Technical Report GIT-CC-02-24, April 2002

A Technique for Dynamic Updating of Java Software*

Alessandro Orso, Anup Rao, and Mary Jean Harrold
College of Computing
Georgia Institute of Technology
801 Atlantic Drive
Atlanta, GA 30332
{orso,prime,harrold}@cc.gatech.edu

Abstract

During maintenance, systems are updated to correct faults, improve functionality, and adapt the software to changes in its execution environment. The typical software-update process consists of stopping the system to be updated, performing the actual update of the code, and restarting the system. For systems such as banking and telecommunication software, however, the cost of downtime can be prohibitive. The situation is even worse for systems such as air-traffic controllers and life-support software, for which a shut-down is in general not an option. In those cases, the use of some form of on-the-fly program modification is required. In this paper, we propose a new technique for dynamic updating of Java software. Our technique is based on the use of proxy classes and does not require any support from the runtime system. The technique allows for updating a running Java program by substituting, adding, and deleting classes. We also present Dusc (Dynamic Updating through Swapping of Classes), a tool that we developed and that implements our technique. Finally, we describe an empirical study that we performed to validate the technique on a real Java subject. The results of the study show that our technique can be effectively applied to Java software with only little overhead in both execution time and program size.

keywords: Data dependences, data flow, static analysis, pointers

1 Introduction

In the software maintenance phase, programs are updated to correct faults, improve functionality, and adapt the software to changes in its execution environment. The typical software-update process consists of stopping the system to be updated, performing the actual update of the code, and restarting the system. Many applications, however, must run continuously and have maximum downtime requirements on the order of a few minutes per year [16]. For example, banking and telecommunication software systems have a prohibitive downtime cost. The situation is even worse for systems such as air-traffic controllers and life-support software, for which the interruption of the service is in general not an option. Furthermore, the number of application domains in which systems should be able to deliver continuous reliable service during update is growing, and dynamic software updating is thus becoming an increasingly important issue nowadays.

Dynamic software updating is the task of updating parts of a program without having to terminate its execution. Dynamic software updating can be performed in several different ways, depending on the specific context considered. In particular, we can distinguish between hardware- and software-based dynamic updating techniques. *Hardware-based dynamic updating techniques* are based on hardware redundancy, are fairly expensive, and target specific contexts (e.g., space mission software). *Software-based dynamic updating techniques*, on the other hand, require no hardware support, and are thus more generally applicable. We can further distinguish software-based techniques based on the degree of support that they require from

*This work was supported in part by a grant from Boeing Aerospace Corporation to Georgia Tech, by National Science Foundation awards CCR-9988294, CCR-0096321, and EIA-0196145 to Georgia Tech, and by the State of Georgia to Georgia Tech under the Yamacraw Mission.

the runtime system. In the particular case of Java, techniques that rely on the runtime system require a customized version of the Java Virtual Machine (JVM [12]) to be applicable.

In this paper, we present a new software-based technique for dynamic updating of Java software that is completely defined at the program level. Therefore, our technique requires no support from the runtime system and can be generally applied to any Java application. Our technique operates by first modifying the application to enable its dynamic updating[1] (through class renaming and code rewriting) and then performing hot-swapping of classes (i.e., abstract data types and their instances) at runtime, when a new version of one or more class(es) is available. Under some assumptions, our technique permits generic updates of Java applications and has the following characteristics/properties:

Semantics preservation. The swapping-enabled application has the same behavior as the original application. Also, after an update, the updated application has the same behavior as if it were built from scratch using the updated set of classes (rather than being dynamically updated).

Atomicity of updates. Updates involving more than one class are performed atomically, so avoiding problems related to inconsistent versions of different parts of the program being present in the code at the same time. In addition, the update is performed so that (1) all instances of an updated class are atomically migrated from the old to the new version of the class, (2) no instances of the old classes are executing after the update is completed, and (3) all instances of the older version are eventually destroyed (i.e., garbage collected).

Minimal or no human intervention required. Given a Java application, our technique generates the corresponding swapping-enabled application in a fully automated way. Given a set of updated classes, our technique also generates the updating code and performs the update of the application in a fully automated way. The only case in which human intervention is needed is when the migration of the state from an old to a new version of the class requires the developer's knowledge of the code.[2] Therefore, in most cases, the use of the technique is completely transparent to both the user and the developer.

Support for changes at different levels. Although optimized to operate at the class level, our technique lets us perform updates at different levels of granularity: from changes involving a single statement to structural modifications of the whole application.

No runtime-system support required. As stated above, our technique is designed to work on any program running on any implementation of the JVM that complies to the standard [12].

Although many solutions to the problem of dynamic software updating have been proposed [2, 3, 5, 6, 9, 8, 10, 13, 16] several international organizations (including the Object Management Group and the Java Community Process) are working on proposals for specifying models and APIs supporting dynamic application updates, to the best of our knowledge, our technique is currently the only technique that works for generic Java applications and does not require a customized JVM to be applied.

To validate our approach, we implemented our technique in a tool called DUSC (Dynamic Updating through Swapping of Classes). In this paper, we describe our implementation of DUSC and an empirical study that we performed using such implementation. In the study, we performed dynamic updates of a real Java subject and (1) compared the behavior of the swapping-enabled applications with the behavior of the corresponding original applications, (2) performed different dynamic updates of the application, and (3) measured the cost of the technique in terms of performances. The results of the studies show that *Dusc* can be effectively applied to Java software with only little overhead in both execution time and program size.

