Implementing Dynamic Language Features in Java using Dynamic Code Generation

Thomas M. Breuel
Xerox PARC, Palo Alto, CA
tmb@parc.xerox.com

Abstract

Compared to dynamic object-oriented languages like CommonLisp or Smalltalk, Java has a fairly simple and restrictive object and type system. Some of the key differences between Java and these other languages is that they offer structural conformance, automatic delegation, and dynamic mixins. When such constructs are needed in a language like Java, they are usually expressed in terms of standard object-oriented design patterns, requiring the manual implementation of "glue" or "helper" classes. This paper describes ways in which such features can be provided efficiently and automatically in terms of Java's platform-independent binary format and dynamic loading mechanisms. The implementation does not require any changes to the Java runtime, bytecodes, or class loader and yields performance comparable to manually implemented design patterns. The approach should prove useful both as a programming tool for Java and as an strategy for building efficient implementations of dynamic languages on top of the Java virtual machine.

1: Introduction

Java is a programming language originally based on a simplification of the C++ programming language. The language has a fairly simple static type system, with some support for safe runtime type casting and reflection. Java is considerably less expressive than dynamic programming languages like Smalltalk or CommonLisp. Some of the constructs available in others languages are:

- Objects can be used interchangeably as long as they implement compatible methods. This is referred to as "structural conformance" (e.g. Smalltalk-80, [5]).
- Methods on instances can be "wrapped" by other methods; that is, the programmer can define methods that are executed before and/or after a method of the target instance. (e.g. CLOS, [1]).
- Method calls on one object can be delegated to another object (e.g. Objective-C, [11]).
- Instances can be created of classes that "mix" multiple parent classes dynamically (e.g., CLOS, [1]).

These and similar constructs do not exist in standard Java. There have been suggestions for implementing some of these facilities by using preprocessing or compilation techniques

or modifications to the Java runtime. For example, Läufer *et al.* [10] propose a method for implementing safe structual conformance using a precompiler and optional modifications to the Java runtime. Viega *et al.* [14] propose a mechanism for automated delegation in class-based languages, primarily as an alternative to multiple inheritance. Aspect Oriented Programming (AOP) (Kiczales *et al.* [9]) is a related approach, providing facilities closely related to wrappers and delegation. Keller and Hölzle [8] have proposed a mechanism for binary component adaptation (BCA) based on load-time modification of Java byte codes. They also provide an extensive bibliography on the subject of component adaptation and the creation of adapter classes.

In addition to these approaches, there are several languages built on to of the Java VM, some of which support structural conformance, multiple inheritance, and/or delegation [3]. Of particular note is Jython [7, 6], which uses a combination of reflection, dynamic compilation, and class loaders to achieve structural conformance and multiple inheritance.

This paper describes implementation strategies for structural conformance, wrapping, delegation, and mixins that do not rely on compile-time type information, do not impose the overhead of reflection on method calls, and do not require modifications to the Java runtime. The techniques described in this paper can also be used for implementing a number of other interesting language features, including method pointers, bound methods (Microsoft's "delegates"), fast marshalling/unmarshalling, member variable indexing, automatically changing overloading into dynamic dispatch over multiple arguments,

The prototype implementation be used as a library with a standard Java compiler and no changes to the Java language. It also does not require changes to the default class loader, or changes to the Java virtual machine. It also appears that it can also be made safe for use with sandboxed Java code (e.g., applets).

2: Examples

Before describing the implementation techniques, let us look at some examples of how these features can be used in Java.

Structural Conformance Structural conformance is perhaps the simplest of the functions that can be implemented using the methods described in this paper. Here, we give only a superficial discussion of the subject; for a more in-depth analysis, as well as a compile-time extension to the Java language permitting structural conformance, see the paper by Läufer *et al.* [10].

In object-oriented languages like Smalltalk or CommonLisp, function arguments and variables carry no type declarations. All that matters when using an object is that it implements methods by the names that the callers expect and that those methods have the right semantics.

