Object Roles and Runtime Adaptation in Java

Mario Pukall

Otto-von-Guericke University, P.O. Box 4120, 39016 Magdeburg, Germany
pukall@iti.cs.uni-magdeburg.de

Abstract. Program maintenance usually decreases the programs availability. This is not acceptable for highly available applications. Thus, such applications have to be changed at runtime. Furthermore, since it is not predictable what changes become necessary and when they have to be applied, highly available applications have to be enabled for unanticipated runtime adaptation at deploy-time [1]. We developed an object role-based approach which deals with these requirements.

1   Introduction

Maintenance of highly available applications, such as banking systems or security applications, is a cost-intensive task. This is due to the fact that maintenance usually causes time periods of unavailability. Unfortunately, such programs can not be prepared statically (i.e., at compile or load-time) for all changes that may become necessary at runtime [1]. For that reason highly available applications must be enabled for unanticipated changes at deploy-time, i.e., for unanticipated changes at already loaded program parts.

Recent work suggests different approaches for runtime program adaptation in Java. Approaches like *Javassist* [2, 3] or *AspectWerkz* [4–6] allow unanticipated changes until load-time, but not at deploy-time. Other approaches allow only for anticipated changes, e.g., object wrapping [7–11]. However, approaches such as PROSE [12, 13] and DUSC [14] allow unanticipated changes at deploy-time. Unfortunately, PROSE uses a modified Java virtual machine (JVM). For that reason the utilization of this approach is restricted to environments which support the PROSE virtual machine. DUSC lacks of object state keeping class updates when simultaneously updating the class interface. We conclude that non of these approaches enables stateful Java programs for unanticipated changes at deploy-time while running in a standard JVM.

In this paper we present the basic idea of an object role-based approach which enables stateful Java applications for unanticipated runtime adaptation even at deploy-time. It works with the Java HotSpot virtual machine[1] and combines object wrapping and Java HotSwap[2].

---

[1] The Java HotSpot virtual machine is the standard virtual machine of Sun's current Java 2 platforms.

[2] Java HotSwap is supported by the Java HotSpot virtual machine.

2 Motivating Example

Similar to static program changes, runtime program changes usually effect different parts of a program. Figure 1 depicts a simple program which manages and displays sorted lists. At the moment of program start it offers the *bubble sort* algorithm in order to sort a list. The length of the lists which have to be sorted grows while the program is running. At some point of execution time it is noticed that the bubble sort algorithm is to slow to sort the lists in reasonable time. For that reason the bubble sort algorithm has to be replaced by a faster sorting algorithm, e.g., the *quick sort* algorithm. In order to apply the required changes class *SortedList* as well as class *DisplayList* must be modified (Figure 1).
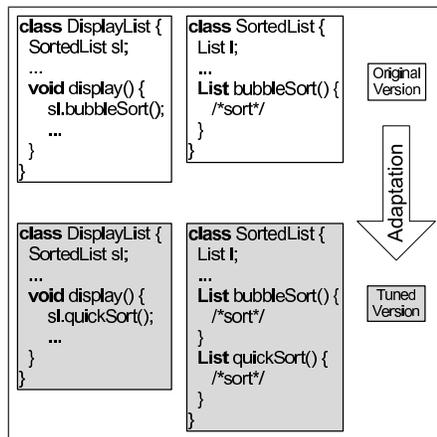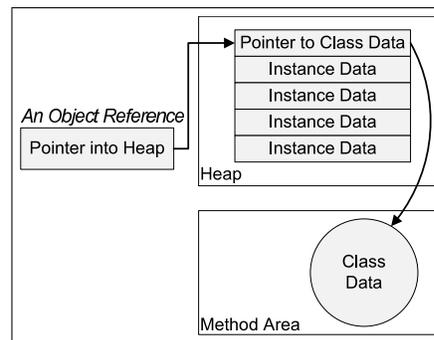

Fig. 1. Unanticipated adaptation. **Fig. 2.** Programs in the HotSpot VM [15].

3 Runtime Changes and the Java Virtual Machine

To understand the restrictions and possibilities for runtime program adaptation in the HotSpot VM it is necessary to know how a program is represented in the virtual machine. As depicted in Figure 2 the *heap* of the HotSpot VM stores the runtime data of all class instances, whereas the *method area* is the memory area of the HotSpot VM which stores all class (type) specific data.

