Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCSE-2004-81

2004-08-24

Live Software Development with Dynamic Classes

Kenneth G. Goldman

Software modification at run-time can facilitate rapid prototyping, streamline development and debugging, and enable interactive educational programming environments. However, supporting live fine-grain program modification while reaping the benefits of a compiled type-safe language is a challenging problem. This paper presents fine-grain dynamic classes that support live object-oriented software development in which a program can be modified during execution. We present an implementation of dynamic classes in Java that does not require modification of the Java Virtual Machine. Our implementation supports full interoperability between instances of dynamic classes and compiled classes, including polymorphism, with minimal overhead. Changes to dynamic classes,... Read complete abstract on page 2.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation
Goldman, Kenneth G., "Live Software Development with Dynamic Classes" Report Number: WUCSE-2004-81 (2004). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/1052

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/1052

Live Software Development with Dynamic Classes

Kenneth G. Goldman

Complete Abstract:

Software modification at run-time can facilitate rapid prototyping, streamline development and debugging, and enable interactive educational programming environments. However, sup-porting live fine-grain program modification while reaping the benefits of a compiled type-safe language is a challenging problem. This paper presents fine-grain dynamic classes that support live object-oriented software development in which a program can be modified during execution. We present an implementation of dynamic classes in Java that does not require modification of the Java Virtual Machine. Our implementation supports full interoperability between instances of dynamic classes and compiled classes, including polymorphism, with minimal overhead. Changes to dynamic classes, such as the declaration of instance variables and methods, as well as the modification of statements and expressions within method bodies, take immediate effect on existing instances of those classes. We describe benefits of using dynamic classes in the context of a tightly integrated development environment.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2004-81

Live Software Development with Dynamic Classes

Authors: Goldman, Kenneth J.

August 24, 2004

Abstract: Software modification at run-time can facilitate rapid prototyping, streamline development and debugging, and enable interactive educational programming environments. However, supporting live fine-grain program modification while reaping the benefits of a compiled type-safe language is a challenging problem. This paper presents fine-grain dynamic classes that support live object-oriented software development in which a program can be modified during execution. We present an implementation of dynamic classes in Java that does not require modification of the Java Virtual Machine. Our implementation supports full interoperability between instances of dynamic classes and compiled classes, including polymorphism, with minimal overhead. Changes to dynamic classes, such as the declaration of instance variables and methods, as well as the modification of statements and expressions within method bodies, take immediate effect on existing instances of those classes. We describe benefits of using dynamic classes in the context of a tightly integrated development environment.

Live Software Development with Dynamic Classes

Kenneth J. Goldman *
Department of Computer Science and Engineering
Washington University in St. Louis
kjg@cse.wustl.edu

August 24, 2004

Abstract

Software modification at run-time can facilitate rapid prototyping, streamline development and debugging, and enable interactive educational programming environments. However, supporting live fine-grain program modification while reaping the benefits of a compiled type-safe language is a challenging problem. This paper presents fine-grain dynamic classes that support live object-oriented software development in which a program can be modified during execution. We present an implementation of dynamic classes in Java that does not require modification of the Java Virtual Machine. Our implementation supports full interoperability between instances of dynamic classes and compiled classes, including polymorphism, with minimal overhead. Changes to dynamic classes, such as the declaration of instance variables and methods, as well as the modification of statements and expressions within method bodies, take immediate effect on existing instances of those classes. We describe benefits of using dynamic classes in the context of a tightly integrated development environment.

1 Introduction

Most software is built incrementally, starting from a prototype that gradually evolves through the addition of features and details. Even for small programs, progress from the specification to a working implementation is rarely achieved in a straight-line path. We generally make a first attempt and then gradually mold the software to suit our needs. Throughout this process, productivity is hampered by the edit-compile-test cycle. When making a series of small changes to the software, precious time is lost each time the program must be recompiled and restarted. Even if compilation is relatively fast, it often takes significant time to bring the program to back to the desired state in order to test each change.

Replacing the edit-compile-test cycle with live software development has the potential for significant advantages. In addition to enhancing productivity, it can enable a more fluid approach to software development. In live software development, small program changes are uneventful. Exception handling code can be added at run-time without terminating execution. Even partially completed classes are instantiable and have executable methods.

However, live program modification presents some significant challenges, especially in the context of a type-safe compiled object-oriented language. At the core of these challenges is support for run-time modification of class definitions, such that those modifications take immediate affect

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

on existing instances. Live software modification also raises questions about concurrent execution and modification, as well as accommodation of transient inconsistencies that inevitably occur as a program is being modified. This paper addresses the problem of live program modification with fine-grain dynamic classes that support object-oriented software development in which programs can be modified during execution.

We begin, in Section 1.1, with background on run-time type information and dynamic class loading. Section 1.2 outlines our design goals. Section 2 provides an overview of the dynamic class implementation. Section 3 explains how we achieve interoperability of dynamic and compiled classes, and Section 4 presents the execution details, including a discussion of execution overhead and concurrency control options for live program modification. In Section 5, we discuss the implications of dynamic classes for development environments, including and a brief description of how dynamic classes are supported in our own development environment, JPie.

1.1 Background

We are concerned with both access to and modification of class definitions at run-time. Run-time access to class definitions enables writing simple queries, such as testing the type of an object, as well as the construction of more complex software systems that use classes unknown to them at compile time. For example, Java [1] provides a class named 'Class' whose instances embody information about each class that is loaded into the Java Virtual Machine. In conjunction with classes in Java's reflection package, the class Class provides run-time access to objects of type Field, Method, and Constructor that describe the fields, methods, and constructors of a class. These classes not only permit the discovery of type information, such as the name of a field or the parameter types of a method, but they also permit operations with side-effects, such as setting the value of a field or calling a method.

Discovery and use of type information at run-time enables construction of a class of applications that we will refer to as *semi-interpreted*. In such programs, parts of the execution take place directly as compiled, while other parts of the execution take place under the control of algorithms that access data structures containing information about when to assign values to fields, call methods, and invoke constructors through the run-time type facility provided by the language. As a typical example, consider a graphical user interface (GUI) builder that keeps a mapping of user events (such as button presses) to methods that should be called when those events occur [13, 22]. Modifying the mapping at run-time results in corresponding changes to the semi-interpreted execution. Relationships in a GUI builder are fairly stylized, but more general forms of semi-interpreted execution are possible, as we will describe in Section 4.

