let the new class loader load the newest version of a class (see also [21]). Due to the fact that the old class version is still alive in the virtual machine

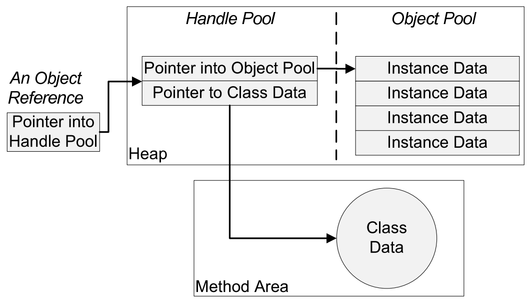


Figure 3. Handle based object representation [32].

all versions of a class as well as their instances have to be managed. What both class replacement strategies have in common is, that each object which replaces an old object has to be initialized with the state of the old object.

Java HotSwap. Beside all limitations regarding runtime program adaptation the Java HotSpot virtual machine allows to swap method implementations at runtime. This mechanism – referred to as *Java HotSwap* – is implemented by the Java Virtual Machine Tool Interface [16].

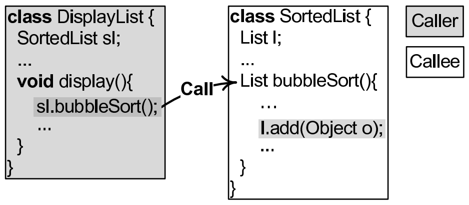
Java HotSwap aims at class data restructuring. The restructuring affects the following class data elements: the *constant pool*, the *method array*, and the *method objects*. The constant pool consists of symbolic constants which the virtual machine refers in order to execute program instructions. The method array indexes the method objects of the class. Method objects store the byte code of methods.

The Java HotSwap algorithm approximately works as follows (for further details see [11, 12]): First, a new version of the class to be modified is created. It contains the reimplemented methods. Second, it is checked if both classes share the same class schema. Third, the links to the constant pool, method array, and method objects of the old class are successively (in the given order) redirected to the (up-to-date) counterparts of these three elements within the new class. After class redefinition all corresponding method calls refer to the reimplemented methods. Unfortunately, Java HotSwap neither allows to swap the complete class data, nor removing or adding methods, i.e., class schema changes are not allowed. This is due to the fact that class schema changes let the data of the heap and the class become inconsistent.

4. Object Roles in Respect to Runtime Adaptation

Systematic runtime adaptation of Java programs requires deep analysis. First, it must be discovered which objects have to be adapted. Second, for each discovered object the necessary changes must be identified. We observed that the degree of adaptation depends on the role an object plays, whether it acts as a *caller* or a *callee*. Figure 4 illustrates the meaning of objects playing role caller and objects playing role callee. It also illustrates that objects usually participate at both roles concurrently. Here, objects of type *SortedList* act as callee while used by *DisplayList*, whereas acting as caller when using object l of type *List*.

Degree of Callee Adaptation. A callee can be understood as a "service provider" which offers functions to its callers. In our example callee *SortedList* offers function *bubbleSort()* to caller *DisplayList*. In order to satisfy the new requirements *SortedList* adds method *quickSort()* to its function set. Table 1 lists some other callee changes which aim at function set modifications. What they all have in



```
class DisplayList {
  SortedList sl;
  ...
  void display(){
    sl.bubbleSort();
    ...
  }
}
```

```
class SortedList {
  List l;
  ...
  List bubbleSort(){
    ...
    l.add(Object o);
    ...
  }
}
```

Figure 4. Object roles.

common is that they cause class schema changes, which is not supported by the Java HotSpot virtual machine.

Adaptation	Description
new protected or public method	new provided function
new private method	new internal function
removed protected or public method	removed provided function
removed private method	removed internal function
changed method signature	changed arguments
new field	changes object state
removed field	changes object state
new superclass	new provided functions

Table 1. Abstract of callee modifications.

Degree of Caller Adaptation. The reason for caller adaptation is to modify calls to functions. The caller may want to access changed, additional, or alternative functions. However, we found that these function calls are implemented within the callers methods, except for function calls which initialize class variables. Due to this fact changing a caller only requires modifications of the callers method implementations such as shown in our initial example in Figure 1, where method *display()* of class *DisplayList* is reimplemented in order to call method *quickSort()* instead of method *bubbleSort()* of class *SortedList*. What is needed is a mechanism which allows runtime method implementation replacement.

