

Javassist — A Reflection-based Programming Wizard for Java

Shigeru Chiba

*Institute of Information Science and Electronics
University of Tsukuba
and Japan Science and Technology Corporation
email: chiba@is.tsukuba.ac.jp, http: www.is.tsukuba.ac.jp/~chiba*

Abstract

This paper presents the Javassist system, which is a programming tool for assisting Java programmers. It enables programmers to write a meta-level program automating some kinds of class definitions. Moreover, a number of applications of runtime reflection can be implemented with this system.

1 Introduction

A wizard is one of key features of today's application software. It assists users in various means so that their work with that software is productive. For example, Microsoft Word provides the letter wizard, which inquires some simple questions and automates common letter elements such as formatting. Also a number of major software development tools, such as Visual C++ and Visual Café, provide programming wizards, which help programmers edit source code and use class libraries.

This paper presents *Javassist*, which is a new kind of wizard for assisting Java programming. It automates class definitions if they are mechanically derived from other classes.

For example, since Java does not provide a mechanism for parameterized types (in other words, a template mechanism), programmers who want to use a vector holding references only to `String` objects are forced to use type casts whenever an element is fetched from that vector:

```
String s  
= (String)aVectorOfString.elementAt(3);
```

Otherwise, the programmers have to define a new class for a vector of `String`:

```
public class StringVector  
    extends java.util.Vector {  
    public void addElement(String s) {  
        super.addElement(s);  
    }  
  
    public String at(int i) {  
        return (String)elementAt(i);  
    }  
}
```

Javassist automatically defines `StringVector` instead of the programmers if the element type (`String`) is specified by a simple annotation.

Javassist does not only handle predefined assistance but also allows programmers to define a new kind of assistance, which is described in Java itself. Programmers write a *meta-level* program with using the reflection API [3] and the Javassist API, and then Javassist performs assistance according to that meta-level program.

In the rest of this paper, we present an overview of the Javassist system and its API. Then we present examples of the use of Javassist and discuss its implementation issues and related work.

2 Javassist

Javassist is a programming tool for *assisting* Java programmers. In this section, we show how Java programmers can use Javassist to automate some class definitions. We first show a program using Javassist for automatically defining `StringVector` and then how that automation is implemented in Java. We present

an overview of the Javassist API and also present that it is available at either compile time or runtime.

2.1 Use of Javassist

To automatically define a `StringVector` class and use it, programmers write a simple annotation for an `import` declaration:

```
import java.util.Vector
    by VectorAssistant(String);
```

The annotation begins with `by`. It means that `java.util.Vector` is imported with assistance by an instance of a class `VectorAssistant`, which is defined in a meta-level program. `String` is a parameter to that instance.

Then this program must be compiled by the Javassist compiler. Suppose that the program is named `foo.j`. Then the programmer should type:

```
% jc foo.j
```

This command invokes the compiler and produces byte code for the `StringVector` class in `tmp/StringVector.class`. It also translates `foo.j` into `foo.java`. In the `foo.java` file, the annotated `import` declaration above is changed into regular declarations:

```
import java.util.Vector;
import tmp.StringVector;
```

The second declaration makes `StringVector` available in the rest of the source file.

2.2 Meta Programming

The assistance shown above is performed by a `VectorAssistant` object, which is a pure Java object. If the Javassist compiler encounters an annotated `import` declaration, it dynamically loads the specified assistant object and invokes the `assist()` method on that object with the class specified by the `import` declaration and given parameters:

```
public Class[] assist(Class imported,
                    String[] params)
    throws CannotCompileException
{
    Class[] results = {
        imported,
```

```
        makeSubclass(CtClass.forName(params[0]))
    };
    return results;
}
```

This method returns an array of `Class`¹. All the class objects in this array are imported in a compiled program. In the example above, the first element of the array is `Vector` and the second one is `StringVector`.

The `makeSubclass()` method receives the class object representing an element type and returns a vector class for that particular type:

```
public Class makeSubclass(Class type)
    throws CannotCompileException
{
    CtClass vec = new CtClass();
    vec.setName(CtClass.toSimpleName(
        type.getName()
        + "Vector");
    vec.setSuperclass(
        java.util.Vector.class);

    Class[] args1 = { type };
    vec.addMethod(Void.TYPE, "addElement",
        args1, null, 1);
    Class[] args2 = { Integer.TYPE };
    vec.addMethod(type, "at",
        args2, null, 2);

    vec.setRuntimeMetaobject(MVector.class);
    return vec.compile();
}
```

`CtClass` (compile-time class) is a class provided by the Javassist API. The `makeSubclass()` method first creates a `CtClass` object and specifies a class name and a superclass. Then it adds two methods `addElement()` and `at()` to that created class. The parameters of `addMethod()` are a return type, a method name, a list of parameter types, a list of exceptions, and an integer identifier. The constructed class is compiled if a method `compile()` is invoked on that `CtClass` object. `compile()` returns a regular `Class` object.