The main contributions of this work are:

1. definition of a new technique for dynamic update of Java software,
2. implementation of the technique in a tool, and
3. empirical validation of the technique on a real Java subject.

[1] We refer to such modified application as *swapping-enabled application* hereafter.

[2] Consider, as an example, a situation in which a new version c' of class c is provided and c' contains a new attribute z that is defined as having the value of $c.x$ plus $c.y$. In such a case, there is no way for the program to infer the way z should be initialized, unless c''s developer provides such information.

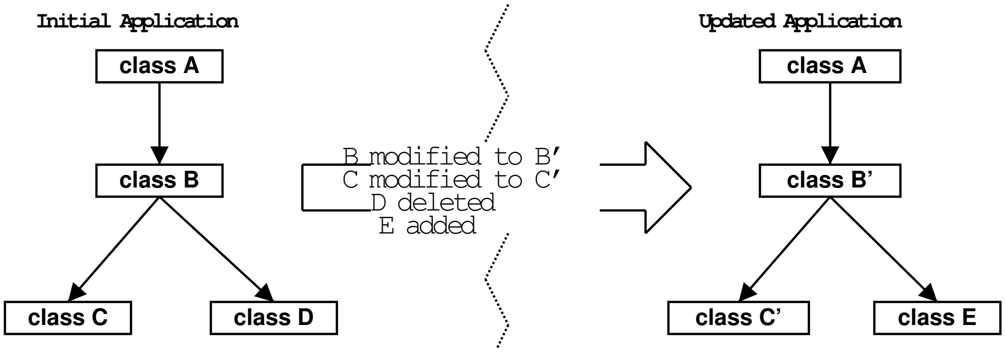

Figure 1: Example of application update.

2 Dynamic Software Updating

In this section, we describe our technique for dynamic updating of Java code. First, we illustrate the problem and provide an intuition of our proposed solution. Then, we describe the assumptions that must be satisfied for the technique to be applicable. Finally, we provide technical details about the technique.

2.1 Overview

Dynamic software updating is the task of updating parts of a program without having to terminate its execution. As stated in the Introduction, our goal is to define a software-based technique for dynamic updating of Java software that requires no support from the runtime system and can be generally applied to any Java application running on any implementation of the Java Virtual Machine (JVM) that is compliant with the Sun's JVM specification [12] (i.e., we define a technique that works without any support from the run-time system).

Given two versions P and P' of a generic Java program, the differences between P and P' can be expressed in terms of three, possibly empty, sets: the set ADD of added classes (i.e., classes in P' and not in P), the set DEL of deleted classes (i.e., classes in P and not in P'), and the set MOD of modified classes (i.e., classes that are both in P and in P', but differ in the two programs). Therefore, to be able to dynamically update a program, we must be able to (1) add classes to the program, (2) remove classes from the program, and (3) substitute classes in the program. (Note that the above description refers to the update of the code only and does not account for the state of the executing program. The problem of migrating the state of P to P' is addressed in Section 2.3.)

Consider, for example, the two versions of a simple application shown in Figure 1. On the left-hand side of the figure, we show the initial application, which is composed of classes A, B, C, and D; class A uses class B,[3] which uses classes C and D in turn. On the right-hand side of the figure, we show the updated application, which is composed of classes A, B', C', and E. In the figure, we also show that, to get from the initial to the updated application, class B has been modified (replaced) with class B', class C has been modified (replaced) with class C', class D has been deleted, and class E has been added. Therefore, in this case, set ADD contains class E, set DEL contains class D, and set MOD contains classes B and C.

Adding classes to a running program is straightforward and requires only some engineering effort; the only requisite is to be able to add class files in any location that is accessible through the class path.[4] Each newly-added class will then be loaded the first time the program uses it (e.g., when the program instantiates the class or accesses one of the class's static members). In the case of the example, after we add class E, the class will be automatically loaded the first time that an instance of class B' uses it.

Removing classes from the application is also fairly straightforward; in the Java run-time system, the garbage collector automatically removes from the program memory objects that are no longer referenced in the application. In the case of the example, after we substitute class B with class B', existing objects of

[3] The specific kind of use of the class—e.g., method invocation on an instance of B—is irrelevant in this context. We consider that a class $C1$ *uses* a class $C2$ if $C1$ may contain a reference to $C2$.

[4] Although different specific cases may need slightly different solutions, we do not describe such solutions because the focus of the paper is on the updating technique, rather than on the engineering of its implementation.

the old type B are no longer referenced (see below) and are thus garbage collected. As a consequence, also objects of class D are no longer referenced (objects of class C can be referenced only by objects of class B) and are garbage collected as well. (The above description assumes that we remove only classes that are no longer used by any of the classes in the application, which is an obvious requirement.) From a practical standpoint, the deleted classes could also be physically removed if needed (e.g., to save space).

Substituting classes in the application is far from trivial. For each class C that is being substituted with a class C', we must (1) modify the application so that any new instance of C that is created after the update is an instance of C', and (2) migrate all the existing objects of type C in the application at the moment of the upgrade to objects of type C'. Therefore, we need a way of addressing these issues and enabling the substitution of one or more (possibly all) classes in the application. More precisely, we need a technique that takes the bytecode of a Java application and a list of the classes that have to be changeable (*target classes*, hereafter) as input, and produces an equivalent program in which the specified classes can be dynamically substituted (i.e., a *swapping-enabled* version of the application).