For live object-oriented software development, we are concerned not only with run-time access to information about class definitions, but also with run-time modification of those definitions. In a sense, we want to support dynamically changing types (such as in SELF [5] and Smalltalk [9]), but with the safety of (and interoperability with) a strongly typed language.Compiled languages, as a whole, do not provide such support. Even languages like Java, which provide extensive support for access to type information at run-time, are built on the assumption that a class definition is fixed at compile time. Java's class 'Class' and the associated reflection classes are immutable, meaning that they do not provide any methods to modify their state. For example, although a running application can retrieve a list of methods declared by a class, there is no support for adding a new method to that list or removing an existing one. Similarly, there is no support for changing the parameter list, add a statement, or to modify an expression within an existing method. In general, compilation is necessary to change a class definition.

In spite of the compilation step, there are compelling reasons for modifying a program while it

is running. For example, there has been extensive work on dynamic class reloading, primarily for the purpose of hot-swapping classes to install software upgrades or to perform code maintenance or migration [6, 8, 16, 19, 20, 23]. One critical issue is how to address existing instances of the dynamically reloaded classes [15]. Possibilities include: waiting (indefinitely long) for old instances to be deleted or become garbage before loading the new version of the class; allowing two versions of the class to coexist, with old instances continuing to use the old version; or switching old instances over to the new version, which may involve a change to the internal representation of the object. The last of these options allows class modifications to affect existing instances, but this benefit comes at the expense of modifying the language and/or run-time system [17]. Fully interpreted approaches, such as the Dynamic Java interpreter [14] lend themselves to on-the-fly integration of new code, but they are not designed to support fine-grained changes to the program text, particularly for already instantiated objects, once the code is loaded and running.

Prior work on live program modification has concentrated on atomic class reloading, replacing entire classes (or sets of classes) to support deployment of patches or upgrades to existing code during execution [2, 3, 19]. Class reloading systems handle the problem of course-grain program modification within a running program, after the replacement code has been developed and tested. However, the present work is concerned with fine-grain incremental code modifications of the kind that occur while software is being developed. One could imagine a brute-force approach in which each modification of a running application is achieved by recompiling and replacing a class using one of the class replacement techniques already discussed. However, class recompilation and replacement at each and every edit would take its toll on the development environment's response time. Moreover, such replacement could be carried out only when the program is in a sufficiently consistent state to compile, thus placing restrictions on its benefits during incremental software development. Finally, class replacement techniques that require modification of the language or run-time system to handle existing instances limit the portability of any development environment that depends on them.

Therefore, we depart from the approach of dynamic class reloading and present an alternative that is designed for live fine-grain class modification. The key to our approach is the concept of class that is truly dynamic, a class that can be mutated at run-time in order to change its interface and implementation. At the same time, we want these dynamic classes and their instances to fully interoperate with traditional compiled classes and their instances. This leads to our specific design goals.

1.2 Design Goals

With live software development as our motivation, the chief contribution of this work is simultaneous realization of the following design goals.

- dynamic classes: The internal representation, interface, and implementation of a class may be modified dynamically at run-time without recompilation.
- liveness: Dynamic class modifications immediately affect existing instances.
- transparency: Dynamic classes have full access to compiled classes and their instances.
- interoperability: Instances of dynamic and compiled classes may refer to each other.
- hierarchical integrity: Dynamic classes may be subclasses of compiled classes and may override their methods.

- polymorphism: Instances of compiled classes may call methods on instances of dynamic classes polymorphically.
- portability: The underlying programming language and its run-time system are not modified.

Throughout the paper, we describe how our design and implementation of dynamic classes achieves the above design goals.

2 Dynamic Classes

The fundamental goal of this research was to support the notion of a dynamically modifiable class, or *dynamic class*, whose members (fields, methods, and constructors) can be added, removed, and modified at run-time. This section explains how dynamic classes are represented, including class members (i.e., fields, methods, and constructors) and executable statements. We also describe how dynamic class type information is integrated with the existing run-time type information provided by the programming language. This provides background for Section 3, which discusses on the interoperability with compiled classes. Run-time handling of dynamic class modification and their implications for object instantiation, field initialization, and method execution are discussed in Section 4.

2.1 Internal Representation of Class Members

We represent each dynamic class as an instance of the class DynamicClass. Each DynamicClass object holds a reference to the parent (direct superclass) of that dynamic class (which may be either a dynamic or a compiled class) and a list of interfaces that it implements. Each DynamicClass has mutable data structures that represent its members. Programmer actions that effect changes to the dynamic class result in corresponding updates of this representation.

Each DynamicClass object has a mutable list of its declared methods, which are represented as DynamicMethod objects. When the programmer declares (or deletes) a method, a DynamicMethod object is simply inserted into (or deleted from) the list. In turn, each dynamic method is represented as an object whose mutable data structures contain its name, modifiers, return type, parameter list, local variables, body statements, a return statement (for methods whose return type is not void), and a list of thrown (checked) exceptions. Each programmer action that modifies the method, such as declaring a parameter or editing an expression, results in a corresponding change to the dynamic method's data structures. Implications of these changes for program execution are discussed in Section 4.

When a method is designated as overriding a method from a parent class, or implementing a method of an interface, a reference to the overridden or implemented method is kept in the DynamicMethod object. In such cases, the name and parameter list of the method are immutable, and compatibility restrictions are placed upon the access modifier and list of thrown exceptions. Whenever the name or parameter list of the overridden method is changed, the overriding method's name or parameter list is correspondingly updated.

The fields and constructors of a dynamic class are represented as lists of DynamicField and DynamicConstructor objects. A dynamic field's mutable representation includes its name, modifiers, type, and initial value expression. For uniformity, all fields refer to objects, so primitive types (int, double, etc.) are handled using Java's corresponding wrapper classes (Integer, Double, etc.). The representation of dynamic constructors is similar to that of dynamic methods, except that they

each have a "super" call to the parent constructor (or a "this" call to another constructor of the same dynamic class), and they do not have a return statement.

For programmer convenience, a DynamicClass automatically declares assessors (get and set methods) whenever the programmer declares a new field of the class. In accordance with standard Java programming practice, the get method's return type matches the return type of the field and it returns the value of the variable. The set method's return type is void, it has a single parameter whose type matches that of the field, and its body assigns the parameter value to the field. For consistency, the accessors' parameter lists are immutable and their names are automatically kept consistent with the name of the associated instance variable. (For a variable named foo, the names are getFoo and setFoo..) The accessor methods are otherwise mutable, including their bodies and return statements, and they may be deleted.

