Dynamic Updating of Software Systems Based on Aspects

Susanne Cech Previtali and Thomas R. Gross

Department of Computer Science
ETH Zurich
CH-8092 Zurich/Switzerland

Abstract

Long-running applications such as network services require continuous uptime but also frequent changes to the software. To avoid downtime for software maintenance, applications must be updated at run-time. We describe a system based on the ideas of aspect-oriented programming (AOP) to manage such updates. Join points as defined by AOP establish locations for code modification in a program. We use these join points to guide software updates. Updating a system is a two-step process: the original (old) and new (updated) versions of a software system are compared and a list of update actions and pointcuts is constructed. We present a case-study to evaluate the applicability of this approach.

1. Introduction

Many applications such as file and authentication servers require continuous uptime. Unfortunately such servers also require updates, i.e., changes to the software. The reasons for such changes range from correcting bugs to installing new functionalities. Updating software usually consists of installing the new binaries (among new configuration files), stopping the application, and restarting it. However these steps are problematic in an environment where users expect uninterrupted availability. And since such servers are usually part of complex systems, restarting one of these servers has often consequences for many parts of the overall system — e.g., explicit and implicit caches are invalidated. Ideally, applications can be updated at run-time without the need to terminate the running application.

An approach often used to avoid downtimes of mission-critical applications that rely on constant accessibility is to build a redundant infrastructure with (manual) failover functionality [2]. A primary system is replicated in a backup (stand-by) system. In case of maintenance work on the primary system, the user requests are transparently redirected to the backup system that mimics the operations of the primary system. While the primary system is offline, the application can be updated to the latest version. In many environments, such a solution generates costs for additional hardware (e.g., servers and shared storage systems) and software licenses. Failover may be difficult to configure (and maintain), leading to inconsistencies.

Aspect-oriented programming (AOP) [12] has been proposed as complement to traditional (class-based) object-oriented programming (OOP) to improve modularity of software systems. The basic premise of AOP is that there exist activities that are common to a set of classes, but the code for these activities cannot fit into the inheritance structure of an OOP program. E.g., the code may logically belong to multiple classes. As this so-called *cross-cutting* behavior is "scattered" among classes, the software base is hard to maintain and difficult to evolve. AOP tries to solve this problem by encapsulating the cross-cutting behavior in another kind of module (i.e., *aspects*). Through the separation of cross-cutting as well as normal ("core") behavior, the development and evolution of the entire system become easier.

An aspect consists of the specification of the place where it should be installed and of the description of the additional code. A *join point* is a point in the program where during execution some additional action could be taken. Join points can be, e.g., method invocation or termination, field accesses or assignments, or non-local control transfers like *throws*. *Pointcuts* specify signature patterns that select the join points.

Aspects must be merged with the core classes to produce the final (complete) application, and this step is usually referred to as *weaving*. Different AOP systems position weaving at different times in the compile-load-execute sequence, i.e., some systems invoke the weaver at compile time, others at load time, and some systems allow weaving at run-time. AspectJ [11], a widely used AOP system, supports static

The work presented in this paper was partially supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

(load-time or earlier) weaving. PROSE [15, 18, 19] is a dynamic AOP system that allows the user to weave and unweave aspects at run-time. In PROSE, aspects and crosscuts are defined by subclassing respective Java classes.

AOP provides a model to modify a software system after it has been released and installed. We want to exploit this model (and the infrastructure that has been developed for AOP systems) for software updates. We consider two contributions of AOP as important for an evolution system: First, weaving allows an application to be modified at run-time. Second, pointcuts provide a mechanism to specify where the code modifications must be applied.

The focus of traditional AOP is to identify cross-cutting behavior and appropriate code that can be added to realize the cross-cutting behavior. In this paper, we investigate how a dynamic AOP system can handle the evolution of core classes in an object-oriented programming language such as Java. Examples are based on Java, but the concepts presented in this paper are not tied to this language and can be applied in other contexts as well. The run-time evolution system must support the dynamic update of the running application. *Dynamic updating* refers to the process of bringing a running application to the latest version without stopping its execution; this requires the addition, removal, and modification of code to evolve the application from one version to the next version.

Before we explain in detail how an AOP system can be used to realize *general* updates in a running application, we first review related work in Section 2. Note that we use AOP as a tool to accomplish software updates. In this paper we do not explore how to structure the software source changes in terms of aspects; instead we want to explore to what extend the AOP model provides a framework to install software changes. There may be some changes to a software that go beyond what can be expressed as changes at join points. However, our (very limited) initial investigation shows that there are also many changes that fit this model. And in this case, AOP may provide a simple path to realize dynamic software updates of applications, without the need to take down the application.

2. Related work

Several software-based approaches exist on how to handle updating an application. We first discuss systems that exploit the dynamic linking capability to realize updating at run-time. Then we present a system adapting the application at load-time. Although this system does not perform run-time software updating, it relates to our approach in the way adaptations to the original software are specified.