Our technique addresses the above problems by using wrapper classes . A *wrapper class* for a class C is a class that has the same name and provides the same interface as class C and acts as a proxy for such class. Given a program P, we generate a new program P_{dusc} in which we create a wrapper class for each target class C in P. We also modify C so that none of its clients can obtain a direct references to any instance of C and all calls to C in P result in call to the wrapper for C in P_{dusc}. Because the wrapper is a proxy, every time a method in the wrapper is called, the wrapper forwards the corresponding call to an appropriate instance of class C. This level of indirection lets us replace class C with a different class at runtime. To perform the substitution of C with C', we (1) create a new instance of C' for each instance of C, (2) transfer the state from the old to the new instance, and (3) update the wrappers so that they refer to the newly created instances after the update. The application can then continue its execution (transparently) using the updated class(es).

Note that we provided a high-level and intuitive view of the way our technique operates, in which most details are omitted. For the technique to work in the presence of Java object-oriented features, such as inheritance, polymorphism, and dynamic binding, we have to address and solve a number of issues and consider several important details. Such details are thoroughly presented in Section 2.3.

2.2 Assumptions and Limitations

In the following, we present restrictions and limitations of our technique.

Access to public and protected fields Our technique requires that no class in the application accesses public or protected fields of any of the target classes directly. All such accesses must be performed through appropriate accessor methods—typically, *get* and *set* methods. (This restriction does not apply to fields that are constant and do not change value from version to version.) This assumption, necessary to maintain the level of indirection required by our technique, is not overly restrictive; it is a common requirement for object-oriented programming in general because it enforces information hiding [4, 14]. If this requirement is not satisfied, we modify the application through bytecode rewriting so that all direct accesses are transformed into one or more invocations of an accessor method.

Reflection Our technique assumes that reflection is not applied to any target class or any component of a target class. *Reflection* "allows programmatic access to information about the fields, methods and constructors of loaded classes, and the use of reflected fields, methods, and constructors to operate on their underlying counterparts on objects, within security restriction" [17]. In this paper, we consider methods that inspect the information about a specific class, such as the methods in `java.lang.Class`, as a form of reflection as well. If a statement uses information obtained through reflection about either a target class or its members, the substitution of that target class with a wrapper would affect the behavior of the application (i.e., it would not be semantics preserving).

Unmodifiable interfaces To preserve the type safety of Java, our technique requires for a new version of a target class to provide the same interface provided by the previous version of the class. In other words, each version of a target class must provide the same set of public methods. This requirement does not apply to private methods and instance variables, that can be freely added to or removed from a new version of the target class. It is worth noting that, if the interface of a class does need to be changed, our technique can still be used. In that case, the class whose interface is changed would be deleted from the application, and a new class would be added that provides the new interface. Consider, for example, the application in Figure 1 and assume that we need to change class E's interface. We would then create a new class F that provides the new interface and update the application by removing class E, adding class F, and replacing class B' with a class B'' that uses the new interface. In this case, though, either the state of existing instances of E would be lost or some hacking on the developer side would be required to be able to migrate the state of instances of class E to the corresponding instances of class F.

Native methods Our technique assumes that target classes do not contain native methods. *Native methods* are methods implemented in another programming language, such as C, that can be used in Java through the Java Native Interface (JNI [11]). If a target class contains a native method, our technique cannot analyze the class to build suitable wrapper and implementation classes (see Section 2.3 for a definition of implementation class). In practice, this assumption holds in most cases because the use of native code is required only for a few applications that have a low-level interaction with the underlying system.[5]

Requirements for class updating Whereas the operations of adding and deleting classes can be performed at any time, substituting a class requires that no methods of the class are executing during the update (i.e., no methods of the class are on the stack). Therefore, when our technique *attempts* to perform a dynamic update, whether or not an update actually takes place depends on runtime factors. Although in most cases the ability to perform the update only depends on waiting for some method of the target class to complete its execution, this limitation could prevent the updating of some specific classes in the application, that is, classes such that one of their methods is always on the stack. For example, assume that class A in Figure 1 has a method `main` whose body contains an infinite loop in which some method of B is called. In such a case, method `main` never terminates its execution, and we cannot update class A without shutting down the application.

Security As we stated above, our technique enables the updating of classes by analyzing the target classes and generating suitable wrapper classes. The process also involves performing some changes to the target classes (as described in Section 2.3). Such transformation of the target classes could involve some security issues—malicious users may obtain access to protected and private members of the target classes. However, this problem does not occur in the original application, which uses the target classes in a secure way by construction. For the problem to occur, a malicious user needs to (1) know about the details of our technique, (2) know about the application internals, and (3) write a suitable program that reproduces the application behavior while exploiting the security leaks.

2.3 The Technique

The process for the dynamic updating involves two distinct parts. The first part consists of transforming the application to make it swapping-enabled by generating wrappers and utility classes for each target class in the application. The second part consists of the actual update, in which target classes are updated with new versions and other classes are possibly added to or removed from the application. In the following, we describe the two parts separately.

2.3.1 Transformation

The goal of the transformation is to obtain a new application that can be dynamically updated, yet behaves exactly like the original application. To this end, we substitute each target class C (possibly all classes in the application), with a compound that consists of an implementation class (C_i), an interface class (C_a), a wrapper class (C_w), and a state class (C_s).

[5] The fact that library classes often use native code is not a problem for the application of our technique. The requirement only affects target classes, which are almost always in the application, rather than in the library.

