JavAdaptor: Unrestricted Dynamic Software Updates for Java

Mario Pukall
University of Magdeburg
mario.pukall@iti.cs.uni-magdeburg.de

Alexander Grebhahn, Reimar Schröter
University of Magdeburg
{alexander.grebhahn, reimar.schroeter}@st.ovgu.de

Christian Kästner
Philipps-University Marburg
kaestner@informatik.uni-marburg.de

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

Sebastian Götz
University of Dresden
sebastian.goetz@acm.org

ABSTRACT

Dynamic software updates (DSU) are one of the top-most features requested by developers and users. As a result, DSU is already standard in many dynamic programming languages. But, it is not standard in statically typed languages such as Java. Even if at place number three of Oracle's current request for enhancement (RFE) list, DSU support in Java is very limited. Therefore, over the years many different DSU approaches for Java have been proposed. Nevertheless, DSU for Java is still an active field of research, because most of the existing approaches are too restrictive. Some of the approaches have shortcomings either in terms of flexibility or performance, whereas others are platform dependent or dictate the program's architecture. With *JavAdaptor*, we present the first DSU approach which comes without those restrictions. We will demonstrate JavAdaptor based on the well-known arcade game *Snake* which we will update stepwise at runtime.

Categories and Subject Descriptors

D. [**Software**]: Programming Techniques; Software Engineering; Programming Languages

General Terms

Languages, Design, Algorithms

Keywords

Dynamic Software Updates, Software Maintenance, Tool Support.

Copyright is held by the author/owner(s).
ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA
ACM 978-1-4503-0445-0/11/05.

1. INTRODUCTION

Research in the field of dynamic software updates (DSU) has a long tradition and a lot of approaches and solutions have been proposed over the years. Nevertheless, DSU is still an active field of research, because most of the existing DSU approaches are too restrictive. Some of them are *inflexible* (i.e., they do not support all updates that are possible with static program changes) whereas others *require specific runtime environments*, *cause significant performance penalties*, or *dictate the program architecture*. With the arrival of virtual machines, which abstract the runtime environment from the OS and thus offer new starting points for DSU approach development, the situation slightly relaxed and less restrictive DSU approaches came up. However, unrestricted DSU remained to be subject to dynamic languages (with a typical associated performance loss). DSU support for statically typed languages, such as Java, is still restrictive. But, with place number three in Oracle's current request for enhancement (RFE) list,[1], unrestricted DSU is one of the top-most requested features for the Java virtual machine (with ongoing new votes pointing out the lively interest in this RFE). The requester of this RFE argues that better DSU support would ease software development and thus reduce times to market. That is, the developer must not stop and restart the application to test the newly added functionality which is time consuming particularly when large amounts of data have to be imported or several initialization steps must be performed at program start. Even if this is also the main reason for our DSU research efforts, we go one step further and argue that unrestricted DSU would also be valuable in the field of productive program execution. This is because downtimes as a result of software maintenance decrease the availability of a program, which may be costly in terms of highly available applications. Additionally, restarts due to program updates force users to interrupt their tasks, which may decrease the user experience. For all these reasons we have developed *JavAdaptor* which offers unrestricted DSU for Java, i.e., it is *flexible*, *platform independent*, *performant*, and does *not affect the program architecture*.

[1] `http://bugs.sun.com/top25_rfes.do`

We present:
(1) A brief description of the basic ideas behind JavAdaptor.
(2) A short introduction on how to use JavAdaptor.
(3) A practical demonstration of JavAdaptor based on the well-known arcade game *Snake* which we will stepwise update at runtime.

2. STATE OF THE ART

Various approaches for runtime adaptations in Java have been suggested in literature. To set the context for JavAdaptor, we give a brief overview. For a more detailed survey, see [6].

The most common solution for dynamic software updates in Java is *Java HotSwap* [2] which was developed by Dmitriev and became a feature of virtually all standard JVMs. It allows the developer to change method bodies even of already loaded and executed classes. However, changing the schema of loaded classes, which is the key to flexibly update a running program, is not possible with HotSwap.

In contrast to HotSwap, many JVM patches exist which made it not into any standard JVM. Such as *Jvolve* [8] or the *Dynamic Code Evolution VM* [9]. The problem with JVM patches is that they belong to one specific JVM implementation (and version) and may not be applicable to other JVMs, which causes platform dependencies.

Another frequently used approach to upgrade Java's runtime update capabilities are wrappers respectively decorators [3]. The idea is to decorate old program parts in order to update them [7, 5]. The principle drawback of decorators is that the implementation of this pattern requires to completely change the program architecture.