2.1. Updating with dynamic linking

Neamtiu et al. [8, 14] propose a dynamic software updating (DSU) system for C. The compiler combines code rewriting and a suite of static analyses. Initially, the first version of a program is compiled to be updatable. The compiler performs static analysis to aid the programmer in finding update points.

To prepare for function evolution, a global function pointer is created for each function. Functions are renamed to include version information. Wrapper functions call then the appropriate version of the function. Type evolution is handled by wrapping the original type and adding a version number and fixed-size extra space to prepare for eventual growth of the type over time. For each field access, the compiler inserts code to return the underlying representation.

When it is necessary to move to a new file version, the developer feeds the old and the new version into a patch generator. The patch generator identifies the modified definitions of global variables, functions, and types. For each changed definition, a type transformer function is generated to convert existing objects to conform to the new type definition. The developer completes the functions by providing explicit conversions. Additionally, she writes state transformer functions to convert global state. The DSU compiler introduces initialization code, which is executed at update-time to change the function indirection pointers, install the type transformer and state transformer functions. Unchanged definitions are declared `extern` so that the resulting C file can be compiled with a standard GCC compiler.

The patch file is dynamically linked into the running application. While this system contains many interesting ideas, C is not an object-oriented language and issues such as inheritance and polymorphism do not exist. The object evolution is limited by the fixed amount of extra space reserved in the initial version. To overcome this problem, Neamtiu et al. suggest using indirection to the type definitions for the cost of an extra dereference per access [14].

Hjálmtysson and Gray [9] present an approach based on dynamic linking for C++. Abstract proxy classes define the interface that must remain constant over all versions. Each new version is another subtype of such a proxy class. Although new versions can add methods other than those defined in the abstract proxy class, these methods cannot be called from other classes. Therefore the approach is limited to changes that preserve the initial interface of the class. Object evolution is not supported; old objects exist as long as they are used. New objects are created always with the latest versions. Furthermore, inheritance is restricted because of the abstract interface classes.

A similar approach for Java based on wrapper classes

is discussed by Orso et al. [16]. Addition, deletion, and modification of classes is supported. Newly added classes are lazily loaded on the first reference. When classes are removed, the garbage collector deallocates existing objects of the removed classes. Class modifications are handled via proxy classes; therefore the interface cannot be changed. The run-time system is not aware of the versioning support.

Our solution does not limit the growth of the objects. We do not use wrappers or proxies for accessing types and functions as we rely on the run-time system to directly adapt the run-time structures of classes and objects. Therefore, the system we describe does not impose a performance overhead during execution of a program. Finally, changes to the visible interface of an application are allowed.

2.2. Load time adaptation

Binary Component Adaptation (BCA) [10] allows the modification of Java classes while they are loaded. This work targets the problem of integration and evolution of third-party components. To integrate such a component, the developer provides a *delta class* describing the differences between the original class file and the desired application-specific variant. At load time, the internal structure of that component is modified to reflect the adaptation. With this technique, it is possible to express binary incompatible changes [6].

The load-time approach does not target the update of running applications. The delta classes must be programmed manually by the developer. With the delta classes only the adaptation of one specific version to another version or component may be accomplished. Continuous evolution or adaptation of an application is difficult as the delta classes have to be programmed manually.

We address the continuous evolution of applications at run-time. Our approach provides for an automatic deduction of the differences between the versions. Our work is influenced in particular by the idea of how to describe the difference between two versions.

3. An approach to dynamic updating

In the following we describe the architecture for our system to update an application at run-time without the need to interrupt the application's execution. Given the classes of both versions of the application, there is a delta function from the old version to the new version. Consequently, applying such a delta function to the old version results in the new version (see Figure 1). We propose to express this delta function with aspects, which can be woven into the application at run-time.

Before describing the architecture of our system, we list our assumptions and the required properties of the system.

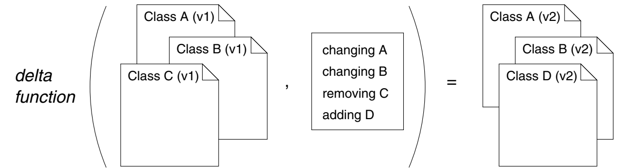


Figure 1. Delta function

(1) The updating mechanism should have the property that after updating the application, the updated version has the same behavior as the new version. In other words, the updater produces exactly the program that the developer wanted to see installed.

(2) Furthermore, any full version of the application should be available to the customers as new customers should not be forced to start with the very first version and then apply all available updates.

(3) The developer should not need to be aware about the updating mechanism when programming and evolving the application code. This means she should not be forced to exclude certain language features, include specific interfaces, or use a particular platform.