5. Role-based Runtime Adaptation Approach

In Section 4, we identified the requirements at caller and callee which have to be satisfied to qualify a Java program for runtime adaptation. As shown in Section 3, Java does not offer adequate functions by its own to serve all these requirements. In this section, we present an approach which overcomes Java's shortcomings in terms of runtime program adaptation. In short, we combine the mechanism of object wrapping to change callees, while utilizing Java HotSwap to change callers and to process the wrapping.

5.1. Callee Runtime Adaptation

As we described in Section 3 class replacement is one strategy to achieve runtime program adaptation which also includes callee changes. However, we also described that the implementation of the Java HotSpot virtual machine (without object handles) makes runtime program adaptation via class replacements an difficult and error-prone task. For that reason we chose another approach for runtime callee changes – *object wrapping*.

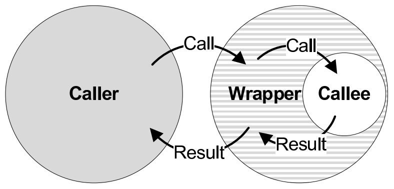


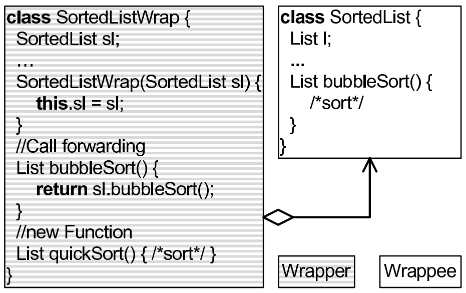
Figure 5. Correlation between wrapper, wrappee, and caller.

As shown in Figure 5 object wrapping means to embed an object within another object. The embedded object is called *wrappee* (in our case wrappee and callee is the same object). The object which encloses the wrappee is called *wrapper*. According the character of object wrapping the wrappee is not truly changed but rather set into a new context. In this context the wrappee continues to provide its functions as usual[7], whereas the wrapper provides new or changed functions. Compared to class replacement object wrapping has two major advantages. First, no class has to be unloaded, redefined and reloaded, i.e., no object has to be de-referenced and re-created. Second, objects keep their state. Another fact that is not considered in this paper but is planned to be exploited in further research is that object wrapping allows to modify single objects instead of whole classes which results in more flexible and more fine-grained program adaptations. Unfortunately, object wrapping blows the program in terms of memory and decreases the readability of the source code. Nevertheless, we believe that object wrapping fits our needs best.

Object Wrapping Application. Having described the general idea of object wrapping we now have to check how to apply it to runtime callee adaptation such as presented in our example, i.e., we have to check how to add method *quickSort()* to callees of type *SortedList*. Figure 6 illustrates a possible solution. Wrapper *SortedListWrap* extends callee *SortedList* by method *quickSort()*, while forwarding calls to *bubbleSort()*. The handover of the callee reference happens in the constructor of *SortedListWrap*.

Dependencies between Wrapper and Wrappee. Considering our simple wrapping example from Figure 6, it is easy to recognize that no complex dependencies between *SortedListWrap* and *SortedList* exist. Method *bubbleSort()* of class *SortedListWrap* only has to take care about forwarding calls to method *bubbleSort()* of class *SortedList*.

[7]The Wrapper has to forward calls to these functions.



```
class SortedListWrap {
  SortedList sl;
  …
  SortedListWrap(SortedList sl) {
     this.sl = sl;
  }
  //Call forwarding
  List bubbleSort() {
     return sl.bubbleSort();
  }
  //new Function
  List quickSort() { /*sort*/ }
}
```

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

Figure 6. Wrapping via composition.

Method *quickSort()* of *SortedListWrap* just needs to access field *List* of *SortedList*. Due to the simplicity of the dependencies the wrapping will produce valid results.

```
1   class SortedList {
2    ...
3    void logging() {...print("SortedList␣ordered"); }
4
5    List bubbleSort() {...this.logging()... }
6   }
7
8   class SortedListWrap {
9    SortedList sl;
10   ...
11   //redefined function
12   void logging() { print("SortedListWrap␣ordered"); }
13
14   //forwarded service;
15   List bubbleSort() { return sl.bubbleSort(); }
16  }
```

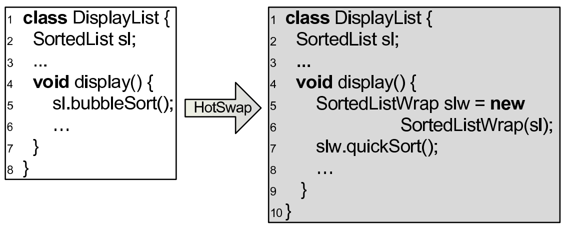
Figure 7. Self-problem and object wrapping.

