Feature-oriented Runtime Adaptation

Mario Pukall
University of Magdeburg
Germany
pukall@ovgu.de

Norbert Siegmund
University of Magdeburg
Germany
nsiegmun@ovgu.de

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

ABSTRACT

Creating tailor-made programs based on the concept of software product lines (SPLs) gains more and more momentum. This is, because SPLs significantly decrease development costs and time to market while increasing product's quality. Especially highly available programs benefit from the quality improvements caused by an SPL. However, after a program variant is created from an SPL and then started, the program is completely decoupled from its SPL. Changes within the SPL, i.e., source code of its features do not affect the running program. To apply the changes, the program has to be stopped, recreated, and restarted. This causes at least short time periods of program unavailability which is not acceptable for highly available programs. Therefore, we present a novel approach based on class replacements and Java HotSwap that allows to apply features to running programs.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Extensibility

General Terms

Design

Keywords

Software Product Lines, Runtime Adaption

1. INTRODUCTION

Creating tailor-made programs based on the concept of software product lines (SPLs) gains more and more momentum. This is because SPLs significantly reduce development costs and time to market while increasing product's quality and use-case suitability. Particularly programs that must be available 24 hours a day, 7 days a week (e.g., security applications, banking software, etc.) benefit from the quality improvements caused by an SPL, e.g., in terms of reliability and performance. However, after a program variant is created from an SPL and then started, the program is completely decoupled from its SPL. This is, changing the SPL after program start (such as depicted in Figure 1 where feature *BubbleSort*, as part of a DBMS[1] SPL, is replaced by feature *QuickSort*) does not affect the running program. Usually, applying the changes of an SPL to its running program variants requires to stop the programs and recreate them (according to the modifications made in the SPL). For instance, the changes within the DBMS SPL require to recreate all variants using feature *QuickSort* instead of feature *BubbleSort*. Figure 2 outlines the program parts that are changed in order to replace feature *BubbleSort* (white background) by feature *QuickSort* (dark-gray background).

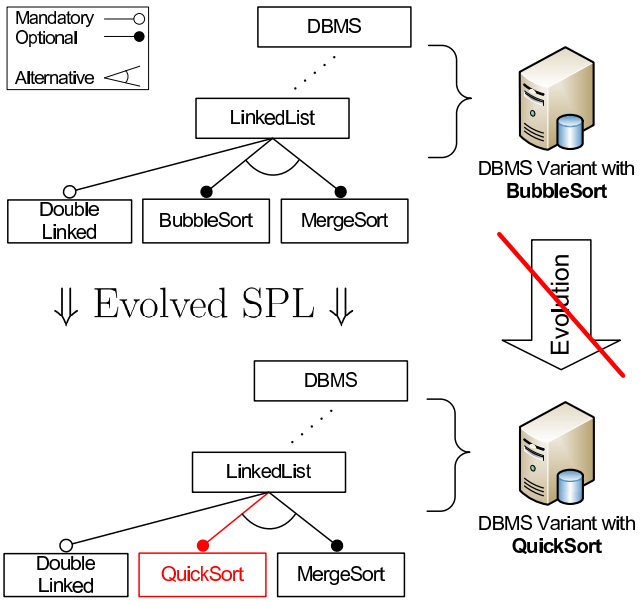

Figure 1: Feature Evolution in a DBMS SPL.

Having recreated the program variants according to its changed SPL, the program variants have to be restarted. The outcome of this update process is that the programs are not available for (at least) short time periods. This downtime is not acceptable for applications that must run 24/7. For that reason we aim at approaches that allow to update running programs according to changes within its SPL.

[1]DBMS – Database Management System.

SINTER'09, August 25, 2009, Amsterdam, The Netherlands.

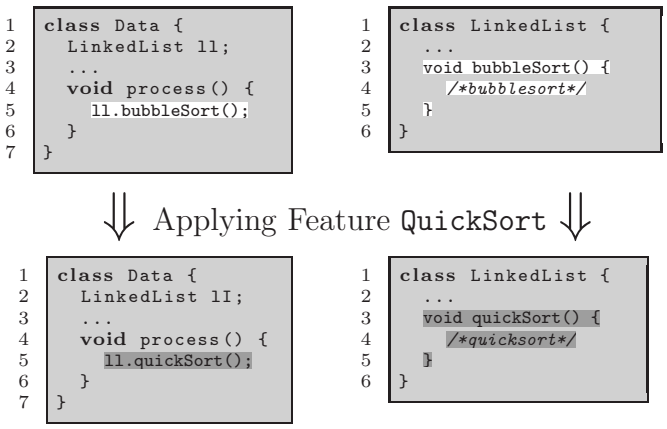

Figure 2: Applying Features.

