WASHINGTON UNIVERSITY
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

RUN-TIME MODIFICATION OF THE CLASS HIERARCHY IN A LIVE JAVA DEVELOPMENT ENVIRONMENT

by

Joel R. Brandt

Prepared under the direction of Professor Kenneth J. Goldman

A thesis presented to Washington University
School of Engineering and Applied Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science

May, 2005

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

RUN-TIME MODIFICATION OF THE CLASS HIERARCHY IN A LIVE JAVA DEVELOPMENT ENVIRONMENT

by Joel R. Brandt

ADVISOR: Professor Kenneth J. Goldman

May, 2005
Saint Louis, Missouri

Class hierarchy design is central to object-oriented software development. However, it is sometimes difficult for developers to anticipate all the implications of a design until implementation is underway. To support experimentation with different designs, we extend prior work on live development environments to allow run-time modification of the class hierarchy. The result is a more fluid object-oriented development process, in which immediate feedback from the executing program can be used to guide hierarchy design.

This thesis presents a framework and developer support for run-time modification of class inheritance relations in JPie, a live visual programming environment for Java. Most notably, the framework supports class reloading without modification of the Java Virtual Machine.

Contents

List of Figures

Acknowledgments

Foremost, I thank my advisor, Dr. Kenneth J. Goldman, for his hours of advice and guidance. I also thank the past and present members of the JPie development team: James Aguilar, Ben Birnbaum, Ben Brinckerhoff, Vanessa Clark, Melanie Cowan, Matt Hampton, Dylan Lingelbach, Oren Melzer, Adam Mitz, Brandon Morgan, Jonathan Nye, Sajeeva Pallemulle, and Richard Souvenir. Additionally, I thank the students in CS123 for their feedback on the class hierarchy GUI.

This work was supported in part by the National Science Foundation under CISE Educational Innovation Grant 0305954.

Joel R. Brandt

Washington University in Saint Louis
May 2005

1 Introduction

JPie is a tightly integrated programming environment supporting live development of Java applications through direct manipulation of graphical representations of programming abstractions [8, 9, 10]. JPie permits class modifications in running applications, with changes affecting existing instances of those classes. This run-time modification eliminates the edit-compile-test cycle.

Enabling run-time modification of class hierarchy relations is important for two reasons. First, it rounds out the set of run-time changes allowed in JPie, which prior to this work consisted only of fine-grain modifications of classes, such as creation, deletion, and modification of fields, methods, and method bodies. Second, run-time modification of the class hierarchy provides a useful tool in computer science education. Allowing run-time modification of the hierarchy lets beginning programmers experiment with system design changes easily, illuminating the full power of object-oriented programming.

The majority of fine-grain run-time changes (addition, modification, and removal of fields, methods, and code) are handled through a pairing of Java's reflection mechanism and JPie's dynamic classes [11]. However, the prior work on dynamic classes in JPie assumes that the parent and implemented interfaces of a dynamic class do not change over time. This thesis removes that assumption to permit coarse-grain changes as well: the class hierarchy can be modified while the program is running.

For interoperability with standard Java classes, JPie's dynamic classes use a compiled proxy class, which we call a compiled peer. If we allow ancestors or implemented interfaces of a dynamic class to change, then a new compiled peer class must be created and loaded. In typical Java applications, the system class loader

built into the Java Virtual Machine (JVM) handles the loading of classes. However, the system class loader does not provide a way to unload or reload classes, a necessary feature to support changes to the class hierarchy. In order for JPie to allow run-time modification of the class hierarchy, a class loader which allows the reloading of classes is necessary.

The remainder of the thesis is organized as follows. We begin with background on JPie, a discussion of related work, and a brief introduction to Java's class loading system. In Section 2 we present a dynamic class loader which provides the abstraction of class reloading. This class loader is implemented completely in Java, and does not require modification of the JVM. Section 3 examines how modifications to the class hierarchy can impact a live system, and discusses how these issues are dealt with in JPie. In Section 4, we present a Graphical User Interface (GUI) for manipulating class hierarchy relations. We conclude, in Sections 5 and 6, with a look at applications for the dynamic class loader outside of JPie, and a discussion of our plan for future work. A discussion of the tests used for our implementation can be found in Appendix A.

1.1 Background on JPie

JPie is a live visual programming environment for the Java Programming Language [8, 9, 10]. With JPie, programmers are able to modify their application while it is running. Possible modifications include the creation of classes, and the addition, removal, and modification of all fields, methods, and code. At any time, these classes can be exported as Java source code or Java bytecode, and used outside of JPie.

JPie permits fine-grain modifications of classes through a visual interface which eliminates the possibility of syntactic errors. In addition to standard Java support for graphical user interfaces and threads, JPie provides shortcuts that streamline the creation of views, events, and behaviors. Views provide a visual representation of object instances, events extend views to provide basic event handling, and behaviors provide simple threading capabilities. A more thorough examination of JPie can be found elsewhere [8, 9, 10].

JPie supports live development through the use of dynamic classes [11]. Dynamic classes are interoperable with compiled classes, including run-time method overriding and polymorphism. Dynamic classes consist of two main parts, the dynamic portion, which can be modified at run-time, and the compiled peer, which enables interoperability with compiled classes. The dynamic portion of a dynamic

class contains the user's code (fields and methods). When a user adds, modifies, or removes code, the dynamic portion of the dynamic class is updated.

