TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Institute of Computer Science
Chair of Theoretical Informatics

Type-safe Java bytecode modification library based on Javassist

Bachelor thesis

Student: Sven Laanela
Student code: IAPB020471
Advisor: Pavel Grigorenko

Tallinn
2015

I declare that this thesis is the result of my own research except as cited in the references. The thesis has not been accepted for any degree and is not concurrently submitted in candidature of any other degree.

Signature	
Name	Sven Laanela
Date	May 26, 2015

Abstract

Dynamic (runtime) Java bytecode modification enables various frameworks to greatly enhance Java programs. They can provide logging, caching, dependency injection, object-relational mapping, etc. without cluttering the application source code, or even when the source code is not available. Existing Java bytecode modification libraries do not provide compile-time type safety or IDE-assisted code completion capabilities with regard to the class under modification. This thesis researches a problem of type-safe Java bytecode modification, building on top of Javassist – a Java bytecode modification library – and taking into consideration the most common Java bytecode modification use-cases. Scenarios implemented in Javassist are described and an alternative approach that implements those scenarios in a type-safe manner is proposed. An implementation library is presented as proof of concept for the proposed approach to Java bytecode modification. An analysis of the applicability of the approach is provided, outlining both the benefits and trade-offs when compared to Javassist.

Annotatsioon

Dünaamiline (s.t. rakenduse töötamise aegne) Java baitkoodi muutmine võimaldab oluliselt täiendada Java rakendusi ilma lähtekoodi risustamata. See lubab lisada programmile logimist, puhverdamist, komponentidevahelisi sõltuvusi, objektrelatsioonilist konfiguratsiooni jpm. ilma rakenduse lähtekoodi muutmata – kood keskendub põhiprobleemi lahendamisele. Olemasolevad lahendused ei võimalda kompileerimisaegset modifitseeritava klassi põhist tüübikindlust ega arendustöövahendites koodi automaatset lõpetamist. Seda isegi mitte siis, kui muudetav klass on kompilaatorile nähtav (klassirajal). Käesolev lõputöö uurib tüübikindlat Java baitkoodi modifitseerimist Javassist'i – üks levinumaid Java baitkoodi muutmise teeke – baasil ning võtab aluseks kõige tihemini kasutatavad Java baitkoodi transformeerimise kasutusjuhud. Need stsenaariumid on välja toodud nii Javassisti baasil kui ka alternatiivse lähenemise alusel, mis säilitab kompileerimisaegse tüübikindluse. Tõendamaks väljapakutud lahenduse rakendatavust Java baitkoodi muutmisel, sisaldab lõputöö ka esiletoodud kasutusjuhtusid lahendavat prototüüp-implementatsiooni. Lisaks on võrreldud väljapakutud lahenduse eeliseid ja puuduseid võrreldes Javassistiga.

Contents

1 Introduction

1.1 Background and motivation

Java nowadays is not only a mature programming language, but a platform with well established ecosystem. It allows to develop scalable and distributed applications for almost any domain. Java programs are compiled into bytecode which at runtime is interpreted by the Java Virtual Machine (JVM). Java bytecode is intended to be platform-independent and secure [1]. The bytecode is the driving force which makes Java platform rich in libraries, frameworks and various JVM-based languages. There are a lot of useful frameworks and libraries which rely on dynamic (runtime) bytecode manipulation which enhance and change the behaviour of Java programs. Some examples are object-relational mapping frameworks (Hibenate, etc), aspect-oriented approach for adding logging and caching (AspectJ), dependency injection, application profiling (constructing method execution traces), dynamic code updates and many more. There is a number of bytecode generation and manipulation tools: ASM [2], Javassist [3], BCEL [4], cglib [5].

The present work uses Javassist [3] as a Java library for Java class-file bytecode modification. It allows the developer to programmatically change the structure of a class without requiring extensive knowledge of the Java Virtual Machine Specification [6]. The Javassist API includes constructs for manipulating the class structure and hierarchy, for example `javassist.CtClass`, `javassist.CtField`, `javassist.CtMethod` for modifying the class itself or the fields and methods contained therein. The declaration, definition and access modifiers can be altered by invoking methods such as `ctField.setModifiers(int modifiers)`, `ctMethod.setBody( String body)`, `ctClass.addField(CtField field)`, `ctClass.addMethod( CtMethod method)` on the `Ct*` object.