In Java, of course, this is not possible, since Java uses conformance by name: types are compatible based on named inheritance, not their structure. Callers need to ensure that objects they pass already implement the required interfaces required by the callee. The problems with this is that new interfaces cannot be added to existing classes after those classes have been defined, even if the existing classes actually implement the new interface. A commonly used workaround is the use of the "Adapter" or "Wrapper" classes (see [4] for a description of the corresponding pattern in the language of design patterns).

For example, the interface `java.io.Reader` requires nine methods to be defined, but many users of that interface probably get by with a much simpler interface, requiring only the presence of the `read()` method. If, in implementing a new method, we require conformance to the `java.io.Reader` interface, we force users of that method that want to pass some other kind of object to implement eight methods that do not need to be implemented. On the other hand, if we define a new interface, say, `SimpleReader`, then no existing class that implements `java.io.Reader` will conform to that new interface. Furthermore, we cannot retroactively make `SimpleReader` a parent of `java.io.Reader`. So, if we want to use an instance of such a class, we have to write an explicit wrapper class:

Example 1

```
interface SimpleReader {
    int read();
}

class UsesSimpleReader {
    void callee(SimpleReader r) {...}
    void caller(Reader r) {
        SimpleReader scr = new SimpleReader() { public int read() { return r.read(); } }
        callee(scr);
    }
}
```

Of course, this is a particularly simple example: creating the anonymous glue class inside method `caller` is only a few lines. However, for more complex interfaces, every method needs to get forwarded manually (a simple form of delegation).

Furthermore, any changes to the interface will also require every location where a glue class is created explicitly to be maintained and updated. Sometimes, this enforced code review may be desirable, but usually, it is unnecessary work and only risks the accidental introduction of errors. Structural conformance gives us the choice to reduce coupling between the interface and its uses.

Delegation Many Java APIs are defined in terms of Java interfaces (rather than classes). Instances of classes conforming to those interfaces are generated using methods on "factory" objects [4]. The `java.sql` (JDBC) package, for example, contains interfaces defining an SQL `Statement` and its extension, `PreparedStatement`. Instances conforming to those interfaces are generated by instances of factory classes (conforming to the `Connection` interface).

Sometimes it is desirable to override some of the methods in a `Statement` or `PreparedStatement`. For example, we might want to track resources associated with a statement or perform performance measurements. If we obtain an object implementing one of those interfaces from a factory method, we cannot necessarily access or extend the actual class that that object is an instance of. Even if we could, we may not be able to invoke its constructor. This means that we are unable to take advantage of the primary mechanism for reuse in object-oriented languages: inheritance.

The only approach that we have for overriding some of the methods is that of delegation. In practice, this may look like:

Example 2

```
final Statement stmt = connection.getStatement();
Statement stmt_with_delegation = new Statement() {
```

```
    // statements we want to instrument

    public boolean execute(String s) {
    start_timer();
        boolean result = stmt.execute(s);
        stop_timer();
        return result;
    }

    public ResultSet executeQuery(String s) {
        start_timer();
        ResultSet result = stmt.executeStatement(s);
        stop_timer();
        return result;
    }

    public int executeUpdate(String s) {
        start_timer();
        int result = stmt.executeUpdate(s);
        stop_timer();
        return result;
    }

    // now we need to forward all the other calls

    public void addBatch(String s) { stmt.addBatch(s); }
    public void cancel() { stmt.cancel(); }
    public void clearBatch() { stmt.clearBatch(); }
    // ... 22 more methods ...
};
```

As this example shows, manual delegation in Java is rather laborious. If we had wanted to do the same for a `PreparedStatement`, we would have had to forward more than 30 additional methods. Furthermore, the class performing delegation becomes tightly dependent on the definition of the interface it is delegating to, because it needs to explicitly forward every method in the interface. That means that if additional methods are added to the interface (as happened in the case of JDBC between versions 1 and 2), the delegation class will fail to compile and need to be updated.