Within statements and expressions, each use of a field, method, or constructor is handled as a reference to the corresponding DynamicField, DynamicMethod, or DynamicConstructor object. Therefore, the textual identifiers of fields and methods are not consulted as part of the execution and serve only as a documentation convenience to the programmer. As a result, identifiers may be freely changed by the programmer, and the system ensures that each use of the field or method is kept consistent. Similarly, each method call's parameter list is updated whenever the corresponding method's parameter list is changed.

2.2 Internal Representation of Executable Statements and Expressions

The internal representation of a dynamic class is sufficient not only to describe its interface, but also to instantiate it (by executing its dynamic constructors), as well as to execute methods on its instances in a semi-interpreted fashion, described below. The statements and expressions that form the initialization statements and method bodies of a dynamic class are represented as objects that provide an *execute* method that takes an environment as a parameter. The environment, which binds variables to values, may optionally have a parent environment. For example, a stack frame is an environment that binds parameter values to their formal parameters and whose parent environment is the binding table for the target object on which the method was called. The execute method is responsible for carrying out the work of that statement or expression. In the case of an expression, the execute method's return value is the result of the expression.

Control constructs: As the execute method runs, it may encounter other statements or expressions whose execute methods are run, in turn, to produce the overall execution of the program. For example, the body of a method consists of a SequentialBlock object whose execute method traverses a list of statements, calling their execute methods in sequence. A conditional (if) statement first executes its test expression, and then, if the result is true, executes the consequent. By providing the programmer with a full complement of statement types that correspond to the control constructs of the language, and by providing mechanisms for nesting these, we can support the construction arbitrary programs whose execution is carried out by traversing the data structures that represent these nested constructs.

Method calls: Each method call within a statement or expression is represented as a MethodCall object with its own execute method. The object keeps a reference to the method being called, an expression that evaluates to the target of the method call, and a mapping from the formal parameters of that method to the corresponding actual parameter expressions for the call. The MethodCall's execute method begins by evaluating the target expression to determine the target. Next, it evaluates each of the actual parameter expressions within the supplied environment. Then, if the method is a compiled method, it is invoked by reflection. On the other hand, if a dynamic method is being called, then its execute method is called. In either case, the target object and

actual parameter values are provided. Finally, the return value of the method invoked is passed along as the result of the MethodCall's execute method.

Variable access and assignment: Each variable read is represented as a VariableReadExpression object that contains a reference to the variable and (optionally) a target expression for accessing fields of other objects. If there is no target expression, the variable is assumed to be a local variable, parameter or instance variable within the environment in the scope of the expression. The execute method of a VariableReadExpression first evaluates the target expression (if any) to determine the target object and then invokes the get method on the field. Note that if the variable being accessed is declared in a compiled class, then the get method is invoked on the corresponding Field object by reflection. Variable assignment is handled similarly, with the set method being called on the field. A CastExpression is also available when a variable or expression must be treated as an instance of a specific type for the purposes of assignment, parameter passing, or method invocation.

Constant expressions: Constants are represented as ConstantExpression objects whose execute methods simply return the value of the constant as an object. (Recall that all primitive values are represented using their corresponding wrapper classes.)

Erroneous Statements and Expressions: Because of the stylized nature of program construction using the various control constructs and expressions described above, syntax errors are not possible. However, type mismatches are necessarily possible. For example, when forming an expression that expects an integer, one might first reference a list variable and then call its size method. Along the way to constructing the expression, there will be some time during which the expression is empty, and then a time when it contains only the list variable. During those times, the expression will not be type safe for execution. Therefore, certain expressions (namely, default value (initialization) expressions, return statements, and actual parameters), have an expected type, which is compared to the actual type of the expression to determine if the expression is legal. In addition, a variable access or method call is illegal if the associated variable or method has been deleted. The handling of illegal expressions during execution is discussed in Sections 4.5 and 5.2.

Exceptions and Exception Handling: Statements and expressions in dynamic classes may throw exceptions just as those in a compiled program. Furthermore, exceptions that originate in compiled code may be handled by statements within a dynamic class. Any statement or block may be placed within a TryCatchBlock, whose execute method first executes that statement or block, and then, if an exception occurs, looks through its list of handled exceptions for the first matching exception, binds that exception to a variable in the environment and executes the execute method of the associated handling block, which may make use of the variable. The exception handling mechanism is fully exploited by the debugger to permit on-the-fly exception handling, as discussed in Section 5.3.

Transparent program representation: The distinction between our program representation and execution technique and the concept of intentional programming [21] bears mention at this point in the discussion. Although the concept is apparently still evolving, our current understanding of intentional programming is that a programmer specifies various facets of a program in somewhat loosely-coupled way. These so-called intentions are stored in a program database and subsequently reduced to an executable unit through a process akin to compilation. Like intentional programming, we maintain a structured representation of the program. However, in our work the programming model is much more direct. The programmer specifies exactly how the program should execute using a traditional object-oriented programming model with typical control-flow constructs. Furthermore, our structured representation is kept in a directly executable form that exactly mirrors the programmer's specification. This keeps the execution model completely transparent to the programmer. Also, it eliminates the need for reduction or compilation, thereby permitting live

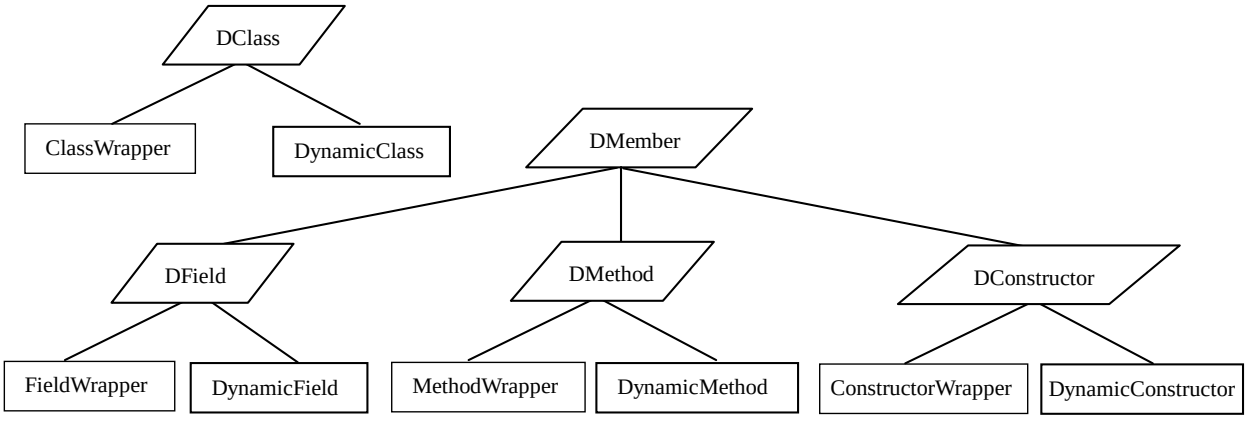

Figure 1: An integrated compiled and dynamic class reflection hierarchy.

