PPPJ WiP Session — Implementing Dynamic Mixins for the Java Virtual Machine

Kerstin Breiteneder, Christoph Wimberger,
Thomas Würthinger
Institute for System Software
Johannes Kepler University
Linz, Austria
{breiteneder,wimberger,wuerthinger}@ssw.jku.at

ABSTRACT

A mixin is a set of functionalities that can be added to many different classes. Typically, there are two common situations where mixins are used: (1) if a feature should be added to a large number of different classes; (2) if a class should have many optional features. In recent years mixins gained a lot of popularity, particularly in dynamically typed scripting languages such as Ruby or JavaScript. The topic of our research is the integration of dynamic mixins into Java using the Dynamic Code Evolution Virtual Machine (DCE VM) [1]. We aim for a stable, easy to use API and hope for a performance gain in comparison to other approaches.

1 Introduction

Historically, mixins were used to realize a sort of multiple inheritance [2]. They are classes implementing certain aspects that can be added to other classes without having to subclass them from the same super class. This is not possible in languages with single inheritance. In such languages, the code of mixin aspects would have to be pasted into all classes that should have these aspects, which would lead to undesired code duplication. Smalltalk and CLOS were among the earliest languages that supported mixins. Nowadays, mixins can also be found in languages such as Ruby, JavaScript, and Perl.

In Java, mixin behaviour can be partially achieved through interfaces. Extra functionality can be specified as an interface that can be added to classes which should have this functionality. However, the implementation of the interface methods still has to be written manually for every class that implements the interface. In this paper we propose a solution for Java where not only the interface but also the implementation of extra functionality can be added to classes on demand. This can even be done dynamically at run time.

The basic idea of our approach is to add mixin classes to other classes by using dynamic class redefinition. The method call `Mixin.addMixin(A.class, B.class)` redefines class `A` in such a way that it now also contains all fields and methods of class `B`. We first explain this with a motivating example and then sketch the implementation.

2 Motivating Example

The following example shows how mixins can be used in our approach to extend the functionality of an existing class. Assume, that we have a class `Point2D`, which implements two-dimensional points.

```
public class Point2D {
   private int x;
   private int y;

   public int getX() { return x; }
   public void setX(int x) { this.x = x; }
   public int getY() { return y; }
   public void setY(int y) { this.y = y; }
}
```

We would like to extend this class so that it can handle three-dimensional points. First we specify the 3D behaviour by an interface `Point3D`.

```
interface Point3D {
   public int getZ();
   public void setZ(int z);
}
```

Then we implement the 3D behaviour by a mixin class `Mixin3D` which implements the interface `Point3D`.

```
public class Mixin3D implements Point3D {
   private int z;

   public int getZ() { return z; }
   public void setZ(int z) { this.z = z; }
}
```

In order to add the 3D behaviour to the class `Point2D` we call:

```
Mixin.addMixin(Point2D.class, Mixin3D.class);
```

This will add all features of the class `Mixin3D` to the class `Point2D`. In other words, the class `Point2D` is dynamically redefined so that it contains a field `z` as well as the methods `getZ()` and `setZ()` with their bytecodes. Since `Mixin3D` implements the interface `Point3D`, the class resulting from adding `Mixin3D` to `Point2D` will implement `Point3D` as well. We can now create an instance of `Point2D` and use it as a `Point3D` object, for example:

```
Point2D point2d = new Point2D();
Point3D point3d = (Point3D) point2d;

point2d.setX(10);
point2d.setY(20);
point3d.setZ(40);
```

```
System.out.println("Point at " + point2d.getX() + "/" +
                   point2d.getY() + "/" + point3d.getZ());
```

Note, that our approach does not require new Java syntax. A mixin is an ordinary Java class with fields and methods and possibly implementing some interfaces. Also the target class does not need special syntax for allowing it to be extended by mixin classes. Any number of mixin classes can be added to a target class using a sequence of `addMixin()` calls. This happens at run time and allows the dynamic extension of classes with extra functionality.

On the downside, however, it might be difficult to see from the source code, which functionality a class like `Point2D` really has. This depends on whether it has been extended by mixins or not. As a solution to this problem one could add all mixins to a class `C` in the static constructor of `C` so that they can easily be found.