Also common practice to improve the runtime update support in Java is the usage of components which can be dynamically updated. Corresponding approaches are Oracle's WebLogic Server [1] and *Javeleon* [4]. The major downsides of components are their influence on the program architecture and their update granularity (i.e., replacing whole components may be more expensive than replacing only the classes that have changed).

3. JAVADAPTOR

Despite their undisputable quality, existing DSU solutions have restrictions regarding the criteria mentioned above. With *JavAdaptor*, we developed a tool which overcomes the restrictions. That is, it offers flexible changes (such as changing the class hierarchy, adding or removing fields and methods, etc.) to already loaded classes and operates on-top of every standard JVM supporting HotSwap, which is true for virtually all important standard JVMs. At the same time, it does not challenge the program performance and has no influence on the program architecture.

The very basic idea behind the update mechanism implemented by JavAdaptor is twofold. In order to update an already loaded class, we rename the new class version and thus can reload it by the same class loader that loaded the original class (which is by far more performant than reloading classes through customized class loaders). Once the new class version is loaded, we update all callers of the original class. At first, we map all program state of the old class to the new class. Afterwards, we redirect all references of the original class to the new class version. The redirection of the references itself bases on Java HotSwap which allows us

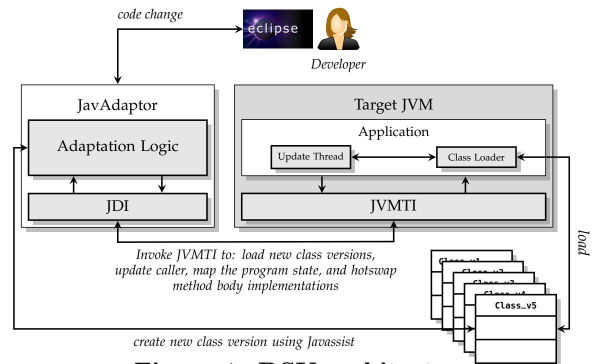

Figure 1: DSU architecture.

to redefine method bodies and thus to modify the references within the method bodies. However, updating the references of global fields as well as method signatures and return types is not possible with HotSwap because those updates would require to change the schema of the caller class. In order to update the callers while avoiding to change their schemas (which otherwise would require to reload the callers as well and let our update mechanism become ineffective), we use constructs such as containers and proxies. These constructs in combination with Java HotSwap allow us to efficiently update the caller side and finally to apply the required changes to the running program. All this comes without significant performance drops (measured under real world conditions). Further details of our update approach can be found in [6].

Technically, we currently provide JavAdaptor as a plug-in, which smoothly integrates into the *Eclipse IDE* (we point out that JavAdaptor could be easily integrated into any other IDE). Figure 2 shows a common application scenario of JavAdaptor from the developers point of view. A developer implements the application in Eclipse (left part of Figure 2), while the current version of the application is already running (right part of Figure 2). Then the developer wants to perform an update without restarting the application. To that end, she simply edits the source code of the application in Eclipse, connects JavAdaptor to the running application (by pushing the tool bar button `C` of JavAdaptor depicted in Figure 2), and triggers the runtime update (by pushing the tool bar button `R` of JavAdaptor shown in Figure 2). JavAdaptor then modifies the bytecode of the classes to be updated in such way that they are compatible with the unchanged and already executed program parts. Furthermore, JavAdaptor reloads the classes to be updated, maps the program state from the outdated classes to the new ones to keep the program state, and updates all callers referring the outdated classes (the corresponding DSU architecture is depicted in Figure 1). All these update steps are transparent to the developer except from state mappings that require user input (e.g., state mappings between different types). After the update, JavAdaptor can be disconnected from the running application. The described update process can be repeated as often as required.

4. SNAKE DEMO

In the accompanying video, we demonstrate JavAdaptor in action. We update the well-known arcade game *Snake* in 4 steps without ever stopping it. In this process we make updates from small changes that only change a method body (that would already been supported by Java HotSwap) to massive changes that introduce new methods, fields, or even