The compiled peer is an automatically generated class that expresses the portion of a dynamic class's interface dictated by its class hierarchy relationships to compiled classes. For example, suppose `Parent` is a compiled class and `Child` is a dynamic class which extends `Parent`. Then the compiled peer of `Child` would contain an implementation of each method inherited from `Parent` (and its ancestors). These methods either pass their calls on to the dynamic portion of `Child` (if an overriding method has been defined by the user) or make the appropriate super calls. For interoperability with compiled classes, each instance of a dynamic class presents itself as an instance of its compiled peer. Compiled classes, then, can call methods polymorphically on instances of the compiled peer, which dispatch dynamically overridden methods by proxy into the dynamic portion of the dynamic class. Through this mechanism, live development is accomplished without modification of the language or the JVM.

A dynamic class's compiled peer works well for interoperability, provided that all changes to a dynamic class are fine-grainedadding and removing methods or fields, and modifying code. However, if the developer changes the position of a dynamic class in the class hierarchy, this may change its set of inherited members and therefore require that the dynamic class's compiled peer be regenerated and reloaded. This thesis addresses how to reloaded classes without modifying the JVM and how to cope with existing instances of dynamic classes whose ancestors have changed.

1.2 Related Work

A great deal of work has been done on providing mechanisms for the reloading or replacement of code in numerous operating systems and run-time environments [1, 2, 3, 5, 7, 14, 16]. Such mechanisms are desired so that large or mission-critical systems can be updated without downtime.

However, none of these solutions meets the unique needs of a live development environment. The majority of systems address the reliability and robustness requirements imposed by live, deployed systems [1, 13]. As a result, these systems require the programmer to provide explicit instructions on how current data should be migrated. The effort required to create migration instructions is justified only for live modification of systems already deployed.

Because JPie is a development environment, rather than a production runtime environment, forcing the developer to specify the migration process for drastic design changes would overshadow any advantages gained from live modification. (The developer would sooner restart the application than have to carefully specify how to upgrade old instances.) Therefore, we seek a fluid development environment that supports maximally live type-safe class hierarchy modification to the extent that there is no extra burden on the developer.

Other solutions to the problem of code replacement involve modifying the language [6] or the execution environment [12, 15]. While it would be possible to create a modified JVM or introduce modifications to the Java language, such approaches are undesirable for JPie. Key goals in the development of JPie have been practicality through the use of the Java language and portability through the use of the standard JVM.

An alternate approach to class reloading without modification of the JVM exists [17]. This approach suggests placing new versions of classes in an alternate package to enable reloading. Not only does this introduce a host of protection issues, it also places a burden on the file system. Additionally, this approach provides no compatibility between old and new versions of classes. Because we are able to utilize the additional level of indirection provided by dynamic classes, our approach is simpler, more compatible, and more extensible than the package-based approach.

1.3 Java's Class Loaders

The Java Virtual Machine (JVM) provides a system class loader. This class loader is responsible for locating and loading classes explicitly referred to by code executing inside the JVM, as well as those requested by the Class.forName method, which allows a class to be loaded by providing its fully-qualified name.

The system class loader will only load a class once, and provides no mechanism for unloading a class. In addition, the system class loader will only search for classes inside the JVM's classpath, and the location of a class's bytecode cannot be explicitly specified. Therefore, the system class loader will simply load bytecode from the first match found.

The Java framework also provides a ClassLoader class. When instances of this class are used for loading classes, the programmer is afforded a little more control. Each ClassLoader instance has a parent ClassLoader (if none is specified at time of

instantiation, the system class loader is used as the parent). An instance may load any class, provided a class with the same fully qualified name has not been loaded by that instance or any ancestor instance. Additionally, a ClassLoader instance may load a class directly from a byte stream, allowing loading of specific files.

The JVM requires that all other classes referred to by a class be loaded by its class loader or one of its class loader's ancestors. (Because the system class loader is an ancestor of all class loaders, it is used to load all referenced classes which are not otherwise explicitly loaded.) Each object has a getClass method which returns a Class instance representing the initial object's class. Each Class object has a getClassLoader method. Therefore, if the same class is loaded by two different class loaders, they will be represented by unique Class instances, and will be unique types.

2 Dynamic Class Loader

We present a dynamic class loader which is at the heart of our framework allowing modification of the class hierarchy. This class loader allows reloading of compiled classes into the JVM through a tree-based technique.

In this section, we first consider the design of the dynamic class loader. Then, we discuss how it is used through an example. Finally, we explain how class versioning automates class reloading in our system.

2.1 Design

The dynamic class loader consists of two classes, `DynamicLoader` and `PeerLoader`. The `DynamicLoader` class presents a static interface for loading and reloading classes. The classes are actually loaded by instances of the `PeerLoader` class.

The `PeerLoader` class is a relatively straightforward extension of Java's built-in ClassLoader class. In this design, each `PeerLoader` instance loads only one class (or, more specifically, one version of one class). Instances of this class are responsible for retrieving bytecode data from a specified file, and loading that bytecode as a specified class name. Additionally, `PeerLoader` instances keep track of version information, as discussed in Section 2.3.

The `DynamicLoader` class manages all of the `PeerLoader` instances and the classes they load. `PeerLoader` instances are constructed in a hierarchy tree mirroring the current class hierarchy. At the top of the `PeerLoader` tree is the system class

loader. All regular Java classes are loaded by this class loader. (Because these classes cannot be modified, they will not need to be reloaded within JPie.) Directly below the system class loader is a special root `PeerLoader`. This `PeerLoader` is not responsible for loading any classes. Instead, it serves as a parent loader for all `PeerLoader`s which load peer classes with non-dynamic parents. Additionally, the root `PeerLoader` is responsible for creating all packages.

Let C be a class, and let parent(C) and loader(C) represent C's parent class, and C's loader, respectively. In order to load C, parent(C) is first determined. (This may be accomplished without first loading C by consulting the dynamic class object for C.) Then, C is loaded using a new `PeerLoader` instance whose parent loader is loader(parent(C)). Finally, the newly loaded class C is then mapped to its fully-qualified name and cached in the `DynamicLoader`. When additional requests for this class are made to the `DynamicLoader`, the cached class is returned.