Property (3) is hard to realize. Consider the addition of fields to class definitions. When a field is added to the application, it must be initialized not to break any invariants imposed on the field's value. To initialize new fields correctly, the programmer may supply an "update constructor", similar to the type transformer functions as suggested by Neamtiu et al. [14]. Involving the programmer in the updating process by providing "glue code" or excluding the addition of new fields conflicts with property (3).

We want to point out a reasonable limitation of the system. The updating mechanism does not take into account other components that use the evolving application. Consider a client-server system where the server runs continuously and is updated at run-time. Clients evolve with the server but are not subject to run-time updating. When a new version of the server is installed, clients may not yet have been updated and may connect to the updated server. As the visible interface of the server might have changed, the client may experience undefined behavior.

The architecture of the system to manage software evolution is visualized in Figure 2. The compiler computes the difference of two versions of a given component guided by a set of updating rules. As an example, consider the renaming of a method. An updating rule may state that if two method bodies and the signatures are the same and only the two names differ, the method was renamed.

The programmer may be involved in the updating only if these rules are not enough to reconcile the two versions. A static analysis of the two versions determines whether the

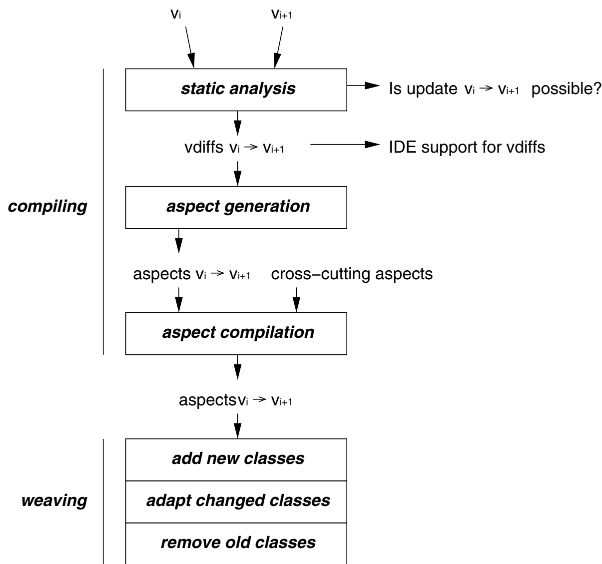


Figure 2. System architecture

system can be updated using aspects; if it is not possible to update the old version, the programmer may be able to modify the new version to be compliant with the updating rules (or, if this is acceptable, fall back on a conventional software installation that restarts the service). Modifications that cannot be handled by this evolution system are, e.g., modifications of third-party libraries.

The static analysis produces version differences (called *vdiffs*) which describe the delta function from the old to the new version. The *vdiffs* serve as input to the aspect generator as well as input for a software development tool (e.g., an IDE such as Eclipse) to visualize the differences to the programmer. The aspect generator determines an ordering for updating class and method definitions. For new attributes, the aspect generator provides a default initialization, which must be completed by the programmer.

The programmer may additionally develop cross-cutting aspects. For now, we do not consider the evolution of the cross-cutting aspects themselves. The updating aspects and the cross-cutting aspects can be compiled and shipped to the customer.

The aspects are input into the run-time system. In our case, the run-time system is based on an aspect-enabled virtual machine (VM) such that we can control and manipulate the execution of the application. This VM must guarantee that after an update the application is left in an updated *and* consistent state. The ordering deduced by the aspect generator enforces that the state of the application will be consistent. The PROSE system provides a proof of concept that such an aspect-enabled JVM can be implemented but other VMs may also serve in this role.

Updates cannot happen at any time. The VM needs to be aware when an update cannot be safely applied; in addition, there may be outside constraints such as a desire (or requirement) to avoid updates during high load periods.

At the next safe evolution point, the aspects are woven into the running application. A safe evolution point implies that a join point is reached and the join point is not part of an active method call. The run-time structures of classes and methods are modified. Proper ordering of the update actions is important to guarantee consistency. E.g., a method definition must not be deleted as long as there are calls to this method. Existing objects must be transformed to conform to the updated class definition.

We identify two main challenges to be explored for such an evolution system. (1) What are the criteria to decide if update is possible? (2) Given a positive decision for updating, what are the criteria that the update is valid?

The following describes the compilation as well as the weaving phase in more detail.

3.1. Generating vdiffs and aspects

The compiler analyzes two versions of a given class together with version differences (*vdiffs*) computed from these versions. Character-based tools (e.g., `diff`) do not take into account structural and syntactical information of the source code. Alternatively, syntactical information can be stored in an abstract syntax tree (AST) with the differences being operations on the graph [13]. As we want to integrate the *vdiffs* into a software development tool, we need to preserve the programmer's view on the source code. Maletic and Collard [13] propose *meta-differencing* using an XML representation of the source code annotated with parts of the AST that provides this mapping.