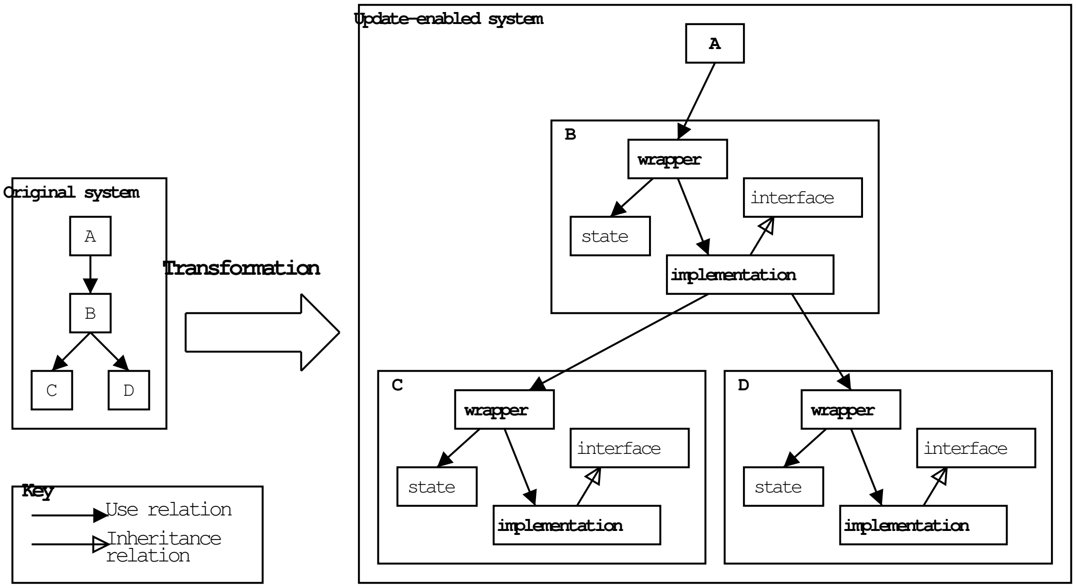

Figure 2: Application transformation.

Implementation class. C_i contains the actual implementation of a version of class C. Right after the transformation, class C_i corresponds to version zero of C, that is, the version of class C in the original application. C_i contains a slightly modified version of all of C's members, both fields and methods. In addition, C_i contains an additional field, which is used to store a reference to the corresponding wrapper class, an additional method, which encode the C_i's state in an instance of the state class and returns such instance, and a set of special constructors, which are used when performing an update of the class (see Section 2.3.2 for details).

Interface class. C_a is an abstract class that is implemented by all implementation classes corresponding to different versions of C. We use C_a within the wrapper to call methods on the implementation class. The wrapper can access different implementation of C through a reference whose static type is C_a. We could obtain the same results without using C_a, by exploiting reflection, but the resulting wrapper would be much less efficient in this case.

Wrapper class. C_w provides the same interface that class C provides and is indistinguishable from C to any client of C in the application. For every method m of C, there is a corresponding method m_w in C_w with the same signature. The goal of each method m_w is twofold: performing bookkeeping for the update process, as described below, and performing the same operation that m performs. m_w accomplishes the latter goal by invoking the corresponding method m in the current implementation class for C and by returning the value that the implementation class's method returns. To be able to call method m in the right instance of class C_i, the wrapper class is provided with a field of type C_a (which is a superclass of C_i). For each instance of the wrapper, such field contains a reference to the object of class C_i that corresponds to that wrapper. In addition, the wrapper class contains the following static members:

- A `Vector` of C_a. This vector is used to store references to all instances of C_i in the application. An element is added to the array every time a constructor of the wrapper is called, after the corresponding constructor for C_i has been called. Every time an instance of the wrapper is garbage collected, the reference to the corresponding instance of C_i is eliminated. We refer to this vector as *instances vector*.
- An integer value that keeps track of the number of methods of C_i currently on the stack. This value is incremented by the wrapper every time a method is called, before calling the corresponding method in C_i, and decremented by the wrapper every time one of C_i's methods returns. We refer to this vector as *stack counter*.

- A method that can be used to request an update of C_i. We refer to this method as *swap-request method.*

State class. C_s is a class whose objects can be used to encode the state of an instance of C_i. Class C_s has the same fields as class C_i and is used to migrate objects of an updated class from the old to the new version, as described in Section 2.3.2.

To give an example of an application transformation, Figure 2 shows how the application in Figure 1 would be transformed to become swapping-enabled.

When substituting a target class with the set of four classes described above, we must pay special attention to how we handle object-oriented features in Java. Although the handling of some features, such as instance methods, is straightforward, the handling of other language constructs, such as `super` calls, can be very complex (and can cause the swapping-enabled application to behave differently with respect to the original application, if performed incorrectly). In the following, we describe how we handle this set of "possibly problematic" Java features.

Inheritance In the swapping-enabled application, the class hierarchy is different with respect to the one in the original application. Due to class renaming, the wrapper classes take the place of the corresponding target classes in the inheritance tree. Each implementation class is a subclass of the corresponding interface class, which does not belong to the original inheritance tree. State classes are also not part of the original inheritance tree. For example, assume that in the system shown in Figure 2 `A` is a superclass of `D`. In the corresponding swapping-enabled system, `A` would then be a superclass of the wrapper class created for `D`, whereas the implementation class for `D` would not belong to the hierarchy any longer. There is a specific reason why implementation classes are not subclasses of their wrappers (a solution which would have simplified the whole design). If implementation classes were subclasses of their wrappers, a new wrapper would be generated every time there is an update, when objects of the new type are created to replace objects of the old type. Because the wrapper is responsible for the bookkeeping during the updates, it cannot change between different versions and thus cannot be created from scratch every time. Therefore, wrappers and implementation classes must be related by delegation, rather than by a class-subclass relation.