Reloading a class C is exactly the same as loading the class initially. First, a new `PeerLoader` instance is created whose parent loader is loader(parent(C)). This new loader is then used to load the new version of C. Additionally, the cache in the `DynamicLoader` is updated, so the new version will be returned with each additional request.

An example of the loading and reloading process is given below. We defer discussion of how JPie initiates the reloading of a class to Section 2.3.

2.2 An Example

Suppose the user is designing a process control system which consists of sensors, actuators and control units. Figure 1 presents two possible class hierarchy design choices. Further suppose that the user initially chooses Configuration 1, and begins implementation. Part way through implementation, however, the user decides that Configuration 2 will provide substantial benefits.

We begin by considering how the classes of Configuration 1 are initially loaded. In JPie, only peer classes of dynamic classes will potentially be reloaded. So, all regular compiled classes (here, `Object` and `Thread`, shown in gray) are loaded by the system class loader.

As discussed above, all compiled peers of dynamic classes are loaded by a `PeerLoader` instance, and each `PeerLoader` instance loads only one class (or more specifically, only one version of one class). Each time a load occurs, a new `PeerLoader`

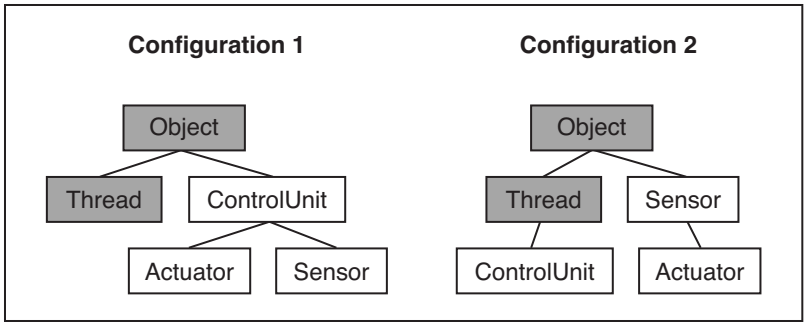

Figure 1: Two possible configurations of a class hierarchy. Compiled classes are shown in gray, dynamic classes are shown in white.

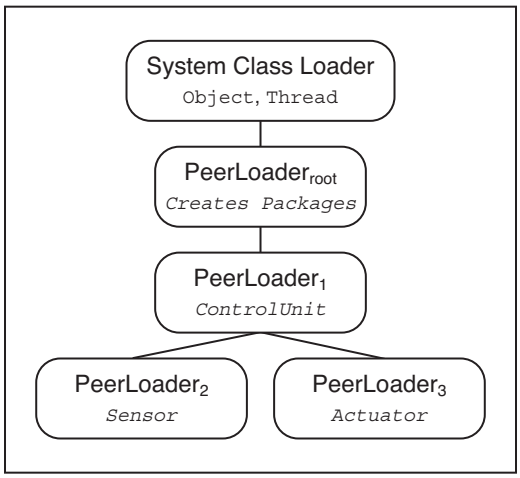

Figure 2: The `PeerLoader` tree configuration after loading Configuration 1.

instance is created. Note that parent(loader(C)) = loader(parent(C)) for all dynamic classes C where parent(C) is a dynamic class. If parent(C) is not a dynamic class (and thus not loaded by a `PeerLoader`), a special root `PeerLoader` is used as the parent. Figure 2 shows the `PeerLoader` tree configuration after loading Configuration 1.

When the user switches to Configuration 2, only classes with modified ancestors must be reloaded according to the process described above. In this example, this includes all dynamic classes shown. The reloading must occur down the hierarchy tree. For example, the compiled peer of the `Sensor` class must be reloaded before the compiled peer of the `Actuator` class. This is necessary to ensure that loader(parent(C)) = parent(loader(C)) for all dynamic classes C. If a class were reloaded before its parent, the JVM would observe that the new parent was not yet loaded, and would load the new parent into the system class loader. As a result, the parent could not be reloaded again.

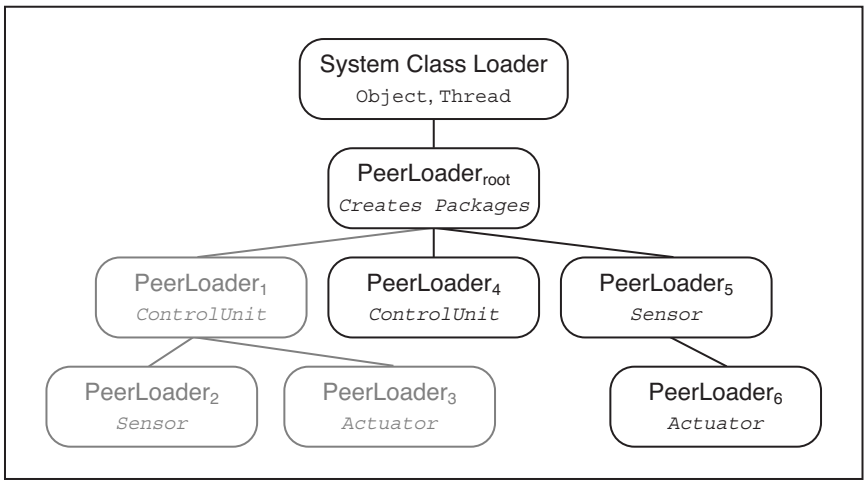

Figure 3: Changes to the `PeerLoader` tree configuration after switching to Configuration 2.