Runtime program adaptation approaches can be characterized by answering the following two questions: (a) are unanticipated changes allowed (i.e., changes for which the application was not prepared before runtime), and (b) can already executed program parts be changed? We think that runtime program updates because of a changed SPL require approaches for which both questions can be answered with „yes". This is because, it cannot be foreseen in which direction the SPL evolves and what changes has to be applied to the running program. Another reason is that potentially the program part to be changed is already in execution.

Notwithstanding that dynamic languages like Ruby and Smalltalk natively support runtime program updates, we choose Java for two major reasons: (a) Java programs mostly execute faster than programs based on dynamic languages [6], (b) Java is commonly used to implement 24/7 applications. Unfortunately, because Java is a compiled language it does not natively offer powerful instruments for unanticipated runtime program changes even of already executed program parts.

This paper presents an approach that allows to update running programs according to their evolved SPL through dynamically applying features [11, 2]. The approach bases on class replacements and Java HotSwap [5].

2. BACKGROUND

Software Product Lines. An SPL represents a family of related programs [4]. The overall goal of an SPL is to achieve a high reusability of software artifacts which results in reduced time to market and development effort. Software artifacts that are once implemented and tested are deployed in different programs of an SPL which leads to improved maintainability and less errors. Variants of an SPL are distinguishable with respect to its feature selection. By choosing the desired functionality (feature selection), a variant is generated based on a mapping of features to software artifacts. SPLs are typically modeled in a feature model [8]. A feature model describes the provided features of an SPL including relationships and constraints between them. A common representation of a feature model is a feature diagram, as shown in Figure 1. Features are visualized using rectangles and their names and are connected in a hierarchical manner whereas the root represents the SPL itself. Features can be mandatory (filled dot of a connection), optional (empty dot), or only selectable depending on other features (e.g., alternative).

Feature-oriented Programming. *Feature-oriented programming* (FOP) is a new programming paradigm that can be used to implement SPLs [11, 2]. FOP is based on object-oriented programming and extends the modularization capabilities in order to modularize also crosscutting concerns. It considers the features of an SPL as fundamental, modularized implementation units. This is, a feature of an SPL corresponds to a feature module in FOP. Feature modules are physically separated and contain classes as well as class fragments. To derive a variant from an SPL, classes are composed from multiple feature modules that correspond to selected features. In the composition process, a base program is step-wise refined with additional functionality whereas each refinement belongs to a feature module. Depending on the order of the refinement, methods and classes that are defined in one feature module can be extended by another one.

Runtime Program Changes and Java. To understand why runtime program changes in Java are challenging, it is necessary to have a look into the internals of Java's runtime environment – the Java virtual machine (JVM). Basic elements of a JVM are the *heap*, the *method area* and stacks (not considered here). The heap stores the runtime data of all objects, whereas the method area stores all class (type) specific data [10]. Runtime program changes that require to change the schema of already loaded classes, such as adding class fields, removing methods, or changing the superclass, require to modify the data of the heap as well as of the method area and their syncronization. Unfortunately, the JVM does not provide such synchronization mechanisms, i.e., class schema changes at runtime are not possible. However, there is one feature provided by the JVM (in more detail Sun's HotSpot VM) that allows to reimplement method bodies at runtime – called *Java HotSwap* [5].

3. RUNTIME ADAPTATION APPROACH

As stated in the previous section, the JVM does not allow to change the schema of already loaded classes. But, this is elementary for the application of features such as shown in Figure 2 where the schema of class `LinkedList` must be changed in order to remove feature *BubbleSort* and add feature *QuickSort* to the program. To overcome this problem and to apply features to running applications, we developed our own runtime adaptation approach based on class replacements and method body reimplementations via Java HotSwap.

3.1 Class Schema Changes via Class Replacements

The only way to change the schema of an already loaded class is to replace it by a new and updated class. Unfortunately, class replacement in Java is a challenging task. This is because of the strict class loading concept Java enforces. In order to load a class the virtual machine requests the programs class loaders (i.e., the root class loader, the extension class loader, and the application class loader) to load the actual class. The class will be finally bound to the first class loader that is able to load the class. However, loading a new version of a class is only possible when the original class was unloaded. The problem that arises from this fact is, that

a class can only be unloaded when its class loader and all other classes the class loader owns are dereferenced which in most of the cases is equivalent to an application stop.