Static methods Static methods in the target class are not represented as static methods in the implementation class, and instance methods are used instead. The wrapper redirects static method calls to a special instance of the implementation class that is created to this end. In this way, the wrapper can exploit method overriding to run the correct version of the method through the interface class, so avoiding the problem of static methods being statically linked.

`this` The explicit use of `this` is a problem for our approach because it allows for bypassing the wrapper class. If `this` is returned by a method of an instantiation class C_i, it is possible for whoever gets the reference to `this` to access C_i directly, without going through the wrapper class. Furthermore, direct accesses to `this` are a problem also within C_i itself, for the same reason. Therefore, when creating the instantiation class, we redirect all references to `this` to references to the corresponding wrapper. (To this end, we add a field to each implementation class to store such reference, and modify the constructors of the implementation class so that the field gets suitably initialized, as described in the next paragraph.) More precisely, any call to a `public` internal method that occurs in an implementation class gets redirected to the corresponding method in the wrapper class, and any direct reference to a `public` field that occurs in an implementation class gets replaced with a call to an appropriate accessor method suitably created in the wrapper class.

Constructors Because of renaming, every time a target class is instantiated, the corresponding wrapper gets instantiated instead. When a wrapper is instantiated with a specific constructor, it calls the appropriate constructor for the implementation class and passes a reference to itself as an additional argument. Such reference is stored in the generated instance of the implementation class and is used in place of `this` to avoid direct accesses to the implementation class. Besides storing the reference to the wrapper and performing some additional bookkeeping, the constructor in the implementation class performs the same operations as the constructor in the target class. Some special constructors are also added to the implementation class:

one or more *copy* constructors, which perform the migration of the state from an old to the new version of the implementation class during an update, and a constructor that builds (once per class) the special object that is used by the wrapper to run static methods.

Finalizers Possible problems may arise if the finalizer for a target class gets executed when an object of the corresponding implementation class is garbage collected. Objects of implementation classes are garbage collected as a consequence of an update, and the finalizer may have a side effect on elements of the object's state that survive the object and migrate to the new version. Consider, for instance, a class C that contains a reference f to a file that is open by C' constructor and closed by C's finalizer. When we update C to C', all instances of C are eventually garbage collected, but references f must keep their state because they migrate to the newly created objects of type C'. Therefore, when creating the implementation class for a target class, we rewrite the code so that an implementation class's finalizer is run only when the finalizer of the corresponding wrapper is run.

Invocations to `super` methods As we described above, in the swapping-enabled system the original class hierarchy is lost. Therefore, invocations of superclass methods in an implementation class need to be suitably modified. To handle this problem, we (1) add to wrapper classes ad-hoc methods that invoke the superclass methods, and (2) modify calls to `super` in the implementation classes so that they result in invocations of those methods instead.

2.3.2 Dynamic Update

Given a swapping-enabled application, in which all classes that can be updated have been suitably processed, actual code updates are relatively straightforward to accomplish. Note that, in the following description, we assume that the user performs valid updates. A dynamic update from program P to program P' using a given state mapping[6] is *valid* if, after the change, the execution is guaranteed to reach a reachable state of P' in a finite amount of time [6]. In this context, we can simplify the above definition and say that a dynamic update is valid if it brings P' into a consistent state.

An update consists of a set of classes that must be added to the system (ADD), a set of classes that must be removed from the system (DEL), and a set of classes that must be modified, that is, substituted (MOD). Addition and deletion of classes are mostly an engineering problem, as discussed in Section 2.1, and can be performed straightforwardly.

When an updated version of a target class is available, we first generate the corresponding implementation class, which will be used during the update. The actual substitution of classes is then performed through interaction with the wrappers of the classes to be updated. For each class that must be updated, the swap-request method is called, passing the new version of the implementation class as an argument. For the sake of the presentation, in the following, we describe the update assuming that a single class is updated. In the case of multiple classes, the process is analogous, but we use a two-phase thread-based locking mechanism to ensure the atomicity of the updates.

When a swap-request method is called in the wrapper for a class C, passing the new version C_i' of C_i as a parameter, the wrapper first checks the value of the stack counter. If the value is greater than zero (i.e., at least one method of class C_i is currently executing), the wrapper returns an error and cancels the update. Otherwise, the wrapper performs the update by iterating through the instances vector and, for each instance (including the instance that we use to "simulate" static members), performs the following actions:

1. invokes on the instance the special method that encodes the current instance state in a state object and returns it;
2. creates a new object of type C_i' using the special copy constructor that takes a state object as a parameter and uses it to suitably initialize C_i''s state; and
3. changes the value of the reference to the old object in the instances vector to point to the newly created object.

[6] The *state mapping* consists of the way we migrate the state from a previous to the next version of the class. In the simplest case, state migration is achieved by performing a shallow copy of corresponding fields in the two classes. If a more sophisticated migration is needed, the developer must provide an ad-hoc state mapping.

After each class for which a new version is available has been suitably substituted, the classes in the *ADD* set are added to the system, the classes in the *DEL* set are removed from the system, and the update terminates.

Note that the constraint that the swapping of a class can be performed only if no methods of that class are currently executing may be overly restrictive. Static analysis techniques may be used to identify methods whose execution does not prevent an update. Based on our experimental findings, we shall decide whether it is worth investigating this and other improvements to the technique.