program modification since each programmer action results in an immediate corresponding change to representation of the executing program.

2.3 A Unified Type System

Java's immutable class 'Class' provides access to compiled type information at run-time. Our DynamicClass serves as a mutable version of the class Class. Therefore, the cleanest and simplest approach to creating a unified type system would have been to create DynamicClass as a subclass of Class. Similarly, it would have been ideal to create DynamicField, DynamicConstructor, and DynamicMethod as subclasses of Java's corresponding reflection classes Field, Constructor, and Method. This would have allowed the dynamic and compiled classes to be treated polymorphically not only within the development environment, but by the JVM as well. But for various reasons, Java's class Class and its associated reflection classes are declared 'final,' so they cannot be extended. Because of this, it would appear at first glance that creating classes that can be dynamically modified at run-time would be impossible in Java without modification of the language (as in [7] or the virtual machine (as in [17] or the class loader (as in [19]). However, in the interest of portability, we wanted to support dynamic classes using the standard Java language and JVM.

Our solution was to devise a unified type system consisting of two sets of parallel classes, one acting as wrappers for instances of Java's Class and its associated reflection classes, and the other serving as our corresponding dynamic (mutable) versions. Based upon these parallel hierarchies, we created a hybrid execution environment in which instances of dynamic and compiled classes could coexist.

To more easily support transparency and interoperability within the implementation of the execution environment, we wanted to integrate Java's reflection classes with ours in a single class hierarchy. That way, each type in the system would be treated uniformly, whether it was actually a compiled or dynamic type. As ancestors of each pair of the parallel class hierarchies for compiled and dynamic classes, we created abstract classes with appropriate access methods to enable polymorphic treatment of both. These abstract classes are shown as DClass, DMember, DField, DMethod, and DConstructor in Figure 1.

This integrated class hierarchy provided the desired polymorphism for the treatment of types within the implementation of the development environment. However, additional work was needed

in order to "trick" the JVM and, therefore compiled classes, into treating instances of dynamic classes as if they were instances of compiled classes. That work is described in the next section.

3 Interoperation with Compiled Classes

Achieving complete interoperability between compiled and dynamic classes, with a minimum of overhead during execution, was one of the most important and difficult challenges in designing the system architecture. Besides encapsulation, a main advantage of object-oriented software development is support for inheritance and polymorphism. Therefore, it was incumbent upon us in the design of this environment to allow methods to be overridden by dynamic classes.

More specifically, we wanted the ability to override a method in a dynamic class at run-time. Furthermore, we wanted instances of compiled classes to be able to hold references to instances of dynamic classes and treat them just as if they were instances of compiled classes. For example, suppose X is an instance of dynamic class named 'Child' that happens to be a subclass of compiled class named 'Parent.' Furthermore, suppose that some instance Y of an arbitrary compiled class has a reference to the instance of Child, and that this reference is held (polymorphically) in a variable of type Parent. Now, if the program is modified so that Child overrides a method defined in Parent and then Y calls the method polymorphically on X, we want the method declared within dynamic class Child to run, even though Parent's method would have run before the programmer made the modification to override the method. Furthermore, if the programmer subsequently deletes the overriding method in Child, we want future calls in the same execution to revert to the inherited method defined in the compiled Parent class.

The remainder of this section describes this implementation in further detail. We begin with the relationship between the dynamic and compiled peer instances. Then we discuss precompilation of the compiled peer, how polymorphism is supported, and a method caching mechanism to enhance run-time performance.

3.1 Dynamic and Compiled Peers

To achieve interoperability, compiled classes must be able to treat instances of dynamic classes just as if they were instances of a compiled class. Consequently, each dynamic class has a compiled peer class that represents the type of the dynamic class within the Java Virtual Machine. Similarly, each instance of a dynamic class has an associated peer instance that is an instance of the compiled peer class. When instances of dynamic classes present themselves to instances of compiled classes, they do so by providing a reference to their compiled peer instance, which keeps a reference to the dynamic instance and has callbacks into that dynamic instance for execution of dynamically modifiable methods.

The compiled peer, whose name matches that of the dynamic class, is precompiled when the dynamic class is first created, as described further in Section 3.2. If the dynamic class is to extend a class and/or implement interfaces, it is the compiled peer that captures this relationship. (Throughout this paper, we assume that the parent class and implemented interfaces do not change dynamically.)

The compiled peer class is responsible for handling those methods that are called by compiled classes and that may be dynamically overridden, but the compiled peer need not be aware of any additional methods that are declared in the dynamic class, because no compiled class would be able to call those polymorphically. Therefore, the compiled peer can be precompiled as soon as the parent class and implemented interfaces are known. No further compilation is necessary to support subsequent modifications of the dynamic class, so that each fine-grain edit can take immediate

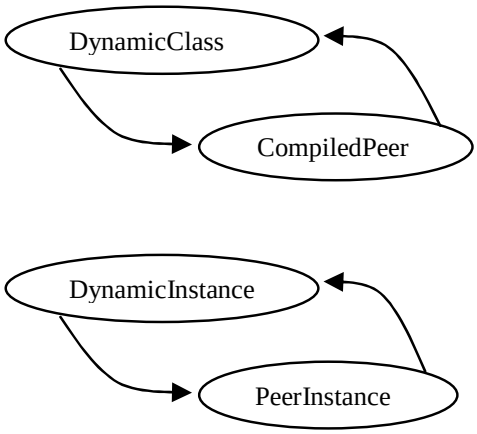

Figure 2: Mutually referential dynamic and compiled peers.

effect on the executing program. A second responsibility of the compiled peer is to act as a proxy on behalf of the dynamic class in order to use reflection to invoke or access (protected) methods and fields that are defined in ancestor classes and accessible to only to subclasses. This is because although the dynamic class represents a descendant of those classes, only the compiled peer is technically a descendant in the Java class hierarchy, and so only instances of the compiled peer are permitted to access those methods and fields.