The updating aspects are generated from the *vdiffs*. This process is relatively straight-forward for binary compatible changes. Binary compatible changes [6] should aid developers in evolving their libraries which are used by clients: "A change to a type is binary compatible with preexisting binaries if preexisting binaries that previously linked without error will continue to link without error." ([6], page 339). This basically states that the visible interface must remain the same. The *visible interface* contains all members that are accessible from outside the defining class, i.e., the access levels public, protected, and package in Java.

Although described for components of different origin, we can apply the rules for an updating system as well: Adding new fields, methods, or constructors to a class can be done without breaking binary compatibility — an additional method does not have any influence to existing code.

Modifications that result in a program that is not binary compatible break the visible interface of classes. Other

dynamic updating systems exclude modifications regarding the public members [9, 16]. This restriction limits the evolution of the application and reduces the expressiveness of the programmer. In Section 4 we show that binary incompatible changes happen frequently, e.g., public methods are removed or the access level of a method is changed from public to private. Changes that destroy binary compatibility are more difficult to handle. The updating system needs to determine an ordering for the correct weaving.

3.2. Algorithms

In the following, we describe the algorithms for the computation of version differences, the correct weaving order and the deduction of aspects. First, the version differences are extracted. Second, we determine the correct ordering and grouping of the updates. In the last step, the aspects can be deduced based on the version differences and the correct ordering.

Determine the version differences Given all classes of two versions of an application, we construct three subsets: (1) the intersection of the classes of version 1 and version 2 gives the set of the enduring classes, (2) the classes of version 1 minus the enduring classes provides the classes that must be removed from the new version, and (3) the classes of version 2 minus the enduring classes result in the classes to add to the new version. Figure 3 illustrates the subsets.

Note that the elements of the set containing the enduring classes are pairs of (version1, version2) classes. The elements of the class pairs have to be compared to distinguish between modified and unchanged classes. Two classes are equal if their headers and all their members (i.e. fields and methods) are equal. Class members are parsed and paired as described above. Member pairs are compared according to their properties — e.g., two fields are equal if the names, types, access and property flags are equal. The version differences are expressed as a set of differences for each class including its methods and fields. See Algorithm 1 in the Appendix.

Impose the correct ordering and grouping For all the classes in the new version of the application, we construct a dependence graph. There is an edge from class A to B if A depends on a class B, i.e., (a) A calls a method of B, (b) A accesses a field of B, or (c) A extends B. All classes in a strongly-connected component of this dependence graph must be updated atomically. These classes depend on each other and consistency of the updating can only be guaranteed by generating one aspect containing these changes. The correct ordering can be obtained by traversing the strongly-connected components in breadth-first order. The algorithm is illustrated in Figure 4 and listed in Algorithm 2.

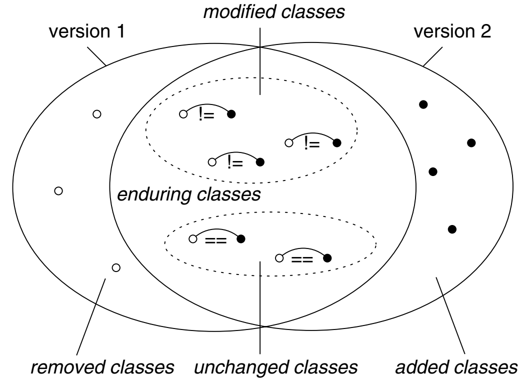


Figure 3. Set operations on two versions

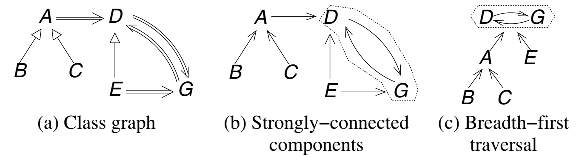


Figure 4. Correct ordering

Construct the aspects For each strongly-connected component an aspect is generated taking into account the version differences in the order as imposed by the breadth-first traversal of the strongly-connected components. No aspects are generated for *new classes* as they will be loaded when they are used. A deletion entry is included for *removed classes*.

Example We provide an example to illustrate these three steps. Figure 5 shows two versions of an application with their respective classes and methods. Class E is removed from the new version, class D is added to the new version, and classes A, B, and C have been modified to adapt to this new situation. In the first step, the version differences are generated and in the second step, the correct ordering and grouping for all classes is imposed. The last step consists of identifying the aspects. All three steps are shown in Figure 6.