3 Empirical Evaluation

In this section, we describe the system that we developed and that implements our dynamic-updating technique. We also report on the results of an empirical study that we performed using our system on a real Java subject.

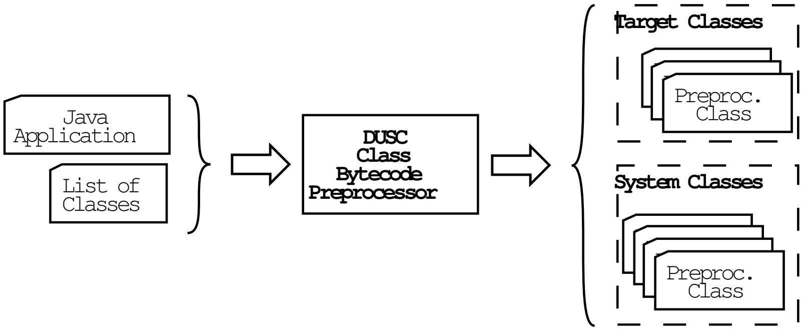

Figure 3: Dusc tool—Preprocessing stage

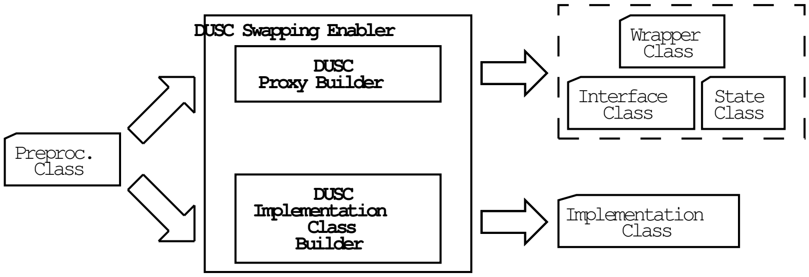

Figure 4: Dusc tool—Swapping-enabling stage

3.1 System

To investigate the usefulness of our technique, we developed a tool that implements our dynamic-updating technique, Dusc (Dynamic Updating through Swapping of Classes). Dusc is written in Java and exploits SOOT [15] capabilities to perform bytecode rewriting. Figures 3 and 4 show a high-level view of Dusc.

Dusc is logically composed of three main components:

Class Bytecode Preprocessor (CBP). CBP takes the bytecode of a Java application and a list of target classes as input and produces two sets of classes: the preprocessed target classes, which are the classes that will be swappable in the swapping-enabled application, and the preprocessed system classes, which are the classes that will be immutable. The preprocessing stage performs some "normalization" of the code, such as modifying all direct accesses to public fields of target classes with calls to appropriate accessor methods and adding accessor methods to the target classes if needed.

Proxy Builder (PB). PB takes the bytecode of a preprocessed class as input and generates a suitable wrapper class, interface class, and state class.

Implementation Class Builder (ICB). Analogous to the proxy builder, this component reads bytecode of a preprocessed class and produces the corresponding implementation class.

PB and ICB builder are used jointly when producing the swapping-enabled version of an application, whereas ICB is used in isolation when a new version of a class is available and only the corresponding implementation class is needed.

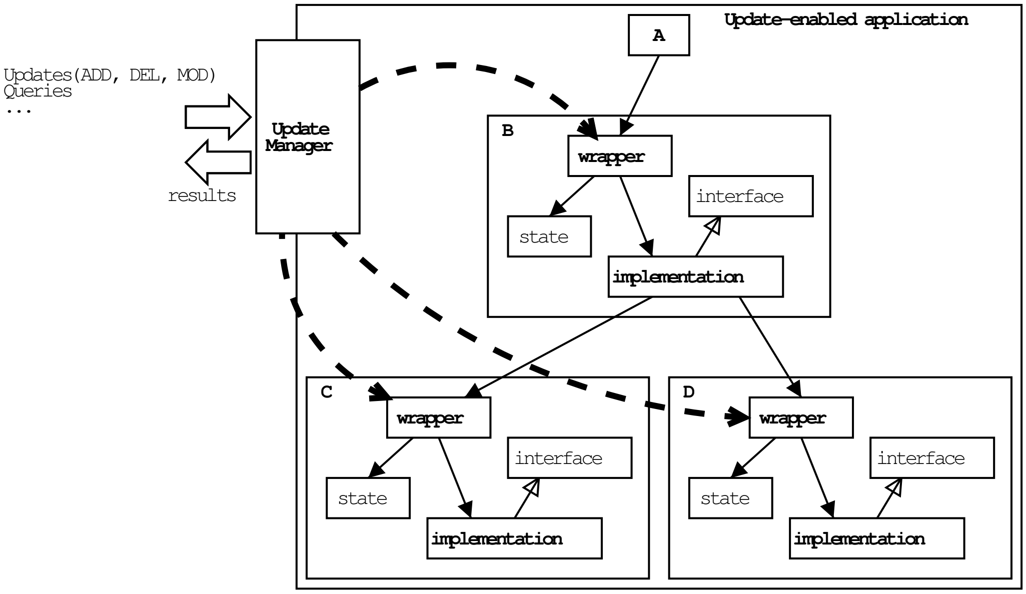

Figure 5: Dusc tool—Swapping-enabling stage

Figure 5 shows how updates can be performed on a swapping-enabled application. We integrate into the application a module, called *Update Manager* (UM), that runs in its own separate thread. The module is initialized by bootstrap code inserted at the entry of the `main` method of the application and is aware of (1) which wrapper classes are in the system, and (2) which version of each implementation class is currently in the system. The UM module provides an interface[7] through which a user can request an update, by providing the three sets of classes ADD, DEL, and MOD.