An interesting technical problem related to the instances of compiled and dynamic peers is that they must be mutually referential. Normally, mutual references between two objects are achieved by fully constructing one object, perhaps within the constructor of the other, and then making the requisite assignments to instance variables so that each refers to the other. With this approach, the constructor of at least one of the two objects is completely finished before the mutual reference is established. However, in the case of compiled and dynamic peers, the mutual reference must be established *prior* to the execution of both constructors: Even during execution of the constructor, the compiled peer may call a method that is overridden by the dynamic class and therefore must execute within the dynamic peer in order to access the required state information. Similarly, code within the constructor of the dynamic class may reference methods or variables inherited from the compiled class, and therefore require access to the compiled peer object.

To establish mutual reference prior to constructor execution, we statically associate a PeerKeeper object with each compiled peer class. This PeerKeeper object provides synchronized methods to store and retreive a reference to an instance of the dynamic peer class. Furthermore, throughout the code of the compiled peer, each use of the dynamic peer instance is accomplished by calling a "getDynamicPeerInstance" method that checks to see if the reference to the dynamic peer instance has already been initialized: if so, it simply returns it; if not, it initializes the reference by obtaining it from the PeerKeeper, passes itself to a method of the dynamic peer instance to establish itself as the compiled peer instance (before any overridden methods are called that might need to refer to inherited members), and then returns the peer dynamic instance.

With this architecture, an instance of a dynamic class is created as follows: First, the dynamic class instance is created and assigned to the PeerKeeper for the compiled peer class. Then, the dynamic constructor is executed on that object, beginning with the base constructor call that constructs the compiled peer instance. During the execution of the compiled peer's constructor, any

methods overridden by the dynamic class can be called on the dynamic instance through the PeerKeeper as needed. Following execution of the base constructor, the reference to the compiled peer instance is established in the dynamic instance, and the remainder of the dynamic constructor is executed. Finally, in case the compiled peer instance did not need the dynamic instance during constructor execution, the "getDynamicPeerInstance" method is called on the compiled peer instance to guarantee initialization of the dynamic instance reference. To ensure that each compiled peer instance is assigned a unique dynamic instance, the PeerKeeper waits inside the setPeer method until its getPeer method has been called. (Otherwise, two different compiled peer instances might be assigned the same dynamic peer.) Since the dynamic peer calls the setPeer method exactly once and the compiled peer instance calls the getPeer method exactly once, PeerKeeper synchronization ensures consistent mutual references and a one-to-one correspondence between dynamic and compiled peers.

3.2 Precompilation

Our goal of "hierarchical integrity" requires that dynamic classes can extend not only other dynamic classes, but compiled classes as well. Furthermore, interoperability demands that when we pass an instance of a dynamic class to a compiled method, the JVM must recognize that object as an instance of the compiled ancestor class. At the same time, polymorphism requires that the dynamic class, of which this object is an instance, be able to selectively and dynamically override methods of the compiled ancestor at run-time, with the changes taking place immediately on every instance without recompilation.

This means that each instance of a dynamic class must not only inhertit, but also be able to override, methods declared in its compiled ancestors. Consequently, when we generate the compiled peer for a subclass of a compiled class, we emit code that overrides every inherited method and implements every interface method that, when called, forwards execution into the dynamic peer instance or calls the inherited (super) method as appropriate. This emitted code uses the PeerKeeper scheme outlined above to obtain and initialize the reference to the dynamic peer. (When the superclass of a dynamic class is also a dynamic class, the compiled peer still acts as the type definition within the virtual machine, but precompilation is simplified since overriding of compiled methods has already been handled within the highest dynamic ancestor.)

When a dynamic class extends a compiled class, only the compiled peer class (not the dynamic class) is technically a child of the compiled parent. Consequently, instances of the dynamic class do not have direct access to protected fields and methods of the parent. However, for both transparency and hierarchical integrity, the dynamic class must appear to the programmer as if it is a true decendant of its compiled parent, we must provide access to these protected members. Therefore, when we generate the compiled peer, we emit methods for accessing the inherited compiled fields and methods by proxy. These are used internally by the dynamic instance so that it appears to have direct access to these members.

Run-time compilation of classes during software development is not a new idea. For example, others have used run-time compilation of classes to dynamically create "glue" classes that conform to interfaces and delegate their work to target instances of compiled classes that do not necessarily implement all the methods of the interface [4]. If we think of the compiled peer as the "glue" and the instance of the dynamic class as the target, one may view our precompilation mechanism as taking this concept to its logical extreme in two important ways. First, we support not only interface conformance, but full inheritance as well. Second we allow the set of methods implemented in the classes of the target objects to be changed dynamically while the program runs.

3.3 Support for Polymorphism

Once the compiled peer has been created, the stage is set for interoperable polymorphism between instances of compiled and dynamic classes. During execution, all references to dynamic instances are passed as references to the compiled peer instance, which in turn invokes methods on the dynamic instance as needed. Since the dynamic peer is an instance of the proper type, inheriting all of the ancestor methods, compiled classes can call methods on these instances as they normally would. Whenever the dynamic class overrides a method, the compiled peer invokes that method on the dynamic peer object, thus enabling polymorphic execution across compiled and dynamic classes.

3.4 Method Caching

Since a dynamic class may override only a small fraction of its inherited methods, we want execution of the remaining methods to proceed with a minimum of overhead. When a method is invoked on a compiled peer instance, it must determine whether or not the method has been overridden in the dynamic class. One approach would be to query the dynamic class on each invocation to see if there is a matching declared method. However, searching for a match would add significantly to the overhead of every method call, even those that are not overriden by the dynamic class.