For example, given a class `SampleClass` with a private field `instanceField` and a public method `instanceMethod(String input)` that uses this instance variable:

```java
public class SampleClass {
  private final String instanceField = "InstanceField";

  private final String instanceMethod(String input) {
    return input + instanceField;
  }

  public final String publicMethod(String input) {
    return instanceMethod(input);
  }
}
```

The Javassist code for getting a handle on the `javassist.CtClass` object repdesigned this class during runtime would look like the following:

```java
public static CtClass getClass(String className) throws
    NotFoundException {
  ClassPool classPool = ClassPool.getDefault();
  return classPool.get(className);
}
```

Subsequently, getting a handle on the `javassist.CtMethod` object representing this class during runtime would look like the following:

```java
public static CtMethod getMethod(String className, String
    methodName, String... argClassNames) throws NotFoundException
    {
  CtClass theClass = getClass(className);
  CtClass[] paramClasses = getClasses(argClassNames);
  return theClass.getDeclaredMethod(methodName, paramClasses);
}
```

And finally, modifying the body of the instanceMethod method to add a null check for the input would be done by:

```java
public void addNullCheckToInstanceMethod() throws Exception {
  CtMethod ctMethod =
      JavassistHelper.getMethod(SampleClass.class,
      "instanceMethod", String.class);
  ctMethod.insertBefore(
    "if ($1 == null) {"+
    "  return null"+
    "}");
}
```

Looking over the above examples and running them with Javassist, the following drawbacks of the API become apparent:

First, supplying Java code as a String argument to the body modification methods leads to runtime syntax checking instead of checking for those errors during compile-time. For example, the above example is missing a semicolon at the end of the `return null` call leading to a `CannotCompileException` thrown when the the `addNullCheckToInstanceMethod()` method is run. To avoid these issues, extra care must be taken to verify that the Java statement supplied as the argument is correct and that there are no typos in the code.

Second, the type-safety of the code is also enforced only during runtime, consequently, method calls and field accesses on the class itself and its contributing classes can also result in a `CannotCompileException` being thrown. Calling `stringVariable.indexof('c');` on a variable of type String is syntactically correct, but since the indexOf method is defined as `indexOf(char c)`, the code fails. Furthermore, care must be taken to either supply a fully-qualified classname of a contributing class, used from the modified method body, or adding an import statement by calling `ClassPool .importPackage(String packageName);`

Third, the bytecode modification construct contains a lot of boilerplate code for looking up the class, the method, and its argument types; making it harder to reason about the actual problem the modification is trying to solve. It is also an unnatural and unintuitive way of writing code for defining what is essentialy an extension or a subclass and the Java developer working with those kinds of constructs is left without IDE support for code completion, syntax checking and debugging.

Practices exist to a certain extent to alleviate the problems above. For larger modifications to the method body it is recommended to implement the modification in a separate Java helper class or static method and insert the call statement of that method into the String passed to Javassist's `insertBefore()`. This ensures syntax checking and type safety in the added helper class, however the wiring – calling the helper method from the existing method – is still prone to errors, and creating the wiring code needs to be added for each class under modification separately, leading to code duplication. Also, calling private methods on the existing class is not possible in this case. The compile-time type safety could be overcome by running the bytecode modification logic during building – either with an annotation processor during compilation, or as a separate build step. In this case, however, the developer has to build a mechanism for enumerating or detecting the bytecode modification methods and ensuring these are run during the build process.

1.2 Goal

The goal of the work is to provide a type-safe alternative to Javassist (building on top of its functionality) for class bytecode modification that would alleviate the drawbacks outlined above and:

- provide a natural extension/subclassing mechanism that is already familiar to Java developers;
- add compile-time syntax checking and type safety;
- leverage IDE support for code completion and refactoring.

1.3 Outline of the thesis

The thesis is organized as follows. The second chapter introduces type systems, Java bytecode modification use-cases with Javassist and the capabilities provided by the Java annotation processor facility. The third chapter describes the proposed implementation for type-safe Java bytecode modification. The fourth chapter analyses the applicability of the selected approach when compared to Javassist. The fifth chapter gives a brief overview of other major Java bytecode modification libraries. Finally, sixth chapter concludes the thesis and lists possible future improvements of the type-safe Java bytecode modification library.

2 Prerequisites

2.1 Type systems

Modern software engineering recognizes a broad range of formal methods that ensure a system behaves correctly with respect to some specification, implicit or explicit, of its desired behavior. A "type system", according to Benjamin Pierce, is: "a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute" [7].

Programming languages can be either statically typed or dynamically typed:

- A "statically typed programming language" is a language with the sorts of compile-time analyses that are used to prove the *absence* of some bad program behavior.
- A "dynamically typed programming language" is a language where run-time *type information* is used to distinguish different kinds of structures in the heap during program execution. The term "dynamically typed" is a misnomer and should probably be replaced by "dynamically checked".