Our loader tree, with one class loader per class, ensures that each time a class C is reloaded we can place its loader at the appropriate place in the tree so that it sees the most recent versions of all the other classes in C's ancestry. Upon transitioning to Configuration 2, new `PeerLoader`s are instantiated to load the compiled peers of the `Sensor`, `Actuator`, and `ControlUnit` classes. Figure 3 shows the new configuration of the `PeerLoader` tree. After each class is loaded, the new compiled peer classes are placed in the `DynamicLoader`'s cache. This completes the reloading process, and all new instances will conform to the user's design change. Note that the `PeerLoader`s which loaded the initial versions of these compiled peers still exist after the classes are reloaded. Thus, old instances can still access the bytecode from the old versions. Section 3 discusses the issues which arise when multiple versions of a class coexist in the system.

2.3 Versioning and Automatic Reloading

Within JPie, each dynamic class has a version. Whenever a course-grain change occurs, such as class hierarchy modification, the version numbers of affected dynamic classes are incremented.

When a dynamic class's compiled peer is loaded by a `PeerLoader`, the current version of the dynamic class is recorded by the `PeerLoader`. When JPie requests a peer class from the `DynamicLoader`, the version of the compiled peer most recently loaded is compared with the current version of the associated dynamic class. If the

version numbers match, the `DynamicLoader` simply returns the cached compiled peer class.

However, modifications to the class hierarchy necessitate the reloading of affected classes' compiled peers. To initiate lazy reloading, the hierarchy modification process will increment the version numbers of affected classes. The next time JPie asks the dynamic class loader for a modified class's compiled peer class, the version discrepancy will be observed. The `DynamicLoader` will then trace the version discrepancies up the class hierarchy until the root `PeerLoader` is reached. The compiled peer with a version discrepancy that is highest in the tree is reloaded first, and then all compiled peers below it are reloaded sequentially by a breadth-first traversal of the tree. This both automates the reloading process and guarantees that all classes are loaded in the appropriate order.

3 Effects of Class Hierarchy Modification in a Live System

Modification to the class hierarchy in a live system has the potential to cause a wide variety of problems, from missing methods to the unexpected execution of old code. These problems may arise in any class which has its ancestry modified. In this section, we first classify the problems which can arise from hierarchy modification. We then discuss how these problems are handled in JPie.

3.1 A Classification of Potential Problems

There are three main effects of hierarchy modification that can lead to problems: the removal of inherited members (fields and methods) from a class, the addition of inherited members to a class, and the difference in types between old and new instances of the class. Recall that each dynamic class has a compiled peer, and that each instance of a dynamic class has two parts: the dynamic instance (which holds the values of the dynamically declared fields) and the compiled peer instance (which holds the values of the fields inherited from compiled ancestors and on which inherited compiled methods are executed). Also recall that the compiled peer's type represents the position of the dynamic class in the class hierarchy. When compiled code holds a reference to an instance of a dynamic class, it does so by holding a reference the compiled peer instance. However, when the position of a dynamic class in the class

hierarchy changes, the compiled peer of that class must change accordingly. When this happens, we say that the previously existing instances are abandoned. In other words, the type of the compiled peer instance is no longer the compiled peer class of its corresponding dynamic class.

Suppose that an ancestor of a dynamic class C changes from A to A'. (That is, either C's parent changes from A to A', or C has some ancestor D whose parent changes from A to A'.) We have three cases: A' is a descendent of A, A' is an ancestor of A, or A' is unrelated to A.

When A' is a descendent of A, inherited members will only be added to C. When A' is an ancestor of A, inherited members will only be removed from C. When A' is unrelated to A, inherited members will be both added to and removed from C. In all three cases, the old and new types of C will differ. So, in all three cases, the existing instances of C will become abandoned.

The distance from C to the closest common ancestor of A and A' gives an indication of the amount of change C undergoes in a hierarchy modification. In the first two cases, this will be the distance to A and A' respectively. However, in the third case (A and A' unrelated) the closest common ancestor will be further away than both A and A'. Loosely speaking, the number of potential problems increases as the distance to the closest common ancestor becomes greater, since the number of classes from which members are inherited and disinherited increases.

3.2 Handling Abandoned Instances in JPie

The JPie philosophy is to aggressively support run-time program modification, even when such changes affect existing instances. In keeping with this philosophy, our goal is to allow abandoned instances to continue to participate in the execution, provided that the fact that they are abandoned cannot be observed. Recall that within JPie, dynamic classes may extend either dynamic or compiled classes, but compiled classes cannot extend dynamic classes. As a result, class hierarchies have a distinct boundary between compiled classes and dynamic classes. This boundary affords us a significant advantage when handling hierarchy modification problems.

We break up the possible hierarchy modifications into two categories: dynamic ancestor changes, and compiled ancestor changes. In the example given in Figure 1, the `Sensor` and `Actuator` classes undergo only dynamic ancestor changes (the only compiled ancestor of these classes, `Object`, remains the same in both configurations).

In contrast, the `ControlUnit` undergoes only a compiled ancestor change, adding `Thread` (a compiled class) to its ancestry.

While we keep these changes isolated in our example for expository purposes, a dynamic class may undergo both types of changes in a single hierarchy modification. However, any hierarchy modification consisting of both dynamic and compiled ancestor changes can be decomposed into a series of hierarchy modifications, each containing only one type of ancestor change.

The next two sections discuss how the problems presented in Section 3.1 are handled for each type of ancestor change.

3.3 Dynamic Ancestor Changes

Dynamic ancestor changes occur when a dynamic class loses and/or gains new dynamic ancestors. In practice, most design changes occur in user-defined classes, rather than in the library classes that support them. As a result, the majority of ancestor changes are of this type. Due to the level of indirection created by the dynamic class system, problems arising from dynamic ancestor changes are easier to handle than those arising from compiled ancestor changes.