To streamline execution and avoid repeated searches for overridden methods, the compiled peer class caches of method references as follows. Suppose that a given dynamic class extends a compiled class and inherits n methods available for overriding. During precompilation, each of these methods is assigned an index (0 to $n-1$). The emitted code for each method contains its index, as a constant literal, and uses that index for accessing two static arrays of length n that are declared in the compiled peer. The entry at index i in the first array is a boolean value that indicates whether or not to use the super (inherited) method for method i. The entry at index i in the second array contains a reference to the dynamic method corresponding to method i, if one has been found in the dynamic class. Thus, there are three possible states for a given index: (1) the boolean is true, so the super method should be immediately invoked, (2) the boolean is false and there is a reference to a dynamic method to be dispatched with the same parameter values, or (3) the boolean is false but the reference is null, signaling that the cache has not yet been updated for this method and that the dynamic class must be queried to determine if an overriding dynamic method exists. So, the first time each method is called, the cache entry is filled. Thereafter, method calls are dispatched immediately from the cache.

Over time, as dynamic methods are declared or deleted, it is important that the values in the caches do not become stale. Therefore, the compiled peer class also provides static methods to reset the caches, either globally or by method index, whenever the method signature of a dynamic class changes.

4 Instantiation, Initialization, and Execution

As outlined above, creating an instance of a dynamic class results in the creation of two mutually referential objects, the dynamic instance and the compiled peer instance. The compiled peer instance is created first, as the representative of the dynamic instance as far as compiled code is concerned. If the dynamic class extends another class, the super constructor executes. Dynamic fields are initialized and the constructor of the dynamic class is executed. From that point on, all changes to the dynamic class are reflected live in the dynamic instance, as described in the following sections.

4.1 Initialization of Dynamic Fields

Full dynamic modification of classes in the face of existing instances requires a field initialization strategy that permits new fields to be added to existing instances. Rather than create replacement objects and map old instances to new ones [8], our solution was to provide each instance of a dynamic class with a dynamic binding table that maps instance variables to values. When new dynamic fields are declared, these are not present in the binding tables of existing instances. Therefore, they are initialized on demand (when they are first referenced during execution) on the basis of the initializer expression defined as part of the field declaraion. This results in a slight departure from the typical initialization on instantiation strategy in order to provide the illusion that the variable was "always there," even if the variable was added later. Obviously, such fields cannot be initialized by the constructor for existing instances, since the constructor executed before the field was declared. Similarly, other changes to constructors affect only new instances of the class.

On-demand initialization gives the programmer the freedom to declare and specify initial values for new fields, knowing that they will not be initialized until another part of the program is modified to reference them and is then executed. Alternatively, one could imagine having the programmer specify a "commit point" (see Section 4.5 below) when the new variable should be initialized in all instances. However, on-demand initialization supports a more fluid development, and saves the programmer from having to remember to commit changes.

4.2 Executing Dynamic Methods

Execution of a dynamic method takes place through traversal of the data structure representing the method. Parameter values are bound to formal parameters and each statement is executed in a semi-interpreted fashion, as described in Section 2.2. Each statement may involve calls into compiled code and/or dynamic code. Editing operations, such as the addition and removal of statements within a method, take effect immediately on the executing program. If the parameter list for a dynamic method is reordered, all calls are updated accordingly and execution proceeds normally because the actual parameter list is represented internally as a mapping from formal to actual parameters, so the change in the order is inconsequential to the execution. Similarly, if a parameter is added or deleted, all calls are updated accordingly. If a parameter is added, all call sites initially contain an empty expression in that parameter slot that must be filled in by the programmer before execution of the call can take place (see Section 5.2). Similarly, any empty expression or type mismatch causes the program to block at that point until the offending expression is corrected. This provides a natural way to block execution of methods while they are being edited. Once the changes are complete, execution can be resumed. However, for many kinds of small changes that one encounters routinely in software development, expressions can be modified without interruption of the execution. Section 4.5 contains further discussion of concurrent program modification and execution.

4.3 Executing Dynamic Constructors

After the dynamic and compiled peer instances have been created, dynamic constructors execute in the same fashion as dynamic methods. To ensure proper initialization, the first line of a dynamic constructor must refer to a parent constructor, or to another constructor within the same dynamic class. The "return value" of a dynamic constructor is implicitly understood to be the newly created object.

4.4 Overhead of Semi-interpreted Execution

We have described a style of execution that we call semi-interpreted. Although there is no parsing of text involved, we use the name semi-interpreted because the structure of each dynamic method is traversed by an execution environment that makes decisions within compiled objects corresponding to predefined control constructs of the language. Each decision, although accomplished in this semi-interpreted way, may (and often does) result in a call to a compiled method using reflection. In practice, we expect that the vast majority of a program's executed statements will occur within compiled code, and therefore that only a small fraction of the execution time will be spent traversing and executing the data structures that represent method bodies. This depends, of course, on how much of the application's computation is specified within dynamic classes, and how much is delegated to method calls in compiled classes. Method caching, as described in Section 3.4, minimizes the overhead for method dispatching.

4.5 Concurrency Control

This paper has presented mechanisms supporting the creation and modification of dynamic classes. The resulting possibility of live software development with dynamic classes raises a number of questions about interleaving program modification with program execution. Although a full treatment of concurrency control issues related to live software development is beyond the scope of this paper, we briefly discuss a variety of concurrency control policies that could be enforced by an integrated development environment supporting dynamic classes.

The choice among concurrency control policies depends on many factors. Correctness for a dynamically evolving program is a slippery issue, since the execution of the program cannot be described in terms of a static implementation, or even a static specification. As such, when choosing a concurrency control policy for dynamic software development, one must define a notion of program consistency such that developers are not surprised by the executions that result from their ongoing modifications of the program. Variable initialization on demand, as discussed in Section 4.1, is one example of providing such consistency.

An important part of any concurrency control policy is defining what constitutes an atomic change. In the case of dynamic program modification, a trade-off exists between program consistency and flexibility of program modifications that directly relates to the grain size of the atomic changes to the program. Larger grain size means increased consistency but less flexibility in making small modifications to a running program. While facilitating modification and exploration is necessary in a fluid development environment, it is important that sufficient consistency be maintained so that programmers are aware of the consequences of their actions. If the chosen concurrency control mechanism permits execution anomalies that result from dynamic modification, it is important that the programmer be aware of the source of these anomalies so that time is not wasted tracking down supposed "bugs" that are really artifacts of live program modification.