When UM receives a request for an update, it contacts the wrappers involved in the update (i.e., the wrappers that corresponds to classes in the MOD set) and, using a two-phase locking mechanism, either performs an atomic update or cancels the update if one or more wrappers cannot perform the swapping (because a method in the corresponding class is currently executing). If the update is successful, UM also suitably handles the addition of classes to and the removal of classes from the system, according to the contents of the ADD and DEL sets.

UM's interface can also be used to perform simple queries on the status of the application with respect to the updating. To date, we have implemented queries for gathering three kinds of information: (1) list of the names of the wrappers in the application, (2) current version of the implementation class for a given wrapper, and (3) current list of pending updates. So far, we have used the query mechanism mostly to check whether a requested update has actually been performed or is still pending.

3.2 Case Study

To validate the tool and the technique, we performed two empirical studies using DejaVOO [7], a regression testing tool developed by some of the authors, as a subject program. DejaVOO is part of the Jaba (Java

[7] The current interface is implemented through UNIX sockets using a simple communication protocol.

Architecture for Bytecode Analysis [1]) analysis framework, and consists of 43 classes and approximately 11KLOC. Given two versions of a program, DejaVOO (1) analyzes them to identify which parts of the code are affected by the changes between the two versions, and (2) reports to the user what needs to be retested based on the results of the analysis.

In the first study, we generated a swapping-enabled version of DejaVOO (DejaVOO$_{dusc}$, hereafter) by selecting all classes as target classes. The goals of the study are (1) to check that the original application and the (fully) swapping-enabled application behave in the same way, and (2) to assess the overhead caused by Dusc in terms of both execution time and memory requirements.

For the first goal, we used a regression test suite that we had developed for DejaVOO, and ran it on both the original application and on the swapping-enabled version. For each test case, we then compared the results and verified that they matched for the two applications.[8]

For the second goal, we ran the same regression-test suite and measured the execution time and the memory requirements for both DejaVOO and DejaVOO$_{dusc}$. More precisely, for both applications, we measured the time and memory required to run each test case. For each test case, we then computed the difference in both time and memory required. The result are reported in Tables 1 and 2.

	Differences in execution time		
	maximum	minimum	avgerage
Absolute	0.79	0.02	0.27
Percentage	3.36%	0.08%	1.13%

Table 1: Execution time results (absolute differences are expressed in seconds).

	Differences in memory requirements		
	maximum	minimum	average
Absolute	6,541	1,034	1,762
Percentage	18.92%	3.98%	6.36%

Table 2: Memory requirements results (absolute differences are expressed in kilobytes).

In Table 1 (resp., Table 2), we show the maximum, the minimum, and the average difference in execution time (resp., memory requirements) over all test cases. The figures are shown both as absolute differences and as percentages. Absolute differences are expressed in seconds and kilobytes for time and memory, respectively.

The results show that our technique can be effectively applied to Java software with only little overhead in both execution time and memory requirements. As far as timing is concerned, the swapping enabled application required a maximum of 0.79 seconds (3.36%) more than the original application to be executed. Although the maximum for the memory requirements is 6,699 more kilobytes (18.92%), which is a worse result with respect to the timing, the average overhead is 1,762 kilobytes (6.36%) over all test cases.[9]

In the second study, we performed different dynamic updates of DejaVOO. The goal of the study is to validate the updating mechanism with respect to both the class swapping and the state migration.

DejaVOO usually terminates its execution after completing an the comparison of two program's versions. Therefore, to simulate a continuously running application, we wrote a driver on top of DejaVOO that indefinitely reads an input from the user, performs its analysis on the two programs, and prints out the results. Based on a set of different releases of DejaVOO in our repository, we selected a base version and a set of four later versions. We then identified the pairwise differences between the different versions and

[8] The comparison was performed by comparing the diagnostic outputs produced by DejaVOO and DejaVOO$_{dusc}$, which also report intermediate analysis results. Although the matching of such outputs is not a proof of correctness as such, almost all parts of DejaVOO are involved in the generation of the diagnostic information, and thus it is unlikely that a problem in the application would not result in any difference.

[9] There are a few executions for which the overhead is considerably higher than the average. We plan to investigate the characteristics of those executions to assess whether there are special conditions in which our technique could be improved.

built four updates expressed in terms of the sets ADD, DEL, and MOD. The sizes of the sets for the four updates are shown in Table 3.2.

	update#1	update#2	update#3	update#4
ADD	0	2	1	0
DEL	0	1	0	0
MOD	2	5	3	3

Table 3: Size of updates sets.

For each update, we performed the following steps:

- perform the update,
- verify that the update actually occurred,
- perform the analysis, and
- collect the analysis results.

We then compared the collected results with the results obtained for the four original versions of DejaVOO considered. The comparison was performed as for the previous study. Because the different analyses are performed on the same two versions of a program, any inconsistency in the class swapping or in the migration of the state during an update should result either in a runtime error or in an incorrect result. The results matched for all the analyses and versions considered.

Some things about the second study are worth noting. For the considered updates, we never ran into the problem of having a method of a class to be updated that was on the stack (which would have prevented the update from occurring). Also, we never needed to define ad-hoc state mapping for any of the considered versions. Finally, the time required for an update was always in the order of magnitude of the milliseconds (which could be problematic for real-time applications, but is in general an acceptable time).