3.3. Weaving the aspects

The aspect-enabled VM needs to deal with three different kinds of changes; classes may be added, removed, or

Version 1	Version 2
E : e() { ... }	—
A : a() { e.e() c.c() }	A : a() { c.c() }
B : b() { c.c() }	B : b1() { c.c() } b2() { ... }
C : c() { b.b() }	C : c() { b.b2() d.d() }
—	D : d() { ... }

Figure 5. Example: Two application versions

I. Vdiffs

- addedClasses = {D}
- modifiedClasses = {A, B, C} with
  - A.modifiedMethods = { a() }
  - B.modifiedMethods = { b() }
  - B.addedMethods = { b2() }
  - C.modifiedMethods = { c() }
- removedClasses = {E}

II. Ordering and grouping

1. add = {D, B.b2()}
2. modify = {A, {B, C}}
3. remove = {E}

III. Aspects

- aspect1 { B.b2() {...} }
- aspect2 { rename B.b() to B.b1();
  replace C.c() with { b.b2(); d.d(); } }
- aspect3 { replace A.a() with { c.c() } }
- aspect4 { remove E }

Figure 6. Example cont.: Algorithms

modified. A newly-added class will be loaded the first time it is used. Its usage (i.e., another class creates an object of this class) will be captured by an aspect updating the client class to the next version.

To remove classes, we must ensure that the class is no longer used by any other class in the application. Clients of the class to be removed have been modified in the new versions; this is reflected in the respective aspects. Once all clients have been updated, the objects of a class to be deleted can been removed by the garbage collector and then the class itself can be removed.

Updating modified classes is challenging as the run-time structures of the classes must be modified. Methods may be added, modified, and deleted requiring the adaptation of the virtual method tables. A method may have to be recompiled even though it did not change itself but a method that it invoked. Existing objects must be updated to conform to changed class definition. In the following, we describe the evolution of objects and methods.

Object evolution. There are two issues to be dealt with: The first concerns when objects can be updated. The second issue considers the representation of objects such that an update is possible at all.

Instead of migrating objects during update time, Neamtiu et al. [14] suggest a lazy approach. The objects are updated on a per-access-basis during program execution. This approach reduces the update time as not all objects need to be found and updated, but the approach also makes each access more costly.

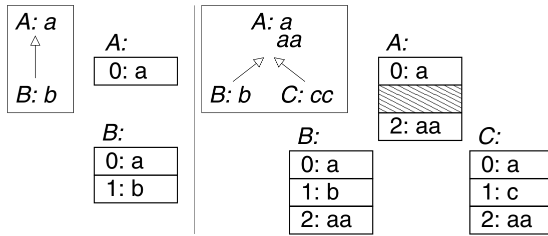
In object-oriented languages such as Java or C#, we can make use of a copying garbage collector during update time. The copying garbage collector traverses the object graph and copies the live objects to a new location. While traversing the object graph, the garbage collector checks for modified and removed class definitions. Objects with new class definitions are updated before moving; objects of removed classes are left behind. Note that this system does not preclude the use of a non-copying garbage collector at other times. For object representation we need to consider that (1) objects tend to grow over time and that (2) existing methods are compiled such that they refer to a field in terms of an offset of the object's address.

Neamtiu et al. [14] suggest a simple padding scheme. The wrapper type allocates a fixed space for future space requests by putting the type definition in a `union`. The new types simply overwrite the old data in the same storage. Although object transformations are simple to handle, memory space is wasted and evolution is limited by the fixed size of the padding. The cache locality of the data is changed.

We propose two different schemes — the first wastes some space, but does not require existing methods to be recompiled; the second does not waste space, but existing methods must be recompiled.

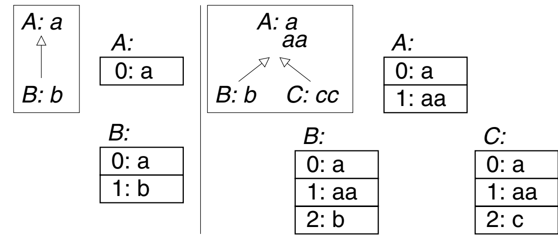
Figure 7 shows an inheritance hierarchy with two classes A and B with B extending A. A and B define the attributes a and b. In the next version, A adds attribute aa; class C extends class A and introduces a new attribute cc.

The first scheme is an incremental coloring algorithm based on a graph coloring algorithm to build virtual method tables (VMT) for object-oriented languages with multiple inheritance such that the methods in each subclass have the same index in the VMT [4]. As Java does not support multiple inheritance, we do not have to create conflict graphs and conflict tables. Instead we can simply traverse the class graph and assign the colors to each attribute. The colors describe the offsets of the fields relative to the starting address



(a) Version 1 (b) Version 2

Figure 7. Incremental coloring algorithm



(a) Version 1 (b) Version 2

Figure 8. Recompilation algorithm

of the object. Furthermore, our algorithm is incremental with respect to the different versions. For each new version, the class graph must be traversed again, but colors of the earlier versions remain thereby guaranteeing that the old field offsets are not modified.

For each class, we store the colors of its attributes and the respective colors of its subclasses. Initially no colors are assigned. The class graph is traversed in a depth-first fashion. For each class, the colors are assigned to all attributes, such that the minimal color of the known colors is selected.