The most conservative approach to program update atomicity is the edit-compile-execute cycle typical of compiled languages. This view says any program modification is potentially dangerous and therefore that execution and program modification may not be interleaved. At the opposite end of the spectrum is an aggressive approach that allows program modifications to take place at any time, with the execution unfolding in accordance with the most recent changes. This second option provides the most fluid development, but it comes at the price of possible inconsistencies during execution due to a dynamically changing program. If the development environment does not support recognizing inconsistencies, developers using an optimistic update policy may need to recognize and handle inconsistencies themselves.

A variety of approaches between these two extremes are possible. One such approach would involve explicit commmit points during program modification. For example, when committing changes to a particular method, one might guarantee that no execution of the method is currently in progress. However, such guarantees would be difficult to ensure in a running system without blocking some method calls. A similar, but more pragmatic approach, would be to provide shadow copies of modified methods using a copy-on-write scheme. Currently executing methods would continue to use the old version, while new method calls would use the new version. For recursive methods, one might continue to use the shadow copy until the entire recursion is complete, or perhaps switch implementations mid-stream, depending upon the policy or the developer's preference. However, granularity at the method level may be too course in some cases and too fine in others. If the commit point is left under the control of the developer, then an appropriate granularity could be chosen for each particular modification. For example, in a long-running method containing a loop, a developer might want to see the effect of a fine-grain change while the loop continues to execute. Alternatively, a developer might want to commit modifications of two methods as one atomic change, particularly if the methods call each other.

In purely event-driven programs (particularly those driven entirely by user actions), atomicity of program modification is relatively inconsequential, provided that the developer reach a consistent program definition before testing the next user action. In other words, since execution is under the control of the developer, there is no risk of execution taking place in the middle of a sensitive change to the program. However, in multi-threaded applications with ongoing computations, dynamic program modification becomes more interesting because the program execution and modification are tightly interleaved. In such programs, the more fine-grain is the update atomicity, the greater is the opportunity for immediate feedback during program development.

Section 5 describes a programming environment we have built to explore dynamic classes in the context of computer science education. Since we wanted to push the envelope of dynamic program modification, we chose to implement the most aggressive atomic update strategy that seemed reasonable. In this system, we allow modification of programs at the expression level to be interleaved with program execution. Any expression (or sub-expression) can be modified at run-time, and the new version will be used the next time that expression is evaluated. As we will discuss, each fine-grain atomic editing action (variable declaration, signature change, added statement, etc.) results in a program modification that is immediately available for execution. If execution of an illegal statement or expression is attempted (based on static type-checking), control is given over to the debugger, as described in Section 5.2. As future work, we plan to investigate concurrency control approaches for live software development in situations where the systems are large (even distributed) and involve multiple developers.

5 Implications for Development Environments

We have presented dynamic classes as a mechanism to support live software development. Dynamic classes are designed to support a fluid model of software development in which programs are molded over time, as they continue to execute. Full realization of the dynamic class concept requires an integrated development environment that can exploit their capabilities. In particular, a supporting programming environment must be aware of the program structure so that editing actions result in the appropriate modifications of the dynamic classes. The environment must also provide a tightly integrated debugger that allows for program modification during execution. The environment also must be able to handle exceptions on the fly, bringing up the debugger at the point when exceptions occur, and giving the programmer control over exception propagation and handling. Finally, the

environment must provide a mechanism for deploying an application following development with dynamic classes.

JPie (Java programmer's interactive environment) is a programming environment we have built that supports live software construction through direct manipulation of graphical representations of programs. Because it supports live development, we have JPie has been useful as an educational programming. To illustrate the implications of dynamic classes for integrated development environments, this section briefly discusses the most important aspects of JPie that are enabled by and/or required by dynamic classes. Other aspects of JPie, such as its user interface and its value as an educational tool, are discussed elsewhere. [10, 11, 12]

5.1 Awareness of Program Structure

To take full advantage of dynamic classes, the editor in the programming environment must be aware of the structure of the program. As the programmer edits the program, corresponding changes are made to the dynamic class by calling methods that mutate the class. In JPie, this is accomplished with a separation of model and view in the user interface. Each dynamic class and its memebers are rendered in a graphical representation that the programmer directly manipulates. Each user action results in a corresponding change to the dynamic class, which in turn, is reflected in the rendering of all views of the class.

Direct manipulation supplants textual identifiers as the link between declaration and use. For example, to create a method call in JPie, the programmer either selects the method or drags it into the expression in which the call is to occur. In this way, the method declaration and the call site are structurally linked, so future changes to the method (its name, parameter list, etc.) are immediately reflected at all call sites. Similarly, direct manipulation avoids the possiblity of variable name masking. Even if two variables have the same name, the explicit choice among them (by selection or drag and drop) makes a structural link between declaration and use.

Because programs are running as they are being modified, the programming environment must make allowances for erroneous statements and expressions. In JPie, syntax errors are prevented by the allowable manipulations of the program. However, while a programmer is on the way to completing a correct statement, two kinds of errors are vitually unavoidable. These are empty expressions (such as might occur in a method call for which not all the actual parameters slots have been filled in) and type mismatch errors (such as might result when a certain type is expected and the programmer is in the process of building an expression of that type). Similarly, a variable access or method call is erroneous if the corresponding variable or method declaration has been deleted. The JPie editor flags these errors to call attention to them, but allows the program to execute until such time as execution of an offending statement or expression is attempted. At that point, control is handed over to the debugger, as discussed in Section 5.2.

5.2 Support for Tightly Integrated Debugging

Because dynamic classes can be changed during execution, a supporting programming environment must be prepared to deal with modifications to programs that result in the attempted execution of erroneous expressions. JPie provides a thread-oriented debugger that uses the same visual representation that is used in the class windows. Programmers can set breakpoints on methods, constructors, behaviors, event handlers, statements, and expressions. Erroneous expressions are treated similarly. When a breakpoint or an erroneous expression is reached within execution of a thread, a debugger window pops up, showing the call stack as a series of tabbed panes. Each pane shows the expanded visual representation of the method (or other item) responsible for that

stack frame. The debugger highlights (within the each stack frame) the expression that is currently executing (or about to execute in the case of the top stack frame). In the debugger, the programmer can control the execution speed of that thread and watch the execution unfold, or can single-step through the execution expression by expression, with pop-up text displaying values for executed expressions. The programmer may also change the program, perhaps to correct errors, and resume execution.