In CommonLisp, Smalltalk and some other dynamic object oriented languages, this kind of delegation could be easily accomplished via a `doesNotUnderstand` method which forwards all the unknown methods to the target object (see [9] for another approach). The dynamic code generation techniques described in this paper alow this form of delegation to be implemented much closer to the way it would be implemented in CommonLisp or Smalltalk. In particular, the programmer only needs to concern himself with the forwarding of the classes he actually wants to override or instrument; all the other forwarding is handled automatically.

Mixins As a third example, consider writing a simulation system in which each agent is composed of a number of "mixins". An important capability of these mixins is to be able to refer to methods contributed to the complete agent by other mixins. For example, we might define:

Example 3

```
interface Animal {
```

```
    void predatorDetected(Direction d);
    void runIntoDirection(Direction d);
}
class VisualBehavior {
    void predatorDetected(Direction d) { runIntoDirection(d.reverse()); }
};
class MotorBehavior {
    void runIntoDirection(Direction d) { ... }
};
class TimidAnimal
    extends VisualBehavior,MotorBehavior
    implements Animal {}
```

Of course, this code is not legal Java because `VisualBehavior` does not actually define a `runIntoDirection` method. We could express this code in Java as:

Example 4

```
interface Animal {
    void predatorDetected(Direction d);
    void runIntoDirection(Direction d);
}
class VisualBehavior {
    MotorBehavior motorBehavior;
    void predatorDetected(Direction d) { motorBehavior.runIntoDirection(d.reverse()); }
};
class MotorBehavior {
    void runIntoDirection(Direction d) { ... }
};
class TimidAnimal implements Animal {
    VisualBehavior visualBehavior;
    MotorBehavior motorBehavior;
    TimidAnimal() {
        visualBehavior = new VisualBehavior();
        motorBehavior = new MotorBehavior();
        visualBehavior.motorBehavior = motorBehavior;
    }
    void predatorDetected(Direction d) { visualBehavior.predatorDetected(d); }
    void runIntoDirection(Direction d) { motorBehavior.runIntoDirection(d); }
};
```

So, in order to achieve mixin semantics, we need to again manually write forwarding methods for each method required by the interface, and we need to introduce additional instance variables that make it possible for the different mixins to refer to each other. This is gets quite complex for any significant use of mixins.

The dynamic code generation techniques described in this paper let us write the mixin code almost as easily as shown in the first (hypothetical) code example. All the bookkeeping, creation of auxilliary instance variables, and method forwarding are handled automatically.

3: Approach

The basic approach we to adding dynamic features as described in the examples above is to use Java's reflection APIs to get information about the different classes (e.g., which methods need to be forwarded) and to use on-the-fly generation of byte-codes to create the "glue classes" that implement the forwarding methods and auxilliary variables (let us

use the term "glue classes", since "adapter" or "wrapper" more specifically refers to the technique of interface adaptation [4]).

The glue classes are generated in their own class loader, but this class loader is used completely internally to the library; this means that the approach works even for systems that themselves manipulate class loaders. Furthermore, the glue classes need to be generated only once for each combination of types; once generated, references to their class objects are maintained in a hash table and can be looked up and instantiated quickly.

Some implementation techniques for advanced language constructs (e.g., genericity or object persistence) use a modified class loader that rewrites method byte codes as objects are loaded. None of the implementations described here require actually examining or rewriting the byte codes of methods on the objects being manipulated.

By choosing semantics of the different operations carefully, we can obtain usable APIs for features like delegation and mixins without any syntactic modifications to Java (and using a standard Java runtime).