2.1.1 Advantages of static typing

The most obvious benefit of static typechecking is that it allows early detection of some programming errors. Errors that are detected early can be fixed immediately, rather than lurking in the code to be discovered much later, when the programmer is in the middle of something else – or even after the program has been deployed. Moreover, errors can often be pinpointed more accurately during typechecking than at run time, when their effects may not become visible until some time after things begin to go wrong. [7]

In practice, static typechecking exposes a surprisingly broad range of errors. Programmers working in richly typed languages often remark that their programs tend to "just work" once they pass the typechecker, much more often than they feel they have a right to expect. One possible explanation for this is that not only trivial mental slips (e.g. forgetting to convert a string to a number before taking its square root), but also deeper conceptual errors (e.g., neglecting a boundary condition in a complex case analysis, or confusing units in a scientific calculation), will often manifest as inconcsistencies at the level of types.

For some sorts of programs, a typechecker can also be an invaluable maintenance tool. For example, a programmer who needs to change the definition of a complex data structure need not search by hand to find all the places in a large program where code involving this structure needs to be fixed. Once the declaration of the datatype has been changed, all of these sites become type-inconsistent, and they can be enumerated simply by running the compiler and examining the points where typechecking fails.

Another important way in which type systems support the programming process is by enforcing disciplined programming. In particular, in the context of large-scale software composition, type systems form the backbone of the module languages used to package and tie together the components of large systems. Types show up in the interfaces of modules (and related structures such as classes); indeed, an interface itself can be viewed as "the type of a module," providing a summary of the facilities provided by the module – a kind of partial contract between implementors and users.

Finally, types are also useful when reading programs. The type declarations in procedure headers and module interfaces constiture a form of documentation, giving users hints about behavior. Moreover, unlike descriptions embedded in comments, this form of documentation cannot become outdated, since it is checked during every run of the compiler.

In conclusion, the advantages of static typing are significant in ensuring program correctness and early detection of programming errors. Java is a statically typed language and the developers are used to relying on type errors being reported early. Still, most Java bytecode modification libraries are not type-safe with regards to classes they are modifying and this is exactly the problem this thesis is attempting to address.

2.2 Javassist

The present section demonstrates various use-cases of the Javassist library's source-level application programming interface for Java bytecode manipulation.

Getting a reference to a `javassist.CtClass` object which represents a Java class being instrumented is done via the `javassist.ClassPool` class. The ClassPool is initialized with the classpath of a specific classloader, or with a custom one.

```java
public static CtClass getClass(Class<?> clazz) throws
    NotFoundException {
  return getClass(clazz.getName());
}

public static CtClass getClass(String className) throws
    NotFoundException {
  ClassPool classPool = ClassPool.getDefault();
  return classPool.get(className);
}
```

Javassist library allows extensive class bytecode modification, but this work focuses on extending the functionality of existing classes and not on the facilities for changing names of an existing classfiles, or creating new classfiles programmatically.

2.2.1 Special identifiers

This section demonstrates some special identifiers available in the body of a `CtMethod.insertBefore(String body)`, `CtMethod.insertAfter(String body)`, `CtMethod.insertAt(String body)` or `CtMethod.addCatch(String body)` call. This is not an exhaustive list, the full selection of available identifiers is available in Javassist tutorial [8]

1. `$0` – the `this` variable. Unavailable in the context of a static method
2. `$1,$2,...` – the method arguments, `$1` being the first argument to the method call
3. `$$` – all method arguments as `Object[]`. calling `someOtherMethod($$)` calls the other method with the same list of arguments as this method
4. `$_` – the return value of the method. In the context of an `insertAfter(String body)`, this identifier is a reference to the return value and assigning to this identifier would override the return value.
5. `$proceed` – in the context of a method instrument block, calling `$proceed()` executes the method whose call was intercepted.

2.2.2 Method body manipulation

Getting a reference to a `javassist.CtMethod` object is done by calling the method `ctClass.getDeclaredMethod(String methodName, CtClass[] methodParams);` of the `javassist.CtClass` object that you want to modify.

```
public static CtMethod getMethod(Class<?> clazz, String
    methodName, Class<?>... argClasses) throws NotFoundException {
  return getMethod(clazz.getName(), methodName,
      toClassNames(argClasses));
}

public static CtMethod getMethod(String className, String
    methodName, String... argClassNames) throws NotFoundException
    {
  CtClass theClass = getClass(className);
  CtClass[] paramClasses = getClasses(argClassNames);
  return theClass.getDeclaredMethod(methodName, paramClasses);
}
```