However, wrapper and wrappee can have more complex dependencies such as shown in Figure 7. *SortedListWrap* forwards calls to method *bubbleSort()* of *SortedList* (Figure 7 line 21). Method *bubbleSort()* of *SortedList* calls method *logging()* (Figure 7 line 8). But, it calls method *logging()* of *SortedList* while a call to method *logging()* of *SortedListWrap* is required. Lieberman [22] first described this phenomenon, also known as the *self-problem*. It results from the fact that the value of ***this*** is bound to the wrappee and not to the wrapper, i.e., wrapper and wrappee make use of consultation instead of delegation. Unfortunately, Java does not natively support delegation. However, over the recent years a couple of approaches have been developed which help to overcome this issue [3, 19, 18, 20].

5.2. Caller Runtime Adaptation

As demonstrated above runtime callee adaptation can be achieved via object wrapping. Now we will have a look into how to change callers and how to apply object wrapping in order to allow unanticipated changes in a running program. Furthermore, we explain how to achieve persistent wrappings by the utilization of dynamic binding and polymorphism.

Wrapping through Caller. In Section 4 we suggested that caller runtime adaptation is motivated by the need to access changed, additional, or alternative callee functions. In order to access these functions the caller has to wrap at first the corresponding callee.



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

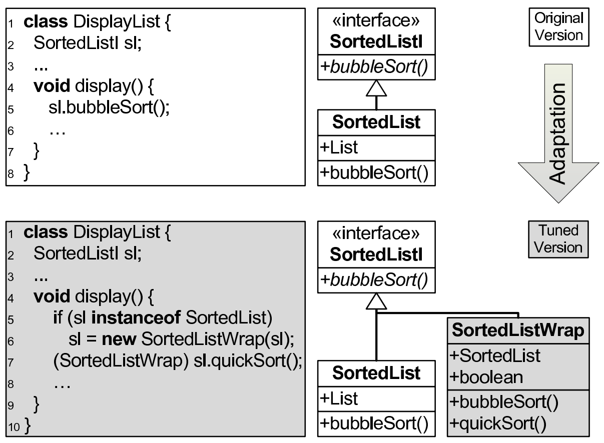
```
1  class DisplayList {
2    SortedList sl;
3    ...
4    void display() {
5       SortedListWrap slw = new
6                 SortedListWrap(sl);
7       slw.quickSort();
8       …
9    }
10 }
```

Figure 8. Dynamic object wrapping using Java HotSwap.

Figure 8 illustrates the proceeding of object wrapping regarding our motivating example. It requires to reimplement method *display()* using Java HotSwap. The reimplementation consists of two parts: the wrapping of callee *SortedList* sl (line 5-6) and the call of aimed method *quickSort()* (line 7).

Temporary vs. Persistent Wrapping. Considering the wrapping example of Figure 8, it can be observed that the wrapping of callee *SortedList* must be proceeded each time function *display()* is called. This is due to the fact that wrapper *SortedListWrap* is just a local variable which is garbage collected when method *display()* has finished, i.e., *SortedList* is wrapped temporary. This is no problem as long as the wrapper is stateless or the state is irrelevant for further program execution. Otherwise the wrapping has to become part of the caller permanently.

In order to achieve such persistent wrappings we exploit Java's ability for dynamic binding and polymorphism. Figure 9 illustrates the required arrangement regarding our example. The static type of callee sl of class *DisplayList* is changed to the interface type *SortedListI* (line 2). Thus, it is possible to assign objects different from type *SortedList* to callee sl. The associated class only has to implement interface *SortedListI*. In our example this affects class *SortedList* and class *SortedListWrap*. The wrapping itself works as depicted down to the left of Figure 9. First, an instance of class *SortedListWrap* is created (line 6). 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 6). In order to call method *quickSort* callee sl has to be casted up to *SortedListWrap* (line 7). This is due to the fact that *quickSort()* is not part of interface *SortedListI*.



```
class DisplayList {
  SortedListI sl;
  ...
  void display() {
    sl.bubbleSort();
    ...
  }
}
```

```
class DisplayList {
  SortedListI sl;
  ...
  void display() {
     if (sl instanceof SortedList)
        sl = new SortedListWrap(sl);
     (SortedListWrap) sl.quickSort();
     ...
  }
}
```

Figure 9. Persistent object wrapping.