All inherited members that are added or removed from a dynamic class as a result of a dynamic ancestor change will be dynamic members. That is, all modified members will be declared within some dynamic class. These modifications, then, are exactly the same fine-grained modifications already supported by dynamic classes [11] in JPie, and will be reflected in both abandoned and new instances. (In JPie, members are used by a reference to the declaration object, not by name matching, so the overriding of methods is not based on the lexical signature of the methods, and fields are not masked by name. Therefore, there is no risk that a newly acquired dynamic member will accidentally override or mask a member declared by a compiled ancestor.)

The compiled peer types of new and abandoned instances will differ as a result of dynamic ancestor changes. However, no problems arise because of this. Because only the dynamic ancestors of new and abandoned compiled peer instances differ, all compiled peer instances will have identical compiled ancestors. References in complied classes can only have compiled types. As a result, compiled classes will not have direct access to the dynamic members which were added or removed. All members directly accessible by compiled class instances will be expressed through both the new and

abandoned compiled peer instances. Furthermore, code contained in compiled classes can only perform type casts to other compiled types. Due to their identical compiled ancestries, both new and abandoned compiled peer instances may be cast identically.

Unlike the type of the compiled peer instances, which are determined and fixed upon creation, the dynamic class system allows the dynamic instance's type to change. Therefore, abandoned instances of a modified dynamic class will automatically express the new type to all instances of dynamic classes.

While dynamic ancestor changes may introduce problems, these problems are dealt with elegantly by the system. As a result, both abandoned and new instances behave identically, and are indistinguishable to the user.

3.4 Compiled Ancestor Changes

Compiled ancestor changes occur when a dynamic class loses and/or gains new compiled ancestors. These changes lead to the addition and removal of inherited compiled members (members declared in compiled classes). In addition, abandoned compiled peer instances will express the wrong type to other compiled instances, a problem that cannot be solved by the ability to modify the type of the dynamic instance.

Suppose class C undergoes a compiled ancestor change, and the user's system contains both abandoned and new instances of C. Code that interacts with these instances may not function properlytype casting may fail, and members may be unavailable. These failures will lead to run-time exceptions, which will be caught by the JPie debugger. However, depending on whether the instance is new or abandoned, these exceptions will mean very different things. We consider each of these cases in turn.

A failure occurring in a new instance is caused by old code that attempts to access a member of C which has been removed (or that attempts to perform a cast to the abandoned C type). This code is incorrect (out-of-date), and must be modified. The error message JPie provides for the offending code will convey the appropriate message to the user, and any attempt to execute the code will pause the execution at that point in the debugger so that the user can correct it.

A failure occurring in an abandoned instance is caused by code that attempts to access a new member of C (or that attempts to cast the abandoned instance to the new type). If we simply allow execution to proceed, the exception thrown in this case could be misleading. The code may in fact be correct, and fail only because it

was executed using an abandoned instance. Our goal is to allow the use of abandoned instances as long as possible, but to prevent execution within an abandoned instance that would expose its type incompatibility. To this end, when a class undergoes a compiled ancestor change, its existing instances are marked as stale. Code is allowed to interact with this class exactly as before. However, any exceptions which occur as a result of stale instances are caught before reaching the debugger, and rethrown as an AbandonedInstanceExcepetion. This exception reveals the true cause of the error to the programmer, and allows the programmer to abort offending threads and resume testing and development with the remaining instances.

Returning to the example shown in Figure 1, we see that new and old instances of `Sensor` and `Actuator` will function identically, because these classes have only undergone dynamic ancestor changes. However, `ControlUnit` has undergone a compiled ancestor change. Any existing `ControlUnit` instances may lead to abandoned instance failures.

In practice, we expect that most hierarchy modifications in a live development environment will occur within the dynamic portion of the hierarchy, and therefore will not result in abandoned instance failures. Furthermore, we expect that programmers would naturally restart the execution after drastic hierarchy changes that could result in such failures. In cases where the programmer does not anticipate abandoned instance failures, the run-time system will call attention to such failures through the exception mechanism, and provide the opportunity to continue executing with the remaining instances.

One might contemplate entirely eliminating abandoned instance failures by adding support for object migration. However, besides placing an unnecessary burden on the developer, comprehensive migration support would require modification of the JVM so that references could be replaced globally [15]. Execution within the standard JVM has been a key goal in the development process of JPie as a whole, and is a constraint in the design of many systems. As a result, the work presented above represents the limit of abandoned instance interoperability that can be achieved under this constraint.

4 Graphical Manipulation of the Hierarchy

JPie provides direct manipulation of graphical representations of programming objects. In order to extend this paradigm, a graphical user interface (GUI) for hierarchy

modification was developed. We briefly present the GUI design and then discuss how the GUI interacts with our class reloading framework to ensure that reloading occurs correctly and efficiently.

4.1 Design of the Class Hierarchy Editor

Within JPie, a programmer's modifications to the system are typically reflected immediately. However, an atomic hierarchy modification often consists of several discrete class ancestry changes. For instance, in the example above, each dynamic class undergoes a parent change. However, the programmer intends the group of modifications to represent one atomic change to the system.

To meet this need, modifications to the class hierarchy must first be specified within the editor, and then committed to the system. The programmer may specify any number of class ancestry changes without affecting the system. Once specification is complete, the programmer commits the changes. The system then determines the most efficient order to update and reload affected classes.

Besides meeting atomicity requirements, the specification and committal process provides several benefits. Foremost, it allows the programmer to experiment with the design without introducing new abandoned instances at the specification of each change. If the programmer chooses not to use all or part of a hierarchy modification before committal, existing instances of the involved classes are not abandoned. Additionally, the committal process allows the system to determine the most efficient way to modify the system so that the user's hierarchy modifications are reflected. A user's thought process may not follow the top-down modification necessary to update the system efficiently. Updating the system after each modification made by the user could result in the wasted creation of many intermediate classes. (This efficiency concern is discussed further in Section 4.3.)