3 Implementation

Since the introduction of the instrumentation API of J2SE 5.0 (1.5.0) in 2004, the Java VM can handle class redefinitions during runtime. However, there are some restrictions: It is not allowed to add, remove or rename fields or methods, to change method signatures or to redefine the inheritance relationship [3]. Therefore, our implementation uses the Dynamic Code Evolution Virtual Machine (DCE VM), [1, 4] which removes these restrictions. An installer that adds the DCE VM features to an installed Java SDK or JRE can be downloaded from `http://ssw.jku.at/dcevm/binaries/`. Once installed, a mixin-enabled VM could be started like this (where `PATH_TO_JRE` is typically the `jre` subdirectory of the Java SDK installation directory or the Java JRE installation directory itself):

```
$ java -javaagent:<PATH_TO_JRE>/lib/ext/dcevm.jar <MainClass>
```

Our implementation can be divided into two parts: The first part handles the creation of the mixed class from the bytecodes of the mixin class and the target class. We use the ASM framework [5] to load the target class and to translate it into a data structure that can be easily modified. Then we add the fields and the methods of the mixin class using the ASM API. Since mixin methods can override methods of the target class we also have to fix certain bytecode instructions. Finally, the data structure is translated back to the Java class file format and loaded by the DCE VM. The other part uses this functionality to implement a Java agent that keeps a list of all mixins for a class and transforms the loaded bytecodes of the target class by mixing to it the bytecodes of all mixin classes in this list.

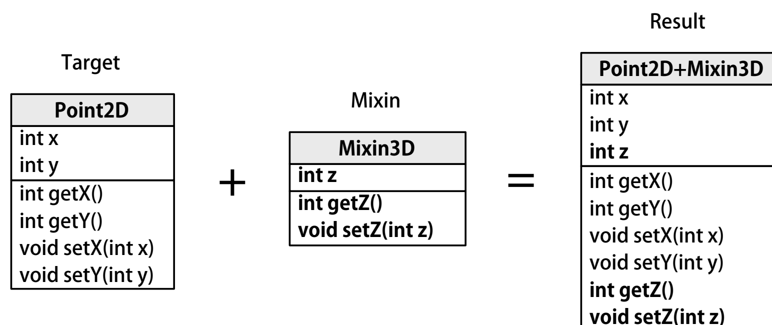

Figure 1: Mixing a class to a target class.

Figure 1 shows how the functionality of a mixin class is added to a target class. The standard mixin process adds all methods and fields of the mixin class to the target class. If a method with the same signature already exists in the target class, it gets replaced. This is not limited to just one mixin class, but multiple mixin classes can be added during run time. The constructor of the mixin class gets invoked automatically right after the constructor of the target class has finished its execution.

4 Future Work

This paper only described the very basic features of our mixin approach. Private fields and methods of the mixin class can be handled in such a way that they do not accidently override fields and methods of the target class. Public fields on the other hand can act as a reference to the field with the same name and type in the target class, so that field values of the target can also be modified. Another nice feature would be the ability to invoke methods of the target class from within the mixin class. But since the mixin class does not know the target class in advance, there is no guarantee that this method even exists. So a way to provide a default implementation of such a method would make sense. To increase the user friendliness it would help if the invocation of the class redefinition in the VM would not need an agent and could be done through a native call to the virtual machine.

Acknowledgments

The work presented was supported by Oracle Corporation and Guidewire Software, Inc.

References

[1] Würthinger T., Wimmer C., and Stadler L. Dynamic code evolution for java. In *8th Intl. Conf. on Principles and Practice of Programming in Java (PPPJ'10)*, Vienna, Austria, September 2010.

[2] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA, 1990. ACM.

[3] Oracle Corporation. *Interface Instrumentation*, 2010. `http://java.sun.com/javase/6/docs/api/java/lang/instrument/Instrumentation.html`.

[4] Institute for System Software, Johannes Kepler University Linz. *Homepage of the Dynamic Code Evolution VM*, 2010. `http://ssw.jku.at/dcevm/`.

[5] OW2 Consortium. *ASM Java bytecode framework*, 2010. `http://asm.ow2.org/`.