4: Structual Conformance

Using the `Dynamic` library, we can automate the process of adapting a class to an interface to implement structural conformance:

Example 5

```
class UsesSimpleReader {
    void callee(SimpleReader r) { ... }
    void caller(Reader r) { callee((SimpleReader)Dynamic.adapt(SimpleReader.class,scr)); }
}
```

We can also move the adaptation step inside the `callee`. This is more convenient for callers but offers somewhat less opportunity for static type checking:

Example 6

```
class UsesSimpleReader {
    void callee(Object obj) {
        SimpleReader r = (SimpleReader)Dynamic.adapt(SimpleReader.class,obj);
        ...
    }
    void caller(Reader r) { callee(r); }
}
```

In a simple example like this, the savings in terms of programmer effort are not very large, but automating the process of adapting interfaces may encourage programmers to define interfaces that actually express the requirements of the code they are writing, rather than picking convenient existing interfaces even if those require more functionality to be implemented than necessary.

Here is how we implement `Dynamic.adapt(Class I,Object O)`:

1. Check whether we have already generated an instance of a glue class for this combination of arguments and hold it in the cache; if so, return it.
2. Check whether we have already generated runtime code for the particular combination of types; if so, skip forward to Step 9
3. Start generating a glue class `G` that implements the given return interface `I`.
4. Add an instance variable `f` of the same class as object `O`.

5. Use `java.lang.reflect` to obtain a list of methods required by the return interface `I`.
6. For each method `m` required by `I`, check whether `m` is implemented by `O`. If it is, generate a method on `G` with the same signature as `m` that forwards the method to the object held by `f`. If object `O` does not implement `m`, there are two choices depending on the style of structual conformance we want to implement: we can either throw a runtime error, or we can generate code that implements `m` by throwing a runtime error.
7. Load the class `G` into the internal class loader.
8. Store a reference to the resulting class in a hash table that is indexed by the interface `I` and the type of `O`. We can use this on subsequent calls to the `Dynamic.adapt` method to avoid re-generating class `G` from scratch.
9. Instantiate `G` and fill in its instance variable `f` with a reference to objects `O`. Return the instance.

Note that because the automatically generated instance of the glue class is stateless, we can cache the instance itself, in addition to caching the code for the glue class. Strategies for when and how glue classes and instances of glue classes are cached is something that requires more extensive benchmarking. The author expects that a strategy similar to backpatching at specific call sites, used in high performance Smalltalk-80 implementations, would work, and such a strategy can still be expressed conveniently in terms of existing Java language constructs.

5: Delegation/Forwarding

Let us look at the delegation example above using the `Dynamic` library:

Example 7

```
Statement stmt = connection.getStatement();
Statement stmt_with_delegation =
    (Statement)Dynamic.wrap(Statement.class,stmt,new Object() {
    public boolean execute(Statement stmt,String s) {
        start_timer();
        boolean result = stmt.execute(s);
        stop_timer();
        return result;
    }

    public ResultSet executeQuery(Statement stmt,String s) {
        start_timer();
        ResultSet result = stmt.executeStatement(s);
        stop_timer();
        return result;
    }

    public int executeUpdate(Statement stmt,String s) {
        start_timer();
        int result = stmt.executeUpdate(s);
        stop_timer();
        return result;
    }
};
```

Here, the call to `Dynamic.wrap` takes three arguments: a reference to the interface the resulting object needs to implement (`Statement.class`), an instance to which methods are to be fowarded (`stmt`), and an instance of `Object` that holds methods that override functionality of the target object.

Note that the overriding methods defined on this object take an additional argument

compared to Example 2. This corresponds to the `forwarder` keyword in [14] and allows the method to which the call is delegated to refer to the delegator

In order to be able to type the static method `Dynamic.wrap`, it needs to return an instance of type `Object` that we then cast explicity to the type we require (`Statement`).