Figure 7 illustrates the algorithm. The algorithm works incrementally such that assigned colors remain unchanged over the versions and only for newly added attributes the color is assigned. Is an attribute deleted, its color can be reused. Therefore we can guarantee that the field offsets remain the same. There is still some space wasted, but we do not impose a limit to the growth of types (as opposed to Neamtiu et al. [14]). Performance of the updating itself is not influenced, as the traversal of the class graph is done at compile time.

As an alternative, we can use the ability of many VMs [1, 5, 17] to recompile and replace methods at run-time for object evolution. This second scheme neither poses limitations on the growth of the objects nor wastes space as we allow the object layout to be changed for the price that methods must be recompiled. When new attributes are added to a class, the compiler reassigns all offsets — this corresponds to the color assignment in the very first versions. Figure 8 shows this assignment.

As a consequence, we need to (1) move fields of existing objects to their new place and (2) recompile existing methods accessing the old fields. The static analysis annotates methods with the attributes they access. At run-time, the methods involved in object layout changes are recompiled to reflect the changes in the field offsets of the objects.

Method evolution. If new methods are added or removed, the virtual method tables of the respective classes must be updated. Addition of members may be handled by static crosscutting as proposed by Aspect/J[11]. Java supports multiple inheritance only on the interface level, and therefore we can use the coloring algorithm as proposed by Dixon et al. [4] on an incremental basis as shown above.

4. Evolution study

We developed a tool to analyze version differences of Java class files. Based on the ASM Java bytecode manipulation framework [3], we implemented the algorithm to construct the version differences (see Algorithm 1). Given the version differences report, we studied how an application evolves over several versions extracting different kinds of changes and their occurrence. With this case study we try to answer the following questions:

(1) *Kinds of changes:* What are common changes, also with respect to the access levels? How does the visible interface change?

A practical dynamic updating system must handle common changes efficiently. This tool enables us to study different applications and classify possible changes according to their occurrence. The visible interface is changed when members are added to or removed from a class. Other approaches (see Section 2) do not handle the changes of the visible interface. If changes to the visible interface of an application are common, a dynamic updating system must support updates to the visible interface as well.

(2) *Applicability of a aspect/join points model:* Can the common changes be handled by using a join point model and modeling the changes as aspects?

We studied the evolution of a small client-server application written in Java following six major versions. The *ETH Lecture Communicator* (lectcomm) is a tool developed at ETH Zurich [7, 20] to be used during lectures to improve interaction between the lecturer and the students. The system is based on a network infrastructure (e.g., a Wireless LAN) and portable computers brought to class. Students connect via a Java applet to the server operated by the instructor. The tool enables the instructor to perform in-class online-

assessments and facilitates organized instant communication for big classes. Students themselves can ask questions, which are rated by other students, so the instructor can answer questions with a high priority first.

Table 1 gives an overview of the application size in terms of numbers of members distinguishing between the access levels of the members. While the number of protected member remains approximately the same over all versions, the numbers of public, package, and private members augment.

Table 1. Member evolution by access level

	0.5	0.6	0.6.1	0.6.2	0.7	1.0rc1
public	578	696	712	798	825	836
protected	52	57	57	65	67	67
package	276	316	324	379	400	403
private	410	482	490	552	597	608
# Members	1316	1551	1583	1794	1889	1914
# Classes	182	219	224	248	260	263

We show the most common changes in Table 2. On a class level, the inheritance hierarchy is changed by changing the superclass of a type of by implementing other interfaces. For fields, the most common change regards the access level. Sometimes the type and rarely the initialization value are adapted. Methods bodies are changes most frequently. Access levels vary seldom, and exceptions are rarely modified.

Table 2. Kinds of changes

	0.5 → 0.6	0.6 → 0.6.1	0.6.1 → 0.6.2	0.6.2 → 0.7	0.7 → 1.0rc1
Class header					
SuperName	2	0	2	3	1
Interfaces	12	1	3	4	0
Fields					
Access	11	1	0	0	2
Type	3	0	0	0	0
Value	1	1	3	1	3
Methods					
Access	4	1	6	1	0
Signature	0	0	0	0	0
Exceptions	1	1	0	0	1
Body	106	32	62	35	133

For a more detailed analysis of the changes of the method bodies, we count the number of changes within a method. One change refers to the change to one bytecode instruction or to a consecutive list of changes to bytecode instructions. Figure 9 shows a histogram for all versions plotting the number of methods modifying a given percentage of the method bodies. Most changes influence 10–20% of the method bodies.