Similarly, a reference to a `javassist.CtConstructor` object is done by calling `ctClass.getDeclaredConstructor(CtClass[] constructorParams);`

```
  public static CtConstructor getConstructor(Class<?> clazz,
      Class<?>... argClasses) throws NotFoundException {
    return getConstructor(clazz.getName(),
        toClassNames(argClasses));
  }

  public static CtConstructor getConstructor(String className,
      String... argClassNames) throws NotFoundException {
    CtClass theClass = getClass(className);
    CtClass[] paramClasses = getClasses(argClassNames);
    return theClass.getDeclaredConstructor(paramClasses);
  }
```

The simplest example of a method body modification is replacing the existing method body entirely:

```
public void overwriteMethod() throws Exception {
  CtMethod method = JavassistHelper.getMethod(SampleClass.class,
      "instanceMethod", String.class);
  method.setBody("return \"Hello world!\";");
}
```

Adding a code block in the beginning of a method:

```
  public void addNullCheckToInstanceMethod() throws Exception {
    CtMethod ctMethod =
        JavassistHelper.getMethod(SampleClass.class,
        "instanceMethod", String.class);
    ctMethod.insertBefore(
      "if ($1 == null) {"+
      "  return null;"+
      "}");
  }
```

We already looked at how an application could insert *before* advice to a method, *after* advice (modifying the return value of the `instanceMethod(String input)` of `SampleClass` by adding an exclamation mark) is done similarly:

```
public void addSuffixToInstanceMethodReturnValue() throws Exception
    {
```

```
  CtMethod ctMethod = JavassistHelper.getMethod(SampleClass.class,
      "instanceMethod", String.class);
  ctMethod.insertAfter("$_ = $_ + \"!\";");
}
```

If we want to add code before the method and after the method, or wrap the method call into a try/catch block, we cannot do that with the `insertBefore()` and `insertAfter()` constructs. Instead we have to create a copy of the method via `javassist.CtNewMethod.copy()` method, overwrite the original method body and call the newly added copy method where required.

```
public void addTryCatchToMethodCall() throws Exception {
  CtClass ctClass = JavassistHelper.getClass(SampleClass.class);
  CtMethod ctMethod = JavassistHelper.getMethod(SampleClass.class,
      "instanceMethod", String.class);
  ctClass.addMethod(CtNewMethod.copy(ctMethod,
      "instanceMethod_copy", ctClass, null));
  ctMethod.setBody(
      "try {"+
      "  instanceMethod_copy($$);"+
      "} catch (Exception e) {"+
      "  e.printStackTrace();"+
      "}");
}
```

2.2.3 Adding new fields and methods

Adding an entirely new method to a class and calling that method from the same class is done by creating a new `javassist.CtMethod` via `javassist .CtMethod.make(String src, CtClass declaring)` with the method body, adding that to the class and calling this new method via the method body bytecode manipulation methods listed above.

```
public void addTrimMethodAndUseLocally() throws Exception {
  CtClass ctClass = JavassistHelper.getClass(SampleClass.class);
  ctClass.addMethod(CtNewMethod.make(
    "public String trim(String input) {"+
    "  if (input == null) {" +
    "    return null;"+
    "  }"+
    "  return input.trim();"+
    "}", ctClass));
  CtMethod ctMethod = JavassistHelper.getMethod(SampleClass.class,
      "instanceMethod", String.class);
  ctMethod.insertBefore("$1 = trim($1);");
}
```

However, when another class wants to access a method added to some class (like above), there is a trick that needs to be used. Calling a newly added method of another class throws a CannotCompileException during execution since Javassist does not heuristically scan what methods were added to other classes when modifying the bytecode of some class – the class definitions for other classes that the bytecode modification code can depend on is the definition of that class during load time. To get around this limitation, one option is to create a Java interface that defines the signature of the added method:

```
public interface Trimmable {
  public String trim(String input);
}
```

Add this interface to the class where our new method is defined and then call the added method from another class by casting an instance of the original class to the interface:

```
public void addTrimMethodAndUseExternally() throws Exception {
  CtClass ctClass = JavassistHelper.getClass(SampleClass.class);
  // adding the trim method body omitted for brevity

  ctClass.addInterface(JavassistHelper.getClass(Trimmable.class));

  CtMethod ctMethod = JavassistHelper.getMethod(OtherClass.class,
      "otherMethod", String.class);
  ctMethod.insertBefore("$1 = (("+Trimmable.class.getName()+") new
      SampleClass()).trim($1);");
}
```