5.3. Conditions of Application

Even though our approach based on object wrapping and Java HotSwap aims at unanticipated runtime adaptations its application requires different arrangements at class load-time. First, in order to allow persistent wrappings each program class has to implement at least one interface. Second, all class or instance references to objects of a program class have to be of type of the interface the class implements. Third, to make all variables of a class accessible by a wrapper they either have to be public or they must be accessible through public get methods. Fourth, all final class or instance variables have to be declared as modifiable (by deleting the final attribute).

Even if these arrangements have to be anticipated at class load-time our approach still aims at unanticipated runtime program adaptations. This is because we only anticipate that a given program might be changed in some way instead of anticipating concrete program changes as other approaches do. Unfortunately, our approach violates one of the principles of object-oriented programming – encapsulation. However, the strong coupling of wrapper and wrappee and the demand for flexible and extensive runtime adaptations necessitates this violation. In ongoing work we evaluate the application of packages and class loaders to solve this problem.

6. Case Study

In order to demonstrate the practicability of our role-based runtime adaptation approach we applied it to an existing application. We selected *SmallSQL*[8] for several reasons. First, SmallSQL is a database management system (DBMS) for which runtime adaptation promises benefits of no-downtime. Second, it is entirely written in Java. Third, SmallSQL is an open source application which source code is available for the latest program version and earlier versions. Fourth, there are wide differences between the source code of the actual version of SmallSQL (version 19) and the previous version (version 18) which represents a huge variability of required runtime program changes.

Setup. In general our setup was as follows: we started the old version of SmallSQL (version 18) and adapted it to the new version without shutting down the application. Before starting SmallSQL we had to prepare it, i.e., for each class of the program we introduced an interface, made all final class or instance variables modifiable, and let all private class and instance variables become public (since SmallSQL is organized in one huge package all other class and instance variables had not to be modified).

Having prepared SmallSQL for unanticipated runtime adaptations we started it. Then we implemented the wrappers which contain the new functionality. Afterwards, we modified all classes which call the new functionality, i.e., we wrapped all relevant callees and modified the function calls according the new functionality. Note that the running application remained completely unaffected by these modifications until we determined their application. In other words, the necessary program changes could be implemented independently from the running application which reduced the time period of processing the changes to a minimum. In order to apply the changes a debugger had to be attached to the Java HotSpot virtual machine running SmallSQL. The debugger identified all classes in the class path that have changed and requested the virtual machine to update the byte code of these classes by invoking the Java HotSwap mechanism. To ensure the activity of SmallSQL during the application of the new functionality we continuously ran benchmarks.

Unanticipated Runtime Adaptations. The major improvement of the actual SmallSQL version is the new multi language error message support which radically changes the error message functionality. In order to allow multi language error messages in version 18 of SmallSQL class *SmallSQLException* had to be extended by wrapper *SmallSQLExceptionWrap* which introduced a class field (which refers to the actual language) and 8 new methods. Additionally we had to define 4 new classes (1 for each language and 1 superclass). The deployment of the wrapping via Java HotSwap required to redefine 231 method bodies distributed at 47 classes (Table 2). Due to the fact that objects of wrapper *SmallSQLExceptionWrap* were stateless, temporary wrapping was sufficient.

Another concern that has changed in SmallSQL version 19 addresses the storage of tables and table views. These are now stored in the same file which eases the migration of SmallSQL to other operating systems. In order to embed the

[8]http://www.smallsql.de/

Change	HotSwap (Class / Method)		Wrapper Class	New Class
	Temporary	Persistent		
Language	47 / 231	0 / 0	1	4
Database	11 / 29	2 / 7	1	1

Table 2. Number of modifications.

functionality in SmallSQL version 18 class *Database* had to be wrapped. Wrapper *DatabaseWrap* introduced a new instance field of type *TableViewMap* which is responsible for storing the tables and table views. In order to instantiate the field class *TableViewMap* had to be defined. The wrapper deployment required to redefine 36 methods distributed at 13 classes. Therefrom wrapper *DatabaseWrap* had to become persistent in terms of class *SSConnection* and class *Table*.

Challenges. One of the most challenging problems we had to deal with was the identification of all places in the source code which were required to be changed. This was due to several facts. First, the code to be changed was scattered across the whole program. Second, we had to apply a lot of (different) changes (see Table 2). The next problem we had to solve was to ensure the program's consistency. In order to adapt SmallSQL while keeping the program consistent we had to take care about the order and the timing of the application of the necessary wrappers. This required either a good knowledge about the control flow of the program or tools which help to retrieve this knowledge. Since we were not familiar with the source code of SmallSQL we had to access it via a tool – namely the debugger which was already attached to the virtual machine running SmallSQL. Using the debugger we were able to request the actual stack of function calls which immediately delivered the information we needed to apply the changes while keeping the program consistent.