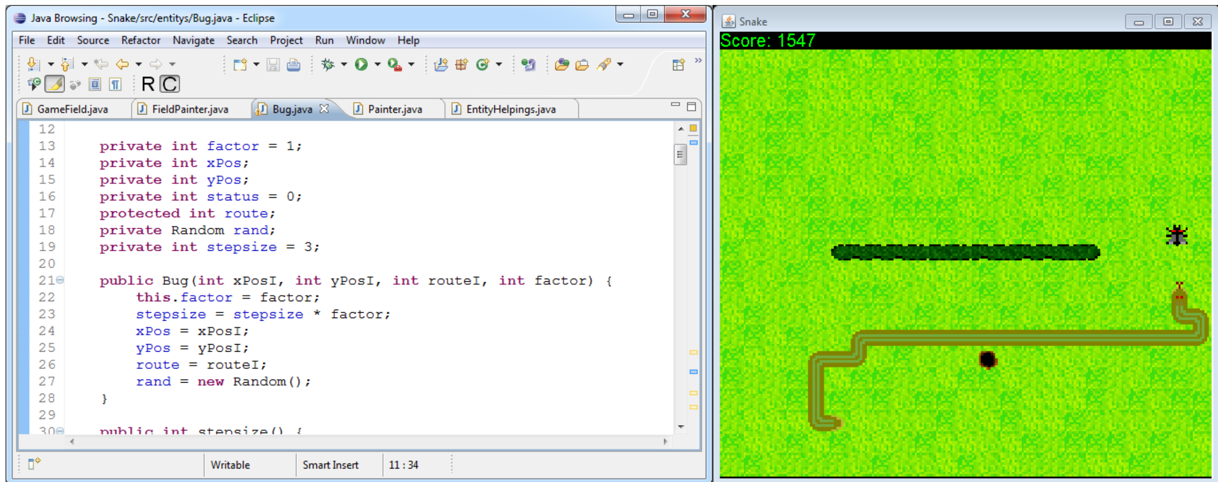

Figure 2: Snake demo.

change class hierarchies (which is not possible with any standard JVM). In the following we go through each update step, describe the functionality it adds, and summarize what kind of code changes it requires.

With the first update step we add a colored map to the running program. Additionally, we introduce a barricade including corresponding extra collision event handling. The program changes required span method redefinitions and method introductions, whereas only method redefinitions are covered by Java HotSwap.

The second update step adds animal objects of type bug to the Snake program. This also includes extra program logic to hunt and catch the bugs. Therefore, we added and redefined methods.

In addition to the bugs, update step 3 adds flies to be hunted to the running program. Because of the similarities of class `Bug` and class `Fly` we changed their inheritance hierarchies, i.e., we let them inherit from newly introduced super class `Entity`. In addition, we redefined and added methods.

With the final update step, we add logic to the program which defines when a level is complete. After level completion, an event is triggered and the snake disappears into a burrow. To implement this functionality, we added a global field and redefined 2 methods.

5. CONCLUSIONS

Unrestricted dynamic software updates are an often requested feature because it eases and improves the software development process, reduces program downtimes, and improves the user experience. Therefore, unrestricted DSU is standard in many dynamic programming languages. But, it is not standard in statically typed languages such as Java. Therefore, many approaches have been suggested in literature to resolve this deficit. But, all approaches suffer from (at least) one of the following restrictions: they are platform dependent, cause performance penalties, affect the program architecture, and/or do not support all updates that are possible with static program changes. With JavAdaptor we presented the first approach (to our best knowledge) which provides Java with flexible dynamic software update capabilities without those restrictions. It allows the developer to flexibly update programs at runtime while keeping the program state and operates on-top of all standard JVMs supporting Java HotSwap. Furthermore, it does not affect the program's performance and architecture.

With our tool demo, we explained the usage of JavAdaptor. Last but not least, we gave a practical demonstration of our tool based on the well-known arcade game *Snake* which we stepwise updated at runtime.

The demo video is available on YouTube:

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

6. ACKNOWLEDGMENTS

Mario Pukall's work is funded by DFG (Project SA 465/31-2). Christian Kästner's work is supported in part by the European Research Council #203099.

7. REFERENCES

[1] Deploying Applications to Oracle WebLogic Server. Technical report, Oracle, 2008.

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

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

[4] 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*, 21(2):81–112, 2009.

[5] R. Pawlak, L. Duchien, G. Florin, and L. Seinturier. Dynamic Wrappers: Handling the Composition Issue with JAC. In *TOOLS*, pages 56–65, 2001.

[6] M. Pukall, C. Kästner, S. Götz, W. Cazzola, and G. Saake. Flexible Runtime Program Adaptations in Java - A Comparison. Technical Report 14, School of Computer Science, University of Magdeburg, 2009.

[7] M. Pukall, C. Kästner, and G. Saake. Towards Unanticipated Runtime Adaptation of Java Applications. In *APSEC*, pages 85–92, 2008.

[8] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic Software Updates: A VM-Centric Approach. In *PLDI*, pages 1–12, 2009.

[9] T. Würthinger, C. Wimmer, and L. Stadler. Dynamic Code Evolution for Java. In *PPPJ*, pages 10–19, 2010.