Beside these restrictions, there are two solution possible that enable runtime program changes through class replacements. The first solution bases on custom class loaders [9]. In order to load a new version of an already loaded class, a customized class loader instance has to be created and method `loadClass()` of this instance has to be called.

```
class LinkedList { ... }
```

⇓ Replacement ⇓

```
class LinkedList_v2 { ... }
```

Figure 3: Class Renaming.

However, the customized class loader approach requires to create a new class loader instance each time a class has to be reloaded. That is, in case of frequent program changes respectively class updates, the program would be polluted by tons of class loader instances that slow down the program and consume memory space. Therefore, we use another class replacement strategy – *class renaming*. The key idea of this solution is to rename the class to be updated and load it under a fresh name (see Figure 3). Because the new class name is not registered in any class loader, the new class version can be loaded without problems.

```
1 class Data {
2   LinkedList ll;
3   ...
4   void process() {
5     ll.bubbleSort();
6   }
7 }
```

⇓ HotSwap ⇓

```
1 class Data {
2   LinkedList ll;
3   ...
4   void process() {
5     LinkedList_v2 llv = new LinkedList_v2();
6     Helper.mapState(ll, llv);
7     llv.quickSort();
8   }
9 }
```

Figure 4: Method Reimplementation using Java HotSwap.

3.2 Caller Update via Java HotSwap

As we described above, our class reloading approach allows to reload an already loaded class even if the class schema has changed. In this first step we only load the updated class into the JVM, i.e., the class is not yet part of program execution. In order to let the class become part of program execution all accesses (calls) to instances of classes that have to be reloaded must be updated. In our example (Figure 2) this requires to update class `Data` which calls reloaded class `LinkedList`.

The update works as depicted in Figure 4. It requires to reimplement all methods of the caller class (here class `Data`) in which the callee (here an instance of class `LinkedList`) is used. The method reimplementation is done using Java HotSwap and consists of three different parts (as shown in the lower part of Figure 4 (Lines 5-7)). First, an instance of the new class version (here class `LinkedList_v2`) has to be created (Line 5). Second, the state of the old callee instance is mapped[2] to the up-to-date instance (Line 6). Third, in order to call new methods, the corresponding method calls are updated (Line 7).

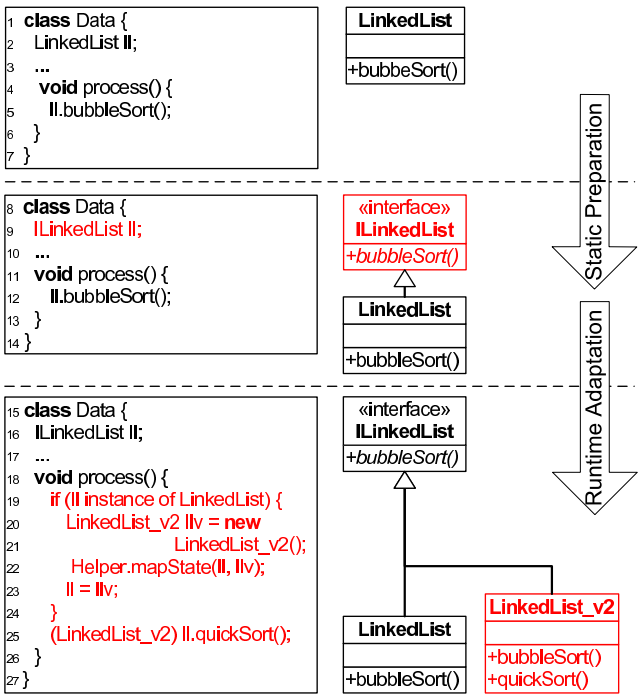

Figure 5: Updating Global References.

We note that the mechanism described above only allows to locally replace instances of the reloaded class, i.e., class member `ll` in class `Data` still remains outdated. In order to also update such global variables, we combine our approach with interfaces (see Figure 5). The static type of class member `ll` within class `Data` is changed (before program start) to interface type `ILinkedList` which enables `ll` for late binding. Thus, it is possible to assign objects different from type `LinkedList` to `ll` during runtime. The application of program changes itself also bases on method reimplementations and works as depicted down to the left of Figure 5. First, an instance of the new class version (`LinkedList_v2`) is created (Line 20 - 21). Second, the state of the old instance, which is still assigned to `ll`, is mapped to the new instance (`llv`) (Line 22). Third, the new instance is assigned to class member `ll`, i.e., the runtime type of `ll` switches from `LinkedList` to `LinkedList_v2` (Line 23). In the final step, the method calls are updated (Line 25). In case the methods that have to be called are not declared within the interface (such as method `quickSort()` which is not de-

[2]Note that a detailed consideration of state mappings is out of the scope of this paper because of their complexity.

clared in interface `ILinkedList`) the updated variable (here `ll`) must be casted to the actual runtime type.