7. Related Work

In the recent past various runtime adaptation approaches were developed which either exploit object wrapping or Java HotSwap.

Java HotSwap. First to say is that the Java HotSwap approach itself was 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* [31, 4, 5], *Wool* [29, 10], *PROSE* [26, 25], and *JAsCo* [30] utilize Java HotSwap in order to apply aspects [17] 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 at modifications similar to class schema changes.

Object Wrapping. Like Java HotSwap object wrapping is also subject of numerous runtime adaptation approaches. Hunt and Sitaraman describe in [14] an object wrapping approach which also allows to stepwise extend the interface of the wrapping using dynamic proxies. Kniesel presents in [19] an object wrapping approach which adds type-safe delegation to Java. Büchi and Weck [7] introduce *Generic Wrappers* which solve the problem of wrapper transparency. Bettini et al. [2, 3] 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.

8. Conclusion

Runtime program adaptation is one reasonable approach to maintain highly available applications while avoiding time periods of unavailability. In order to allow extensive runtime adaptations programs must be enabled for unanticipated changes even of already deployed 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.

In particular, for Java numerous approaches exist which aim at runtime adaptation. This is because Java is a mainstream programming language which is also commonly used to implement highly available applications. Unfortunately, none of these approaches fulfills all requirements which unanticipated runtime adaptation implies – namely: (1) program changes at deploy-time, (2) the run in a standard virtual machine, (3) state keeping program changes, (4) techniques similar to class schema changes. In this paper we presented an approach based on Java HotSwap and object wrapping which fulfills these requirements. In a case study we demonstrated the practicability of our approach.

The lesson we have learned from the case study is that our approach fits unanticipated runtime adaptations of real life applications. Nevertheless, the manual processing of the required adaptations is a complex and error-prone task. For that reason we plan to provide tool support for runtime program adaptations based on our approach. This includes routines which automatically identify the program parts to be changed as well as routines which ensure the program to be consistent after adaptation. Additionally, we plan to provide solutions for the self-problem possibly based on the Lava compiler which is part of the Darwin project[9].

[9]http://roots.iai.uni-bonn.de/research/darwin/

9. Acknowledgements

Mario Pukall is funded by German Research Foundation (DFG), Project SA 465/31-2. The presented work is part of the RAMSES project[10] funded by DFG.

References

[1] The Java HotSpot Virtual Machine. Technical report, Sun Microsystems, 2001. `http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html`.

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

[3] L. Bettini, S. Capecchi, and B. Venneri. Extending Java to dynamic object behaviors. In *Proceedings of the ETAPS Workshop on Object-Oriented Developments*, 2003.

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

[5] 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.

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

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

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

[9] 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.

[10] S. Chiba, Y. Sato, and M. Tatsubori. Using HotSwap for Implementing Dynamic AOP Systems. In *Proceedings of the Workshop on Advancing the State-of-the-Art in Run-time Inspection*, 2003.

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

[12] 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.

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

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

[15] Java Virtual Machine Debug Interface Reference. Technical report, Sun Microsystems, 2004. `http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jvmdi-spec.html`.

[16] Java Virtual Machine Tool Interface Version 1.1. Technical report, Sun Microsystems, 2006. `http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html`.

[17] 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.

[18] G. Kniesel. Type-Safe Delegation for Dynamic Component Adaptation. In *Proceedings of the Workshop on Object-Oriented Technology*, 1998.

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

[20] G. Kniesel. *Dynamic Object-Based Inheritance with Subtyping*. PhD thesis, University of Bonn, 2000.

[21] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. *SIGPLAN Not.*, 1998.

[22] 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.

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

[24] S. Meloan. The Java HotSpot Performance Engine: An In-Depth Look. Technical report, Sun Microsystems, 1999. `http://java.sun.com/developer/technicalArticles/Networking/HotSpot/`.

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

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

[27] 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.

[28] M. Pukall and M. Kuhlemann. Characteristics of Runtime Program Evolution. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2007.

[29] 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.

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

[31] A. Vasseur. Dynamic AOP and Runtime Weaving for Java – How does AspectWerkz Address It? In *Proceedings of the AOSD Workshop on Dynamic Aspects*, 2004.

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

[10] http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/ramses/