The most adequate approach to alter a running program in Java is to replace a class in the JVM and update its objects according to the changes. However, this is difficult to realize in the HotSpot VM, since object references, object data, and class data are directly wired (see Figure 2). In order to replace a class in this virtual machine all instances of the class have to be destroyed and recreated.

Beside these restrictions the HotSpot VM enables method implementation swapping at runtime. This mechanism is called Java HotSwap and is provided

by the Java Virtual Machine Tool Interface [16]. Unfortunately, Java HotSwap does not allow to remove or add methods.

4 Runtime Changes and Object Roles

To systematically adapt a running program it is necessary to identify what objects have to be changed and what changes have to be applied to each object. We observed that the degree of changes depends on the role an object plays in the adaptation context, whether it acts as a *caller* or a *callee*. For instance in Figure 1 an object of class *DisplayList* acts as a caller (i.e., it uses functions of class *SortedList*), whereas an object of class *SortedList* acts as a callee.

4.1 Kinds of Callee Changes.

A callees job is to offer its functions to other objects (callers). To satisfy the requirements of its callers it may have to provide new or changed functions. For example callee *SortedList* must be extended by method *quickSort()* in order to speed up the display function of caller *DisplayList*. Due to the variety of callee changes we believe that, in order to offer new or changed functions, nearly each part of a callee has to be changeable.

4.2 Kinds of Caller Changes.

The reason for changing an object in its role as a caller is to call new, changed, or alternative functions provided by the callees it owns. These calls are largely implemented within the callers methods. For that reason changing a caller only requires modifications of the caller method implementation that contains the function call chosen for adaptation. For instance in our scenario method *display()* of class *DisplayList* has to be changed in order to call method *quickSort()* instead of method *bubbleSort()* of class *SortedList*.

5 Object Wrapping and Java HotSwap

In the following we present the basic idea of a runtime program adaptation approach which serves the required changes at objects playing role callee and objects playing role caller.

5.1 Callee Adaptation using Object Wrapping

An appropriate strategy for runtime callee adaptation is object wrapping. It means to embed the callee within another object denoted as *wrapper*. Within the wrapping the callee still provides its functions as usual, whereas the wrapper adds the necessary changes. Compared to the strategy of class replacement (as suggested in Section 3) object wrapping induces two major advantages. First,

the callee's class must not be unloaded, redefined and reloaded, i.e., the class instances must not be destroyed and recreated. Second, the callee keeps its state.

Remembering our motivating example from Section 2 a callee of type *SortedList* can be extended via a wrapping such as shown in Figure 3. Here wrapper *SortedListWrap* adds the required quick sort algorithm (method *quickSort()*), while it forwards calls to method *bubbleSort()* of callee *SortedList*. The handover of the callee reference happens in the constructor of the wrapper.
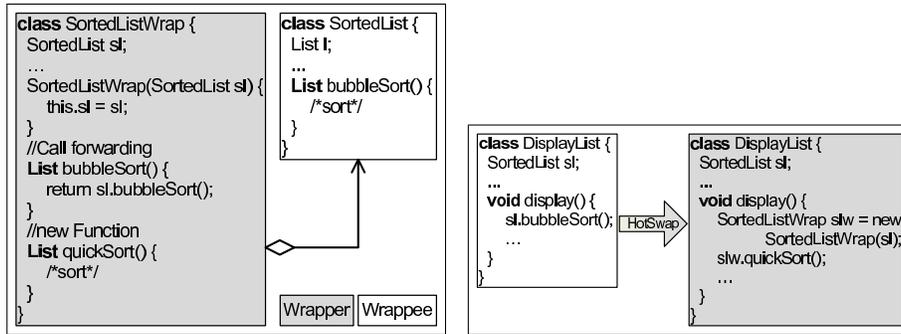

Fig. 3. Callee adaptation via wrapping. **Fig. 4.** Caller adaptation via Java HotSwap.