Figure 10 aims to answer the question about the changes to the visible interface of the application. It visualizes the removed and added members which have the access levels

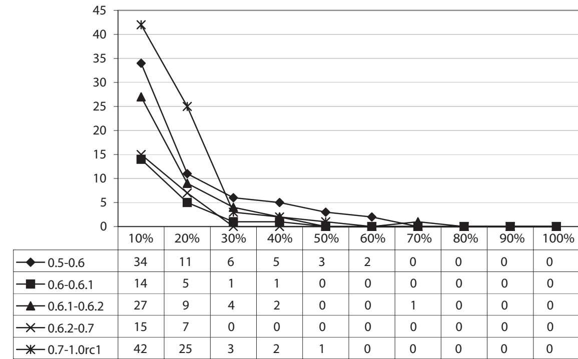


Figure 9. Changes to the method bodies

public, protected, and package. We can see that the visible interface is always changed between the versions. In this particular application more members are added then removed.

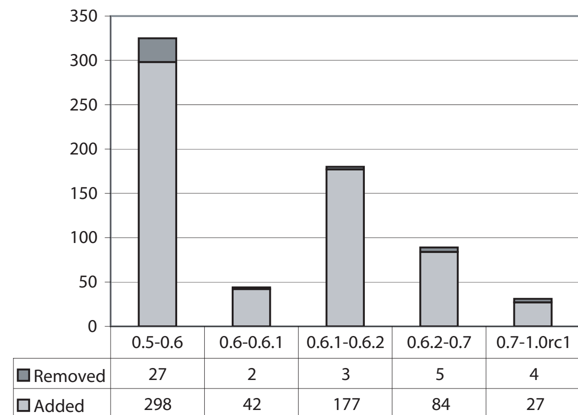


Figure 10. Visible interface modifications

Class evolution is shown in Table 3. The majority of classes are not modified (around 60% from version 0.5 to version 0.6, and around 80% from version 0.6 to the version 1.0rc1). 12–40% of the classes are modified, very few removed. Initially 20% of the classes are modified, then less then 10%. Looking at the evolution of the fields (see Table 3, the the majority of the fields remain the same. In every version fields are added, but rarely removed or modified. Method evolution (in Table 3) follows the same trend, but methods are changed more frequently.

For our dynamic updating approach we can fix the join points not only at the method boundaries, but even at basic block boundaries. Working with smaller updates influences the weaving process. Join points on a basic block level allow fine-grained changes to methods. With an aspect model, version updates may be handled directly and effi-

Table 3. Evolution of classes, fields, and methods

	0.5 → 0.6		0.6 → 0.6.1		0.6.1 → 0.6.2		0.6.2 → 0.7		0.7 → 1.0rc1	
	no.	%	no.	%	no.	%	no.	%	no.	%
Class evolution										
Unchanged	112	61.5%	191	87.2%	186	83%	219	88.3%	185	71.1%
Modified	69	37.9%	27	12.3%	38	17%	29	11.7%	75	28.8%
Added	38	20.9%	6	2.7%	24	10.7%	12	4.8%	3	1.2%
Removed	1	0.5%	1	0.5%	0	0.0%	0	0.0%	0	0.0%
Field evolution										
Unchanged	467	92.1%	587	99.7%	594	98.8%	673	99.1%	703	98.7%
Changed	13	2.6%	2	0.3%	3	0.5%	1	0.1%	5	0.7%
Added	109	21.5%	12	2.0%	82	13.6%	38	5.6%	15	2.1%
Removed	27	5.3%	0	0.0%	4	0.7%	5	0.7%	4	0.6%
Method evolution										
Unchanged	639	79.0%	917	95.3%	901	91.8%	1070	96.0%	1038	88.2%
Changed	106	13.1%	32	3.3%	65	6.6%	35	3.1%	133	11.3%
Added	217	26.8%	33	3.4%	149	15.2%	72	6.5%	20	1.7%
Removed	64	7.9%	13	1.4%	16	1.6%	10	0.9%	6	0.5%

ciently without adding indirections for every version. Other approaches mentioned in Section 2 continue to add layers of indirections. Considering the evolution over several versions and years, the performance of these applications degrades. Once the aspect-enabled VM has woven the aspects into the run-time system, the performance of the updated version should approach the performance of the new version of the application.

Although we publish only the analysis of a single application, the results provide an idea about the number and types of changes. The analysis of applications is important for this work to determine the kinds of changes that happen frequently and that therefore must be supported efficiently.

5. Concluding remarks

We present an approach to support software evolution at run-time based on the ideas of aspect-oriented programming. The compiler analyzes the classes of two complete versions of an application to generate aspects automatically from the version differences. Aspects allow to specify version differences in a fine-grained way. We rely on the aspect-enabled VM to include the aspects at run-time and to adapt run-time structures of classes and methods. Objects may be upgraded to conform to the new class definition with a copying garbage collector. We discuss two schemes to adapt existing objects to the change of class definition. We do not need wrappers or indirections to access the classes and methods and thereby avoid performance overheads.