The committal process opens the door for one potential problem, however. Due to Java language constraints, some modifications to the hierarchy are not allowed. For example, final classes cannot be extended. Additionally, the interfaces and ancestors of a class must not have methods with conflicting signatures. (Conflicting signatures arise when two or more methods have identical names and parameter types but different return types, or when the exceptions thrown by a method are incompatible with those thrown by the method it will potentially override.) Without some analysis of a programmer's modifications during the specification process, he or she could proceed

down a design path which is not allowed. Presenting the user with the problems during committal would both confuse the user and go against the grain of immediate feedback central to a live development system.

In our system, this problem is solved through the use of a signature builder. When the user attempts to specify a modification to the hierarchy, the signature builder constructs signatures of each affected class to determine if the modification would violate any language constraints. If such a violation would occur, the signature builder throws an exception which details the type of violation. The GUI does not allow specification of the modification, and provides immediate feedback as to the reason.

4.2 Use of the Class Hierarchy Editor

The class hierarchy editor initially displays no classes. The programmer drags the classes he or she wishes to work with from a class listing into the class hierarchy editor. The representation of classes is shown in Figure 4. Both compiled and dynamic classes may be placed in the hierarchy editor, but modifications may only be made to dynamic classes. Classes which are not involved in a modification may also be removed from view at any time.

In the class hierarchy editor, modifications are made through a set of drag-and-drop actions, called gestures. While a complete discussion of the supported gestures is beyond the scope of this thesis, an example is shown in Figure 4. There, the line connecting a child to its parent class is being dragged to a new parent, specifying an inheritance change. Additional gestures exist for class creation, class extension, and interface implementation.

When a modification would violate Java language semantics (as discussed in Section 4.1), gestures for this modification are disallowed. When the user attempts a disallowed gesture, a visual cue (unique to the gesture) and a message in the editor's status bar explain the reason the gesture is not allowed.

When the new design is complete, the programmer selects an option to initiate the commit process. Alternatively, the user may chose to save the current configuration for later work, or discard the changes entirely.

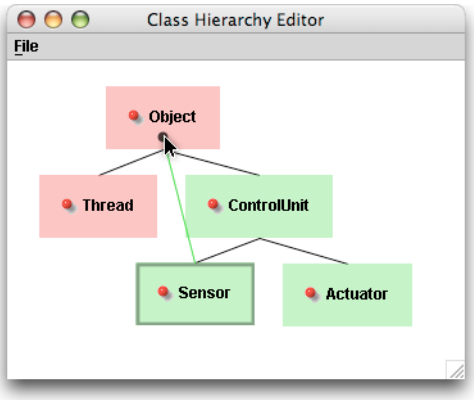

Figure 4: A view of the class hierarchy editor. Here the parent of the `Sensor` class is being modified.

4.3 The Commit Process

The committal process is intended to be an atomic change to the system. However, each affected dynamic class must be modified and its compiled peer reloaded sequentially. To guarantee atomicity, then, execution of dynamic code (and thus, creation of dynamic instances) must be halted during the entire committal process.

The signature builder guarantees that the user's modifications represent a valid Java hierarchy at any point in the editing process. Therefore, the committal process must only be concerned with how to perform the modifications.

A Java class file contains information not only about the class itself, but also about the class's parent. Information about a class's additional ancestors is determined when the class is loaded. Therefore, dynamic classes that undergo any type of ancestor change must be have their compiled peers reloaded after a hierarchy modification, but only dynamic classes that undergo a parent change must have their compiled peers recreated and recompiled.

As discussed in Section 2, the order in which classes are reloaded is important for correctness. The design of the dynamic class loader enforces this order. While the order in which new peer classes are created and compiled is not important for correctness, it is important for efficiency. Suppose dynamic classes A and B have both undergone parent changes, and thus their peer classes must be recreated and recompiled. Further suppose that B is now the child of A. If B's compiled peer is recreated and recompiled before A's compiled peer, the resulting class file will contain information regarding the old version of A. Once A's new compiled peer is created and compiled, B's compiled peer will have to be created and compiled again. This

problem is solved by recreating and recompiling from the root of the new hierarchy downward.

The versioning system implemented in the dynamic class loader (Section 2.3) guarantees that classes will be reloaded when necessary. As a result, the committal process does not need to explicitly force the reloading of affected classes. The committal process, then, consists of three steps:

1. Determine an efficient committal order.

2. Recreate and recompile peer classes which have undergone parent changes in the determined order.

3. Update version numbers of dynamic classes which have undergone ancestry changes, and mark existing instances of those classes as stale.

As stated above, the efficient committal order is determined by a breadth-first traversal of the new hierarchy tree. New peer classes are created and compiled through the same mechanism used to create the initial peer classes. The final step, updating version numbers and marking existing instances as stale, is straightforward and may be done in any order (since execution of dynamic code is halted during committal).

5 Further Applications

The pairing of dynamic classes and the dynamic class loader could be used to provide support for run-time code modification and updating in any system. This improves on prior approaches because it does not require modification of the language or the JVM. It is also simpler to use and maintain than alternate approaches.

If modification of the JVM was allowed in a given application, the use of the dynamic class loader would be even more versatile. By adding a global reference replacement mechanism to the JVM, references to old versions of a class's instances could be replaced with pointers to new replacement instances after reloading and migrating.

6 Future Work

One area of future work is to allow class renaming and package reassignment in addition to inheritance changes. Many of the same underlying reloading techniques will be used to accomplish this.