3.3 Feature Modules

As we described in this section, our runtime adaptation approach consists of two different techniques, i.e., class replacements for class schema changes and Java HotSwap used to update program parts that call the classes to be replaced. These two techniques used in combination allow to change nearly each part of a running program. In particular, it allows to apply features to a running program and thus to update it regarding the changes (e.g., bugfixes, added or removed functionality) within its software product line. A feature is organized in a feature module which can contain the following elements: classes, caller updates, and class updates. Whereby, classes build the base and caller updates as well as class updates act as refinements of existing classes. Figure 6 exemplifies how feature modules are organized regarding our DBMS SPL. Each feature module only contains the code fragments that belong to the feature. For instance, feature module *QuickSort* contains a refinement of class `Data` as well as of class `LinkedList`. Applying a feature means to apply all code fragments of its feature module to the running program as described above.

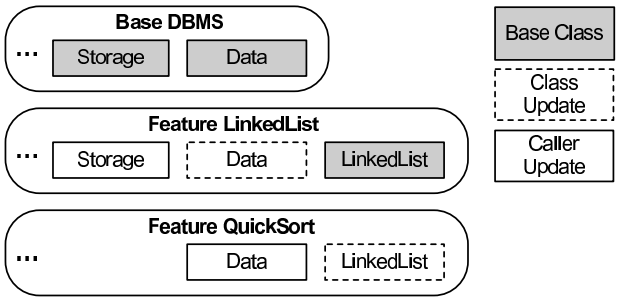

Figure 6: Feature Modules.

4. RELATED WORK

Several approaches exist that aim at runtime program adaptation. In previous work, we developed a runtime program adaptation approach based on object wrappings which was appropriate to change running real world applications but only simulated class schema changes [12]. In CaesarJ [1] and Object Teams [7] software modules can be statically defined and activated or deactivated at runtime. Because it must be statically defined which modules can be activated or deactivated at runtime, the approaches cannot be used to change a program in an unanticipated way. Tools like *AspectWerkz* [3], *Wool* [14], and *JAsCo* [15] allow to deploy aspects at runtime, even in an unanticipated way. But, they do not permit class schema changes. With FeatureC++ Rosenmüller et al. [13] implemented an approach to allow static and dynamic composition of features. However, our approach allows also to change an already instantiated SPL which is currently not supported in FeatureC++.

5. CONCLUSION

In this paper we described how 24/7 programs written in Java can be updated at runtime regarding changes (e.g., bugfixes, added or removed functionality) made within a software product line (SPL) the programs were created from. The presented approach overcomes Java's shortcomings regarding runtime program adaptations. It allows unanticipated program changes (including class schema changes) even of already executed program parts which is necessary to apply features (which are the basic elements of SPLs) to running programs. The approach works on top of Sun's HotSpot VM and bases on class replacements and Java HotSwap.

Acknowledgments

Mario Pukall's work is part of the RAMSES[3] project which is funded by DFG (Project SA 465/31-2). Norbert Siegmund's work is part of the ViERforES[4] project which is founded by BMBF (Project 01IM08003C).

6. REFERENCES

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. *An Overview of CaesarJ*. 2006.

[2] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6), 2004.

[3] J. Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *Proceedings of AOSD'04*.

[4] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

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

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

[7] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *Proceedings of Net.ObjectDays'02*.

[8] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[9] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of OOPSLA'98*.

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

[11] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of ECOOP'97*. Springer Verlag.

[12] M. Pukall, C. Kästner, and G. Saake. Towards Unanticipated Runtime Adaptation of Java Applications. In *Proceedings of APSEC'08*.

[13] M. Rosenmüller, N. Siegmund, S. Apel, and G. Saake. Code Generation to Support Static and Dynamic Composition of Software Product Lines. In *Proceedings of GPCE'08*.

[14] Y. Sato, S. Chiba, and M. Tatsubori. A Selective, Just-in-Time Aspect Weaver. In *Proceedings of GPCE'03*.

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

[3] http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/ramses/
[4] http://vierfores.de