The two core concepts of AOP, i.e., the pointcut definition to capture points in the program execution and the weaving technology to merge aspects into a given application at run-time, can support the continuous evolution of long-running applications. Further empirical studies are necessary to explore when this approach can be applied in practice. Our initial investigation is encouraging but far from conclusive. Furthermore, an architecture for a dynamic updating system as described here may provide additional opportunities to support the software evolution. Since the old and new software versions must be compared, it may be possible to provide the maintainer or installer with summaries of how the software is evolving. Such summaries may help later to understand how the system was changed.

Acknowledgments

We thank Oliver Trachsel for the numerous productive discussions and Christoph Angerer, Nicholas Matsakis, and the anonymous reviewers for their helpful comments and suggestions.

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. *SIGPLAN Not.*, 35(10):47–65, 2000.

[2] M. R. Barber. Increased server availability and flexibility through failover capability. In *LISA*, pages 89–98. USENIX, 1997.

[3] E. Bruneton. ASM. Available at http://asm.objectweb.org/.

[4] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *OOPSLA*, pages 211–214, 1989.

[5] S. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 241–252, Washington, DC, USA, 2003. IEEE Computer Society.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java language specification, Third edition*. Addison-Wesley Professional, 2005.

[7] T. Gross, L. Szekrenyes, and C. Tuduce. Increasing student participation in a networked classroom. In *Proc. of Frontiers in Education*, Boulder, CO, Nov 2003.

[8] M. Hicks and S. Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, 2005.

[9] G. Hjalmtysson and R. S. Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proceedings of the 1998 USENIX Technical Conference (USENIX '98)*, New Orleans, Louisiana, May 1998.

[10] R. Keller and U. Hölzle. Binary component adaptation. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 307–329, London, UK, 1998. Springer-Verlag.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[12] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97-Object-Oriented Programming, 11th European Conference*, volume 1241, pages 220–242. Springer, 1997.

[13] J. I. Maletic and M. L. Collard. Supporting source code difference analysis. In *ICSM*, pages 210–219. IEEE Computer Society, 2004.

[14] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2006.

[15] A. Nicoara and G. Alonso. Dynamic AOP with PROSE. In *Proc. of International Workshop on Adaptive and Self-Managing Enterprise Applications (A SMEA'05) in conjunction with CAISE'05, Porto, Portugal, June*, 2005.

[16] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 649–658, Montreal, Canada, October 2002.

[17] M. Paleczny, C. A. Vick, and C. Click. The Java HotSpot server compiler. In *Java Virtual Machine Research and Technology Symposium*. USENIX, 2001.

[18] A. Popovici, G. Alonso, and T. Gross. Just in time aspects: Efficient dynamic weaving for Java. In *2nd Intl Conf. Aspect-Oriented Software Development*, pages 100–109, Boston, MA, March 2003. Springer.

[19] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *1st Intl Conf. Aspect-Oriented Software Development*, pages 141–147, Enschede, NL, April 2002.

[20] O. Trachsel. ETH Lecture Communicator. ETH Zurich, available at http://lectcomm.sourceforge.net/.

Appendix

Algorithm 1 The version differences

```
1: /* Construct subsets of classes: */
2: enduringClasses ← classesV1 ∩ classesV2
3: removedClasses ← classesV1 \ enduringClasses
4: addedClasses ← classesV2 \ enduringClasses
5: for all class pairs (cV1, cV2) ∈ enduringClasses do
6:    /* check class pair: version, access, name, signature, superclass, interfaces */
7:    hdiff ← cV1.compare (cV2)
8:    /* Construct subsets of members: */
9:    enduringMembers ← cV1.members ∩ cV2.members
10:   removedMembers ← cV1.members \ enduringMembers
11:   addedMembers ← cV2.members \ enduringMembers
12:   for all member pairs (mV1, mV2) ∈ enduringMembers do
13:      /* check member pair: */
14:      /* for fields: access, name, type, signature, value */
15:      /* for methods: access, name, parameters, return value, signature, exceptions, body */
16:      mdiff ← mV1.compare (mV2)
17:      if mdiff then
18:         changedMembers.add (memberPair (mV1, mV2))
19:      else
20:         unchangedMembers.add (mV1)
21:      end if
22:   end for
23:   /* Has class changed? */
24:   vdiff ← hdiff or mdiff
25:   if vdiff then
26:      changedClasses.add (classPair (cV1, cV2))
27:   else
28:      unchangedClasses.add (cV1)
29:   end if
30: end for
```

Algorithm 2 The correct weaving ordering

```
1: for all cm ∈ new classes and new members of existing classes do
2:    cm.order ← MIN
3: end for
4: for all cm ∈ removed classes and removed members of existing classes do
5:    cm.order ← MAX
6: end for
7: for all classes in the new version do
8:    construct a dependence graph
9: end for
10: for all c ∈ strongly-connected components do
11:    c.order ← level
12: end for
```