Another area of future work involves modifying JPie so that the application being developed runs in a separate JVM from JPie itself. This would allow the programmer's application to be killed and restarted without restarting JPie. With this modification, the programmer would have the option of restarting the application after drastic hierarchy modifications, eliminating the possibility of abandoned instance failures.

7 Conclusion

We have presented a class reloading mechanism which supports the ability to modify the class hierarchy in a live development system without modification of the language or the JVM. Live modification of the class hierarchy encourages experimentation, creating a fluid development process lacking the penalties typically associated with making design changes after implementation has begun.

We provide a fluid development environment that supports maximally live type-safe class hierarchy modification to the extent that there is no extra burden on the developer. While drastic changes may require a developer to restart an application, the more frequent kinds of incremental changes and minor class hierarchy design changes can proceed during live execution without the overhead of specifying object migration.

A Implementation Verification

As stated in Section 3.1, we can classify the effects of hierarchy modification into three broad categories. Modifying the class hierarchy can lead to: the removal of inherited members from a class, the addition of inherited members to a class, and differences in types between old and new instances of a class. These effects manifest themselves very differently in compiled and dynamic code.

In this section, we discuss the verification procedure used to insure that all modification scenarios behave as expected. We first explain the test scenario in general, and then we examine each type of effect in detail.

A.1 Test Scenario

A rather complex class hierarchy modification is required to generate all possible effects of a hierarchy modification. The class hierarchy modification we used is presented in Figure 5. Each class is shown with the methods it declares (or overrides).

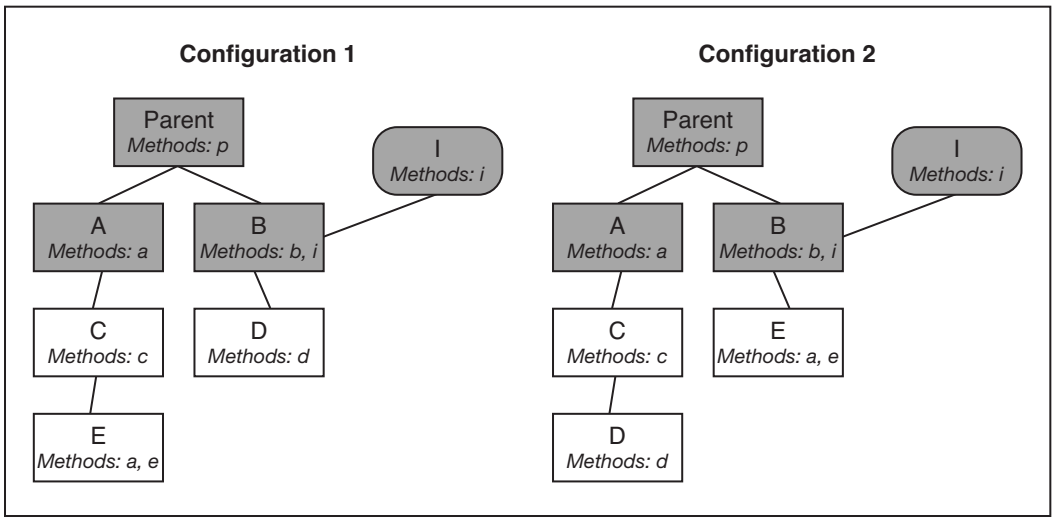

Figure 5: Class hierarchy used for testing, shown before and after modification. Compiled classes are shown in gray, dynamic classes in white. `I` is an interface.

Test Case Justification

This test hierarchy, when coupled with both compiled and dynamic test code, allows us to test all effects of class hierarchy modification. First, because the hierarchy

contains both compiled and dynamic classes, we can ensure that both compiled and dynamic code operate as expected after the modification.

Second, because both `D` and `E` gain, lose, and keep some of their inherited members, we are able to test the effects of each of these changes. (For example, `D` loses the inherited method `b`, gains the inherited method `c`, and keeps the inherited method `p`). It is important to note that the members gained and lost are both compiled and dynamic.

Finally, this hierarchy lets us test two special cases. First, the use of the interface `I` lets us test correct functionality when classes gain or lose inherited interfaces. Second, because `E` overrides an inherited method `a`, we are able to verify the behavior when the inherited method is lost during modification.

A.2 Effects of Hierarchy Modification

In this section, we examine the effects of hierarchy modification to ensure that they are handled correctly within the system. We first consider the effects on compiled code, and then the effects in the dynamic domain.

Effects of Hierarchy Modification in Compiled Code

In order to verify appropriate behavior in the compiled domain, a compiled `Tester` class was created. A portion of the code for this class is given in Figure 6. Essentially, the `test` method takes in a `Parent` object, and determines its true compiled type. The compiled type is printed, and the appropriate methods are called.

Objects of type `D` and `E` created both before and after the hierarchy modification are passed to this test method. As discussed in Section 3.2, we expect instances created before hierarchy modification to behave like their old types when interacting with compiled code (and new instances to behave as their new types.) We choose not to throw `AbandonedInstanceExceptions` within compiled code, since compiled code may not necessarily be written to expect such exceptions.

The only real case of interest here is when compiled code calls dynamic code polymorphically. In our test scenario, this occurs when objects of type `E` are passed to the `Tester` class. New instances of type `E` will be of type `B`, and will have method `b` called. However, compiled code will see old instances of type `E` as instances of type `A`. Because `a` is overridden in `E`, when the inherited method `a` is lost, the user is prompted to chose whether to convert `a` to a normal method in `E`, or to delete it.

```
public void test(Parent p) {
  System.out.println("Tester: p is an instance of Parent - result: "
                     + p.p());
  if (p instanceof A)
    System.out.println("Tester: p is an instance of A - result: "
                       + ((A) p).a());
  if (p instanceof B)
    System.out.println("Tester: p is an instance of B - result: "
                       + ((B) p).b());
  if (p instanceof I)
    System.out.println("Tester: p is an instance of I - result: "
                       + ((I) p).i());
}
```

Figure 6: Implementation of test method for testing compiled code.