The debugger also provides proactive support for detecting common logic errors before they become fatal errors. This includes dynamically adjustable stack bounding to detect infinite recursion, dynamically adjustable loop bounding to detect infinite loops, and deadlock detection. In the case of deadlock, a separate window appears with a visualization of the cycle in the wait-for graph. Within that visualization, the programmer can click on threads involved in the cycle in order to bring up debugging windows for them, and optionally terminate them to break the deadlock.

5.3 On-the-fly Exception Handling

Our goal of live program modification extends not only to normal execution, but exception handling as well. In other words, we want live software development to include the ability to add or modify exception handling code in the course of program execution. The JPie debugger supports on-the-fly exception handling as follows. When an exception occurs that is not explicitly caught or thrown by a method, the debugger appears and provides the programmer with the opportunity to catch, throw, or propagate the exception and resume execution.

On-the-fly exception handling brings to the forefront questions about where control should resume following program modification. For example, suppose a thread is paused within an expression (due to an exception, an erroneous expression, or a breakpoint), and suppose that the programmer replaces the entire statement containing that expression. Where should execution resume? One could abort the threads and require the programmer to start over, but that would severely limit live incremental software development.

Certainly, questions like these would normally not come up once the program under development is completed and deployed. Even in the case of live software upgrades, the development is considered finished and the program fully tested. However, in order to achieve the goal of live software development, consistent handling of modifications during execution is critical to providing reasonable execution semantics to the programmer during the development process. Because this is a new area of research, we first had to ask ourselves what programmers would expect the environment to do in the face of a variety of dynamic changes, and then devise a consistent semantics based on those expectations. As such, the answer to this control resumption question has more to do with programmer expectation than with correctness, for (as mentioned in Section 4.5) correctness takes on a different meaning when the program is changing dynamically.

JPie answers this question by preempting execution of the deleted expression and resuming execution with its replacement. In the case of exceptions, JPie also allows the programmer to use the debugger's single-step function to propagate the exception as far out as desired (through both scope and stack frames) and then retry execution with the new statement in place. Because programs contain a mixture of compiled and semi-interpreted code, side effects from the partial computation are visible in JPie. However, one could also imagine an environment providing rollback capabilities to restore the state of the program before re-trying execution due to an exception.

5.4 Deployment of Dynamic Classes

One can imagine two options for the deployment of dynamic classes following development, either as compiled classes, or within an interpreter. One option JPie currently provides is to automatically generate Java source code for dynamic classes and compile them. (For both verification and educational purposes, we have taken care to generate source code that is human-readable.) Deploying dynamic classes this way results in no overhead at run-time, but also does not support further modification during execution. A second option would be to run the dynamic class within JPie's execution environment. This would incur the same overhead as during development, but would permit modification in the field. We are currently exploring this second option in the context of implementing a medium-sized mail service application.

6 Conclusion

We have presented fine-grain dynamic classes as a means to support live program modification during software development. We have described an architecture for and implementation of dynamic classes in Java that provides full inter-operability between dynamic and compiled classes without modification of the Java virtual machine.

The implementation described in this paper assumes that the ancestors of a class do not change over time (i.e., subclass relationships are not changed). We are currently working to remove this restriction by allowing the compiled peer of the dynamic class to change. This has interesting implications for existing instances of the dynamic class that were created with the old compiled peer.

We believe that dynamic classes offer significant potential for improving the way professional programmers construct software, as well as to make the power of object-oriented software development more accessible to "casual programmers" who have the occasional need to write software for use in another discipline, as well as to students who are just beginning their study of computer science. Furthermore, dynamic classes open up a range of possibilities in particular application domains. For example, we have been working on mechanisms to support live construction of client/server applications using standard communication technologies, such as SOAP [24] and CORBA [18], but in which the server's interface (and the client's use of that interface) may change dynamically while the software is running and the client and server are connected.

Acknowledgements

I thank the past and present members of the JPie development team: James Aguilar, Ben Brinckerhoff, Ben Birnbaum, Joel Brandt, Vanessa Clark, Melanie Cowan, Matt Hampton, Dylan Lingelbach, Oren Melzer, Adam Mitz, Brandon Morgan, Jonathan Nye, Sajeeva Pallemulle, and Richard Souvenir. I thank Keith Sawyer and the students in CS123 for their feedback on using dynamic classes in JPie. I also thank James Aguilar, Adam Mitz, and Sajeeva Pallemulle for their comments on earlier versions of this paper. This work was supported in part by the National Science Foundation under CISE Educational Innovation Grant 0305954.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[2] T. Bloom. Dynamic module replacement in a distributed programming system. Ph.D. dissertation, 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] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *OOPSLA*, pages 49–70, October 1989.

[6] Sean Dorward, Ravi Sethi, and Jonathan E. Shopiro. Adding new code to a running C++ program. In *Proceedings of the Usenix C++ Conference*, pages 279–292, April 1990.

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

[8] 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.

[9] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation.* Addison-Wesley, 1983.

[10] 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.

[11] Kenneth J. Goldman. A concepts-first introduction to computer science. In *ACM SIGCSE Technical Symposium on Computer Science Education*, pages 432–436, March 2004.

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

[13] Graham Hamilton, editor. *The JavaBeans 1.01 Specification.* Sun Microsystems, August 1997.

[14] Stphane Hillion. DynamicJava. http://koala.ilog.fr/djava/, 1999.

[15] Gsli Hjlmtsson 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.

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

[17] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *ECOOP*, pages 337–361, June 2000.

[18] Common object request broker architecture: Core specification, 3.0.3. Technical report, Object Management Group, March 2004. http://www.omg.org/docs/formal/04-03-01.pdf.

[19] Raju Pandey, Scott Malabarba, Tim Stapko, and Brant Hashii. Dynamically evolvable distributed systems. In *COMPSAC*, pages 1022–1027, August 2002.

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

[21] C. Simonyi. Intentional programming: Innovation in the legacy age. Presented at IFIP Working group 2.1 workshop. http://www.research.microsoft.com/research/ip/, 1996.

[22] BDK 1.1: tool compliance in JavaBeans 1.0. Technical report, Sun Microsystems, 1996. http://java.sun.com/products/javabeans/software/beanbox.html.

[23] Bala Swaminathan and Kenneth J. Goldman. Dynamic reconfiguration with I/O abstraction. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing*, pages 496–501, October 1995.

[24] Simple object access protocol (SOAP) 1.2. Technical report, World Wide Web Consortium, June 2003. http://www.w3.org/TR/SOAP/.