Let us look now at how `Dynamic.wrap(Class I,Object E,Object R)` is actually implemented. Here are the steps it performs:

1. Immediately return a cached instance of the glue class if it exists for the type/object combination.
2. Immediately skip to the instantiation step (Step 9) if we have already generated code for the particular type combinations on a previous call.
3. Start generating a glue class `G` that implements the given return interface `I`.
4. Add two instance variables to the glue class. The first instance variable holds the wrapping object `R`, the second instance variable holds the object being wrapped `E`.
5. Use `java.lang.reflect` to obtain a list of methods required by the return interface `I`.
6. For each method `m` required by `I`, first check whether there exists a method on the wrapping object R that has the same name and a type signature that is derived from `m` by inserting an additional argument at the beginning that has the type of the object being wrapped, `E`. If so, generate a forwarding method with the same signature as method `m` that forwards to the corresponding method on `R` with `E` as its first argument. If not, generate a forwarding method that forwards to the corresponding method on the object being wrapped, `E`.
7. Load the class `G` into the internal class loader.
8. Store a reference to the resulting class in a hash table that is indexed by the interface `I` and the types of `R` and `E`. We can use this on subsequent calls to the `Dynamic.wrap` method to avoid re-generating class `G` from scratch.
9. Instantiate `G` and fill in its instance variables with references to objects `R` and `E`. Return this instance.

Viega *et al.* [14] describe a similar delegation mechanism for Java. Their system is based on a preprocessor and requires compile-time declarations of delegation relationships.

6: Mixins

For implementing mixins, our goal is to implement a function `Object Dynamic.mix(Class I,Class A,Class B)` that takes an interface `I` and two classes `A` and `B` as arguments and returns an instance of a class constructed by mixing the methods of `A` and `B`, making the resulting class conform to interface `I`, creating an instance of that class and returning it (the extension to mixing more than two classes is straightforward).

In Example 4, we have already seen the general approach to generating classes that inherit from multiple parent classes. However, we encounter the following problem when trying to define mixins that call methods on each other: since Java has no means of expressing dependency on another mixin, the Java compiler will not compile a mixin class that calls a method defined by another mixin.

There is a fairly simple solution to expressing these mixin relationships using existing syntactic constructs in Java: each mixin defines the methods it expects to be contributed to the complete object by other mixins and that it wants to call as abstract methods. With this modification, our mixin example above (Example 3) looks like this:

Example 8

```
interface Animal {
    void predatorDetected(Direction d);
    void runIntoDirection(Direction d);
}
class VisualBehavior {
    abstract void runIntoDirection(Direction d);
```

```
    void predatorDetected(Direction d) { runIntoDirection(d.reverse()); }
};
class MotorBehavior {
    void runIntoDirection(Direction d) { ... }
};

TimidAnimal timidAnimal =
        Dynamic.mix(Animal.class, VisualBehavior.class, MotorBehavior.class);
```

Here is how `Dynamic.mix(Class I,Class A,Class B)` implements the creation of mixins:

1. Immediately return a cached instance of the glue class if it exists for the type/object combination.
2. Immediately skip to the instantiation step (Step 9) if we have already generated code for the particular type combinations on a previous call.
3. We generate a container class `C` that implements all the methods required by `I`.
4. We generate two classes, `AX` and `BX` that extend classes `A` and `B`, respectively. These new subclasses add an instance variable `c` of type `C` each.
5. We add two instance variables `ax` and `bx` to `C` of type `AX` and `BX`, respectively.
6. For each method required by interface `I`, we forward it either to the object held by `ax` or by `bx`. In case of ambiguities, the simplest and least surprising thing is to raise an error, but some form of method combination would be possible as well.
7. For each abstract method in mixin `A` that refers to a method defined by mixin `B`, we generate a non-abstract method on class `AX` that forwards the method call through `c.ax`. We do the analogous thing for mixin `B`.
8. In the constructor of class `C`, we create instances of classes `AX` and `BX` and store them in the instance variables `ax` and `bx`, respectively. We also initialize the instance variables `c` to the container class instance.
9. Instantiate `C` and initialize all the instance variables of the generated classes as needed. Return this instance.

`Dynamic.mix` gives us something very similar to multiple inheritance based on delegation (for a more in-depth discussion of the relationship between multiple inheritance and delegation, see [12, 13, 14]).