5.2   Caller Adaptation using Java HotSwap

While callee adaptation can be achieved using object wrapping, two open issues exist. First, the wrapping must be deployed. Second, the function calls of the caller have to be changed. Both tasks can be performed using Java HotSwap. Figure 4 illustrates the procedure according our motivating example. In order to apply the quick sort algorithm to *DisplayList* the implementation of method *display()* is swapped. The new method implementation wraps callee *SortedList* by wrapper *SortedListWrap* and calls the *quickSort()* method.

6   Conclusion and Future Work

In this paper we proposed unanticipated runtime program adaptation at deploy-time as an issue of changing objects. We suggested that the necessary degree of object changes depends on the role an object plays, i.e., whether it acts as caller or callee. Unfortunately, standard Java virtual machines, such as the Java HotSpot virtual machine, do not natively offer functions for all required object changes. For that reason we developed an approach which serves the whole bandwidth of required object changes. It combines object wrapping and Java HotSwap in order to enable unanticipated runtime adaptation at deploy-time.

Even though the basic approach presented in this paper is suitable for many use cases, a lot of open questions exist. In current work we look into how to

achieve consistency and how to apply persistent wrappings. We also evaluate the execution speed of programs which are modified using our runtime adaptation approach.

References

1. Pukall, M., Kuhlemann, M.: Characteristics of runtime program evolution. In Cazzola, W., Chiba, S., Coady, Y., Ducasse, S., Kniesel, G., Oriol, M., Saake, G., eds.: Proceedings of ECOOP'2007 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'07), Berlin, Germany (2007) 51–57
2. Chiba, S., Nishizawa, M.: An easy-to-use toolkit for efficient java bytecode translators. In: Proceedings of the second International Conference on Generative Programming and Component Engineering (GPCE'03). (2003)
3. Chiba, S.: Load-time structural reflection in java. Lecture Notes in Computer Science (2000)
4. Vasseur, A.: Dynamic aop and runtime weaving for java – how does aspectwerkz address it? In: DAW: Dynamic Aspects Workshop. (2004)
5. Bonér, J.: Aspectwerkz – dynamic aop for java. Invited talk at 3rd International Conference on Aspect-Oriented Software Development (AOSD). (2004)
6. Bonér, J.: What are the key issues for commercial aop use: how does aspectwerkz address them? In: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD). (2004)
7. Hunt, J., Sitaraman, M.: Enhancements: Enabling flexible feature and implementation selection. In: Proceedings of the 8th International Conference on Software Reuse (ICSR'04). Lecture Notes in Computer Science, Springer (2004) 86–100
8. Kniesel, G.: Type-safe delegation for run-time component adaptation. In: Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99), London, UK, Springer-Verlag (1999) 351–366
9. Bettini, L., Capecchi, S., Venneri, B.: Extending java to dynamic object behaviors. In: Proceedings of the ETAPS'2003 Workshop on Object-Oriented Developments (WOOD'03). Volume 82 of ENTCS. (2003)
10. Büchi, M., Weck, W.: Generic wrappers. In Bertino, E., ed.: Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00). Volume 1850 of LNCS. (2000) 201–225
11. Bettini, L., Capecchi, S., Giachino, E.: Featherweight wrap java. In: Proceedings of the 2007 ACM symposium on Applied computing (SAC'07), New York, NY, USA, ACM (2007) 1094–1100
12. Nicoara, A., Alonso, G., Roscoe, T.: Controlled, systematic, and efficient code replacement for running java programs. In Sventek, J., Hand, S., eds.: Proceedings of the 2008 EuroSys Conference, ACM (2008) 233–246
13. Nicoara, A., Alonso, G.: Dynamic aop with prose. In Castro, J., Teniente, E., eds.: Proceedings of the CAiSE'2005 Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA'05), FEUP Edições, Porto (2005) 125–138
14. Orso, A., Rao, A., Harrold, M.: A technique for dynamic updating of java software. In: Proceedings of the International Conference on Software Maintenance (ICSM'02), Washington, DC, USA, IEEE Computer Society (2002) 649–658
15. Venners, B.: Inside the Java 2 Virtual Machine. Computing McGraw-Hill. (2000)
16. Sun: Java virtual machine tool interface version 1.1. Technical report, Sun Microsystems (2006) http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html.