The behavior of the methods of the constructed class is implemented by runtime metaobjects specified by `setRuntimeMetaobject()` (Figure 1). In the example above, the methods `addMethod()` and `at()` are implemented by a class `MVector` (meta vector)². `MVector` is a subclass of `Metaobject`. The methods

¹`Class` is part of the regular Java language. It is provided by the reflection API and enables introspection of Java classes.

²`class-name.class` is a special notation to get a reference to the `Class` object representing that class.

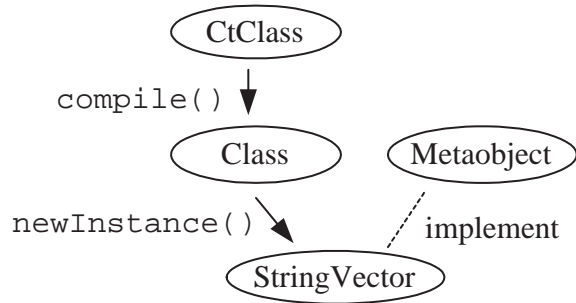


Figure 1: The Meta-level Architecture of Javassist

`addMethod()` and `at()` of `StringVector` only delegate a method call to an instance of `MVector`. Every instance of `StringVector` is associated with a distinct instance of `MVector`.

All the method calls on a `StringVector` object is delegated to `trapMethodcall()` on the metaobject. This method simulates the methods of `StringVector` shown in Section 1:

```

public Object trapMethodcall(
    int identifier, Object[] params)
{
    Vector baseobj = (Vector)getObject();
    if (identifier == 1) {
        baseobj.addElement(params[0]);
        return null;
    }
    else if(identifier == 2) {
        Integer intobj = (Integer)params[0];
        int i = intobj.intValue();
        return baseobj.elementAt(i);
    }
    else
        return super.trapMethodcall(
            identifier, params);
}
  
```

`identifier` is an integer identifier specifying the called method at the base level. `params` is a list of actual parameters. This method calls `addElement()` or `elementAt()` on the base-level object but the called methods are of the super class `Vector`. `getObject()` is part of the Javassist API; it returns the base-level object associated with this metaobject. The classes `Object` and `Integer` belong to the reflection API and are part of the regular Java language.

Although all the method calls on a base-level object are normally forwarded to `trapMethodcall()` on a metaobject, programmers can specify another method

instead of `trapMethodcall()`. Suppose that programmers add another method to `vec` in `makeSubclass()` as follows:

```

vec.addMethod(Boolean.TYPE, "isEmpty",
              null, null, "checkSize");
  
```

This expression adds a method the signature of which is:

```

boolean isEmpty()
  
```

Unlike the other methods shown at the beginning, this method `isEmpty()` is implemented by `checkSize()` of `MVector`. Note that the last parameter to `addMethod()` is not an integer but a character string specifying the implementing method.

If `isEmpty()` is called on a base-level object, it calls `checkSize()` on the metaobject associated with that object. The method `checkSize()` of `MVector` must have the same signature as `isEmpty()`:

```

boolean checkSize() {
    Vector baseobj = (Vector)getObject();
    return baseobj.size() == 0;
}
  
```

The base-level method `isEmpty()` returns the resulting value of `checkSize()`.

A uniqueness of the implementation of `isEmpty()` is that actual parameters to this method is not converted to an array of `Object`. This means that a meta-level method cannot implement a base-level method in a generic way. It cannot implement multiple base-level methods with different signatures. On the other hand, `trapMethodcall()` can implement base-level methods with any kind of signature since parameters including built-in types are converted to a canonical type: array of `Object`.

2.3 Runtime Assistance

Although the Javassist API is designed for compile-time use, programmers can also use it at runtime. They can create a `CtClass` object and compile it at runtime. No special annotations to `import` declarations are needed.

This ability is useful if a class dynamically loaded over a network requires an adapter class so that an

instance of the loaded class can communicate with existing objects. Such an adapter class can be created on demand with the Javassist API. Also, this ability enables a distributed object library without a stub generator. For example, Sun Microsystem's RMI system forces programmers to compile proxy classes by the `rmic` command in advance, but the Javassist API allows proxy classes to be dynamically created at runtime.

The runtime assistance by Javassist, however, has limitations because Java is a statically typed language. Instances of a class created at runtime with the Javassist API cannot be bound to variables of that class type. For example, the following program causes a compile error:

```
CtClass ct = new CtClass();
ct.setName("intVector");
ct.setSuperclass(Vector.class);
Class rt = ct.compile();
intVector v
    = (intVector)rt.newInstance();
```

The last line causes a compile error since `intVector` has not been defined yet at compile time. An instance of `intVector` must be bound to a variable of the superclass `Vector`, which exists at compile time. Otherwise, as shown at the beginning, `intVector` must be introduced by an annotated `import` declaration. In this case, `intVector` is created at compile time and regularly imported so that `intVector` can occur in the rest of the program.