A significant extension compared to multiple inheritance in languages like C++ or Eiffel, and compared to the method described in [14], is that `Dynamic.mix` allows the parent classes of a derived class to be determined at runtime. Among widely used languages, this feature is mostly known in CLOS, Perl, and Python, and there it is generally made possible because method lookup uses general purpose but slow data structures. Because `Dynamic.mix` compiles glue classes on the fly, its performance is as good as if the required delegation code had been statically compiled.

7: Implementation and Benchmarking

The techniques described in this paper were implemented in a prototype implementation initially under the Sun JDK 1.1 distribution and later ported to the IBM Jikes compiler and JDK 1.3. Using the prototype implementation, it was verified that for cache hits in the `Dynamic.adapt` case, calls through automatically generated glue classes execute at approximately the same speed as calls through a manually written glue class, and they execute at about four times the speed of calls through the `java.reflection` API. More detailed benchmarking results will be presented elsewhere.

8: Discussion

While dynamic features would be most naturally implemented by simply extending the Java virtual machine (JVM), for practical reasons, the definition of the JVM is unlikely to change. This paper has shown how dynamic language features can be implemented efficiently without extensions to the JVM using dynamic code generation.

Of course, dynamic code generation and dynamic compilation techniques have been used widely over the last decade in the implementation of object-oriented programming languages (see, for example, [2]), and dynamic and adaptive native code generation in Java runtimes themselves is an example of this.

The main contribution of this paper is to show how dynamic and adaptive code generation can also be used by Java programs themselves at the level of Java bytecodes to implement high-level dynamic constructs efficiently and under program control. In different words, the availability of a well-defined platform-independent binary format as part of the Java standard and the ability to generate and dynamically load code in that format gives us a very powerful tool for extending the behavior of the Java runtime with features ordinarily only found in dynamic programming languages.

The techniques described in this paper may find application in the efficient implementation of dynamic languages like Smalltalk and Lisp on top of the JVM, as well as for standardizable extensions of the Java language and libraries with more dynamic features.

References

[1] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, and D.A. Moon. Common lisp object system specification. i. programmer interface concepts. *LISP and Symbolic Computation*, 1(3-4):245–98, 1989.

[2] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings of OOPSLA'89*, pages 49–70, 1989. Published in SIGPLAN Notices 24(10).

[3] R. Tolksdorf (ed). Languages for the Java VM. `http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html`.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] A. Goldberg and D. Robson. *Smalltalk-80 – The Language and Its Implementation*. Addison-Wesley, 1983.

[6] J. Hugunin. Python and Java: The Best of Both Worlds. In *Proc. 6th Int. Python Conference, San Jose, CA*, 1997.

[7] J. Hugunin, B. Warsaw, et al. The Jython/JPython Web Site. `http://www.jython.org/`.

[8] R. Keller and U. Hölzle. Binary Component Adaptation. In *ECOOP'98 – Object-Oriented Programming 11th European Conference Proceedings*, 1998.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. *ECOOP'97 - Object-Oriented Programming 11th European Conference. Proceedings*, pages 220–42, 1997.

[10] K. Läufer, G. Baumgartner, and V. F. Russo. Safe structural conformance for java. Technical report, Computer and Information Science Department, Ohio State University, 1998. `http://www.cis.ohio-state.edu/~gb/`.

[11] L. J. Pinson and R. S. Wiener. *Objective-C*. Addison-Wesley, 1991.

[12] L.A. Stein. Delegation is inheritance (object-oriented programming). *SIGPLAN Notices*, 22(12):138–46, 1987.

[13] E. Tempero and R. Biddle. Simulating multiple inheritance in java. *Journal of Systems and Software*, 55(1):87–100, 2000.

[14] J. Viega, P. Reynolds, and R. Behrends. Automating delegation in class-based languages. *Proceedings. 34th International Conference on Technology of Object-Oriented Languages and Systems - TOOLS 34*, pages 171–82, 2000.