If the method is deleted, compiled code will no longer call a method polymorphically on old instances of `E`. Instead, the code declared in `A` will be executed. If, however, `E` contains dynamic code matching the signature of `a` (i.e. the method is converted to a regular method), this code will still be called polymorphically on old instances of `E` when method `a` is called. Should the dynamic code ever be deleted (or its signature modified), the compiled code will simply begin executing the code declared in `A` for old instances.

Effects of Hierarchy Modification in Dynamic Code

Because the user interacts directly with dynamic code, we require far greater support and tolerance for the effects of hierarchy modification. For example, a hierarchy modification may change in a variable's type, which can cause dynamic code to become invalid. Additionally, code written after the modification may not be able to execute on old instances because old instances do not contain the appropriate compiled inherited methods. Finally, method overriding may change as a result of hierarchy modification. All of these issues must be dealt with appropriately.

In order to verify each of these scenarios, a dynamic class `DynamicTester` is used. When instances of this class are created, instances of `D` and `E` are created as member variables. So, we can encapsulate old and new instances of `D` and `E` by creating `DynamicTester` instances before and after hierarchy modification.

There are four cases of interest in the dynamic domain which arise from hierarchy modification. We first consider the effects of losing inherited members due

to hierarchy modification. Dynamic code accessing these members should become invalidated. To test this, we write code in `DynamicTester` which calls methods that will be removed (for example, code which calls method `b` on an instance of `D` could be used). Then, the hierarchy is modified, and we certify that the code is invalidated with the appropriate error message.

Next, we consider the effects of losing inherited members on overriding members in dynamic code. For example, `E` loses the inherited member `a`, which it overrides, during the hierarchy modification. Assuming `E`'s implementation of `a` contains no `super` calls, the dynamic code may still be syntactically valid. However, it may have a completely different semantic meaning. So, when performing the hierarchy modification, the system recognizes this, and asks the user if she wants to remove the method or convert it to a normal method.

Additionally, we consider the effects of gaining inherited members as a result of hierarchy modification. If these members are dynamic, they should be accessible to both new and old instances of the dynamic class. (Consideration of new compiled members is given below.) For example, both new and old instances of `D` should be able to access the inherited method `c`. We test this by writing code in `DynamicTester` which calls this method, and test the code on both old and new instances.

Finally, we consider the effects of gaining *compiled* inherited members. Because these members are accessed through the dynamic class' compiled peer instances, these members will be unavailable in old instances of dynamic classes. While we are able to write dynamic code which accesses these members, the code will not execute on old instances. Instead, `AbandonedInstanceException`s will be thrown, notifying the user that the code is valid, but cannot be executed on old instances. New instances, however, will be able to access the compiled inherited members as expected.

A.3 Conclusion

Using the test scenarios presented, we have exhaustively considered the effects of hierarchy modification. The majority of hierarchy modifications are well supported within the system. When the effects cannot be easily supported (i.e. without migration of data to new instances), the execution fails gracefully. We feel that this guarantees a quite sufficient amount of functionality for the user, while avoiding the burdens of data migration. This issue is discussed further in Section 1.2.

References

[1] Miles Barr and Susan Eisenbach. Safe upgrading without restarting. In *Proceedings of the International Conference on Software Maintenance*, pages 129–37, September 2003.

[2] T. Bloom. *Dynamic module replacement in a distributed programming system.* PhD thesis, Massachusetts Institute of Technology, 1996.

[3] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *Proceedings of the 18th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 403–14, October 2003.

[4] T. M. Breuel. Implementing dynamic language features in Java using dynamic code generation. In *Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*, pages 143–52, July 2001.

[5] Sean Dorward, Ravi Sethi, and Jonathan E. Shopiro. Adding new code to a running C++ program. In *Proceedings of the Usenix C++ Conference*, pages 279–92, April 1990.

[6] Sophia Drossopoulou, Mariangiola Dezani-Ciancaglini, Ferruccio Damiani, and Paola Giannini. Objects dynamically changing class. unpublished, August 1999.

[7] Jonathan J. Gibbons and Michael J. Day. Shadows: A type-safe framework for dynamically extensible objects. Technical Report TR-94-31, Sun Microsystems, November 1994.

[8] Kenneth J. Goldman. A demonstration of JPie: An environment for live software construction in Java. In *In the Conference Companion, 18th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 403–14, October 2003.

[9] Kenneth J. Goldman. A concepts-first introduction to computer science. In *In ACM SIGCSE Technical Symposium on Computer Science Education*, pages 432–36, March 2004.

[10] Kenneth J. Goldman. An interactive environment for beginning Java programmers. *Science of Computer Programming*, 53(1):3–24, October 2004.

[11] Kenneth J. Goldman. Live software development with dynamic classes. Submitted for publication, August 2004.

[12] Stéphane Hillion. DynamicJava. http://koala.ilog.fr/djava/, 1999.

[13] Gísli Hjálmtýsson and Robert Gray. Dynamic C++ classes–a lightweight mechanism to update code in a running program. In *In proceedings of the USENIX Annual Technical Conference*, pages 65–76, June 1998.

[14] Robert Laddaga and James Veitch. Dynamic object technology. *Communications of the ACM*, 40(5):36–38, March 1997.

[15] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *ECOOP*, pages 337–61, June 2000.

[16] Mark Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. In *IEEE Software*, pages 381–404, March 1983.

[17] John Zukowski. Unloading and reloading classes. Technical report, Sun Microsystems, 2003. http://java.sun.com/developer/JDCTechTips/2003/tt0819.html.