The runtime assistance is effective if the name of a created class does not need to occur in a program. A proxy class for distributed computing is one of the examples. Suppose that the interface of a remote object is `Bank`. A proxy class implements the `Bank` interface and an instance of this class is bound to a variable of the `Bank` type. Only `Bank` occurs in a client program but the proxy class is hidden since it is part of implementation details.

3 Runtime Reflection

A number of typical applications of runtime reflection are also applications of Javassist. The Javassist system includes an assistant called `WrapperAssistant`, which is written within the confines of the Javassist

API. This assistant produces a wrapper class of a given class. For example,

```
import sample.Point
by WrapperAssistant(VerboseMetaobj);
```

This declaration creates a wrapper class of a class `sample.Point`. The wrapper class implements interception of method calls by metaobjects, which is a fundamental mechanism of runtime reflection. The metaobjects are instances of `VerboseMetaobj` in this example.

Programmers can create the wrapper class as an independent class or a subclass of `sample.Point`. In either case, the wrapper class provides the same set of methods as `sample.Point`. Those methods delegate all the calls of them to `trapMethodcall()` on metaobjects so that the metaobjects can interpret the method calls.

The created wrapper class is named `wrapper.Point`³ and it is imported instead of the wrapped class `sample.Point`. The `import` declaration is translated by the Javassist compiler into:

```
import wrapper.Point;
```

After the translation, `wrapper.Point` is substituted for `sample.Point` so that all the occurrences of the class name `Point` in the program indicates `wrapper.Point`. Therefore, without modifying a program, programmers can transparently introduce the reflection mechanism implemented by the wrapper class.

4 Implementation Issues

The Javassist system consists of two components: the Javassist compiler and the Javassist engine (Figure 2). The Javassist compiler processes annotated `import` declarations and the Javassist engine compiles `CtClass` objects. In the current implementation, the Javassist engine executes an external Java compiler such as `javac` to compile a `CtClass` object.

³The real implementation uses a different package name.

In a future version, the Javassist engine will be a stand-alone component not using an external compiler. This version will achieve faster compilation because Javassist does not require the full capability to compile a Java program. Especially, Javassist does not allow programmers to directly specify a method body. The methods added to a `CtClass` object perform nothing except calling `trapMethodcall()` on a metaobject. This makes code generation for method bodies very simple and fast. Note that the Javassist engine does not compile a metaobject. It is separately compiled by a regular Java compiler in advance.

5 Concluding Remarks

This paper presented Javassist, which enables programmers to automate definitions of some kinds of classes in the Java language. Javassist is based on our experiences of OpenC++ [1] and OpenJava [2]. However, Javassist is not a compile-time reflective system. The meta programs of Javassist are executed at either compile time or runtime. Furthermore, unlike OpenC++ or OpenJava, a meta-level program of Javassist does not directly deal with source code or a parse tree. In principle, it does not need source files of processed classes. If these classes need to be inspected, Javassist uses the reflection API of the Java language. On the other hand, since Javassist does not perform source-to-source translation, it cannot handle syntax extensions although OpenC++ and OpenJava can do.

Javassist is not a runtime reflective system such as MetaXa [4] and Dalang [5]. It provides a similar capability through `WrapperAssistant` but it can be used for other applications such as parameterize types, which traditional runtime reflective systems cannot implement. Javassist is a system for producing a new class on demand whereas runtime reflective systems are for customizing existing classes.

References

[1] Chiba, S., “A Metaobject Protocol for C++,” in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, no. 10

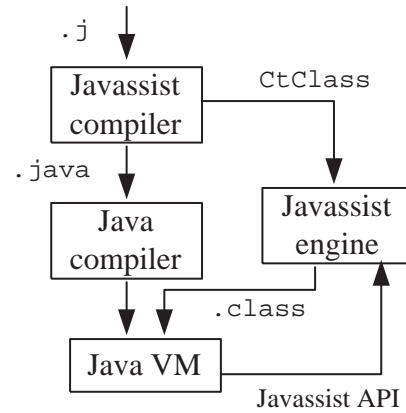


Figure 2: The Javassist System

in SIGPLAN Notices vol. 30, pp. 285–299, ACM, 1995.

- [2] Chiba, S. and M. Tatsubori, “Yet Another `java.lang.Class`,” in *Proc. of ECOOP’98 Workshop on Reflective Object-Oriented Programming and Systems*, July 1998.
- [3] Java Soft, *Java™ Core Reflection API and Specification*. Sun Microsystems, Inc., 1997.
- [4] Kleinöder, J. and M. Golm, “MetaJava: An Efficient Run-Time Meta Architecture for Java,” in *Proc. of the International Workshop on Object Orientation in Operating Systems (IWOOS’96)*, IEEE, 1996.
- [5] Welch, I. and R. Stroud, “Using Metaobject Protocols to Adapt Third-Party Components,” in *Proc. of ECOOP’98 Workshop on Reflective Object-Oriented Programming and Systems*, July 1998.