Like any empirical study, this study has limitations. We have considered the application of our technique to a single program, a single set of four subsequent versions, and a single test suite. We cannot therefore claim generality for our results. However, the program and modified versions we used are derived from an actual implementation, the updates we considered are real updates, and the test suite we used is a coverage-adequate test suite for the program. Nevertheless, additional studies with other subjects are needed to address such questions of external validity.

4 Related Work

Several dynamic software update techniques and systems have been presented in the literature [2, 3, 5, 6, 9, 8, 10, 13, 16]. Here, we do not consider updating techniques based on hardware redundancy, which are quite costly and have very limited application, as we stated in the Introduction.

Among the software-based techniques, several approaches are targeted to very specific languages and environments and do not directly compare with our technique [2, 3, 5, 10, 13].

Other approaches, such as the one proposed by Segal and Frieder [16] and the one presented by Hicks, Moore, and Nettles [8], address the problem in a more general way, but, unlike our technique, rely on the support from the runtime system to perform dynamic updates.

Gupta, Jalote, and Barua present a framework for dynamic updating and address several theoretical issues related to this task [6]. Although Gupta et al.'s theoretical findings are of general validity, the work is mostly focused on validity of updates.

The approach that is most closely related to our technique is the one presented by Hjálmtýsson and Gray [9]. By exploiting the C++'s template mechanism, Hjálmtýsson and Gray's approach allows for defining C++ classes that can be dynamically updated. Similar to our technique, the approach is based on the use of a wrapper/proxy that adds a level of indirection and permits class swapping. However, the approach cannot be applied to Java programs because of the lack of some program features, such as templates, in the Java language. Moreover, unlike our technique, the approach by Hjálmtýsson and Gray does not permit the updating of static members and, most important, requires the programmer to explicitly implement a class as swappable.

To the best of our knowledge, our technique is the first technique that, at the same time, (1) works on Java code, (2) does not require any runtime-system support and can thus be applied on any platform that provides a standard implementation of the JVM, (3) lets parts of the program that are not involved in the update continue their execution during the update, (4) automatically builds the swapping-enabled system without requiring the developer's intervention (thus facilitating clear separation between development and updating), and (5) permits updating of both types and implementation.

5 Conclusion

We presented a new software-based technique for dynamic updating of Java software that permits substituting, adding, and deleting classes without having to stop the program. Our technique requires no support from the runtime system and can thus be applied to any program running on any standard Java Virtual Machine. Our technique first modifies the application using class renaming and code rewriting, to enable the dynamic update, and then performs the updates by dynamically swapping classes at runtime.

We also presented a tool, DUSC (Dynamic Updating through Swapping of Classes), that we developed and that implements our technique. We used DUSC to perform an empirical study to validate the technique on a set of real Java subjects. The results of the study showed that our technique can be effectively applied to Java software with only little overhead in both execution time and program size.

In future work, we plan to improve the existing tool and to extend our experimentation in two directions. First, we want to apply our technique to a bigger number of subjects, to verify the statistical meaningfulness of our experimental results. Second, we want to study different releases of existing subjects to categorize the types of changes occurring between different versions of an application; such a classification will provide important insights on whether our approach needs to be extended (e.g., if changes in the interface of classes between different versions are common, or if ad-hoc state mapping are often needed).

References

[1] Aristotle Research Group. JABA: Java Architecture for Bytecode Analysis. `http://www.cc.gatech.edu/aristotle/Tools/jaba.html`.

[2] T. Bloom. Dynamic module replacement in a distributed programming system. Technical Report MIT-LCS//MIT/LCS/TR-303, Massachusetts Institute of Technology, Laboratory for Computer Science, Mar. 1983.

[3] R. Fabry. How to design A system in which modules can be changed on the fly. In *Proceedings of the Second International Conference on Software Engineering*. IEEE, Oct. 1976.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, Mass., 1995.

[5] H. Goullon, R. Isle, and K.-P. Löhr. Dynamic restructuring in an experimental operating system. *IEEE Transactions on Software Engineering*, 4(4):298–307, July 1978. Special collection from the Third International Conference on Software Engineering.

[6] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, Feb. 1996.

[7] M. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, November 2001.

[8] M. Hicks, J. Moore, and S. Nettles. Dynamic software updating. In C. Norris and J. J. B. Fenwick, editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and*

Implementation (PLDI-01), volume 36.5 of *ACM SIGPLAN Notices*, pages 13–23, N.Y., June 20–22 2001. ACMPress.

[9] G. Hjálmtýsson and R. Gray. Dynamic C++ classes. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 65–76, Berkeley, USA, June 15–19 1998. USENIX Association.

[10] I. Lee. *DYMOS: A Dynamic Modification System.* PhD thesis, Department of Computer Science, University of Wisconsin, April 1983.

[11] S. Liang. *Java Native Interface: Programmer's Guide and Specification.* Addison-Wesley, Reading, MA, USA, 1999.

[12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison Wesley Longman, Inc., second edition, Apr. 1999.

[13] J. Magee, J. Kramer, and M. Sloman. Constructing distributed systems in conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, June 1989.

[14] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, second edition, 1997.

[15] Sable Group. SOOT: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`.

[16] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2):53–65, Mar. 1993.

[17] Sun Microsystems. Java2 Platform, API Specification. `http://java.sun.com/j2se/1.3/docs/api/`.