Adding a new method and accessing it locally is similar to adding a new method. However using the newly added field from another class directly is not possible. To access it from another class, a getter method (and an interface declaring that method) must be added to the class and the method called similar to above.

```
public void insertFieldUseLocally() throws Exception {
  CtClass ctClass = JavassistHelper.getClass(SampleClass.class);
  ctClass.addField(CtField.make(
      "private String addedField = \"addedField\";",
      ctClass));

  CtMethod ctMethod = JavassistHelper.getMethod(SampleClass.class,
      "instanceMethod", String.class);
  ctMethod.insertBefore("$1 = $1 + addedField;");
}
```

2.2.4 Method and Constructor instrumentation

Javassist provides a mechanism for instrumenting a specific method body and overriding method calls, field accesses and other calls done inside the body of that method. For example, given a class InstrumentClass with the following definition:

```
public class InstrumentClass {
  private String field = "field";

  private String method1() {
    return "method1";
  }

  private String method2() {
    return "method2";
  }

  public String instrumentMethod() {
    return field + method1();
  }
}
```

We can override the calls to `method1()` inside the method body of `instrumentMethod();` to call `method2()` instead:

```
public void instrumentMethodCall() throws Exception {
  CtMethod method =
      JavassistHelper.getMethod(InstrumentClass.class,
      "instrumentMethod");
  method.instrument(new ExprEditor() {
    public void edit(MethodCall m) throws CannotCompileException {
      if ("method1".equals(m.getMethodName())) {
        m.replace("$_ = method2($1);");
      }
    }
```

```java
  });
}
```

Similarly, it is possible to override or modify the field accesses present in a method call. For example, to return an empty string instead of a null value from the field access, the method body could be instrumented like this:

```java
public void instrumentFieldAccess() throws Exception {
  CtMethod method =
      JavassistHelper.getMethod(InstrumentClass.class,
      "instrumentMethod");
  method.instrument(new ExprEditor() {
    public void edit(FieldAccess f) throws CannotCompileException {
      if ("field1".equals(f.getFieldName())) {
        f.replace(
            "String s = $proceed();"+
            "if (s == null) {"+
            "  $_ = \"\";"+
            "} else {"+
            "  $_ = s;"+
            "}");
      }
    }
  });
}
```

The full list of expressions that can be overridden/modified can be found for the ExprEditor class [9] in the Javassist API documentation.

2.2.5 Modifying access modifiers

As with reflection, Javassist allows the developer to modify the modifiers of an existing class, field or method. For example:

```java
public void addSynchronizedMakePublic() throws Exception {
  CtField field = JavassistHelper.getField(SampleClass.class,
      "instanceVariable");
  int fieldModifiers = field.getModifiers();
  field.setModifiers(Modifier.setPublic(fieldModifiers));

  CtMethod method = JavassistHelper.getMethod(SampleClass.class,
      "instanceMethod", String.class);
  int methodModifiers = field.getModifiers();
  method.setModifiers(methodModifiers | Modifier.SYNCHRONIZED);
}
```

2.2.6 Type-safety of Javassist API

From the examples above it is clear that Java source code written in strings and passed to Javassist methods in unreliable and error-prone. Type and syntax errors are revealed only at runtime upon class loading instead of failing during the compilation phase.

2.3 Java Annotation Processing

Java Annotation Processor [10] is a mechanism for interfacing with the Java bytecode compilation process. With the introduction of annotations to the Java Language Specification and the Java Virtual Machine Specification, a separate tool - `apt` (short for Annotation Processing Tool) - was introduced. In JDK7, the separate apt tool was deprecated and the annotation processing step was incorporated into the *javac* compiler.

The *javac* compiler can be configured to load specific annotation processors from user-specified .jar or .class files or use its own automated discovery mechanism that searches the classpath for `META-INF/services/javax.annotation.processing.Processor` files containing fully-qualified classnames of the annotation processor classes that should be registered to the compiler.

At a high level, the compilation process consists of the following steps [11]:

1. Compilation begins
2. A list of classes (types) and the annotations of those classes and their members are generated
3. For each annotation, the annotation processors registered for that annotation are looked up
4. The annotation processors found are executed
5. If any new classes were generated during the current annotation processing round, a new annotation processing round is started to also process the generated classes
6. All .class files for existing and newly generated classes are generated
7. Compilation ends

2.3.1 Annotation Processing Rounds

Annotation processing consists of a series of *rounds* (iterations) during which the annotation processor can inspect the program elements (fields, methods, variables, etc.) annotated with a given annotation and take actions such as registering warnings or errors, generating additional class files, or invoke custom code. After the round has completed and all annotation processors have been given a chance to do their work on currently existing classes, if a new set of classes were created, a new annotation processing round is started for the newly created classes. If no new classes were created during an annotation processing round, that round will be the "final" round of annotation processing.

2.3.2 Available APIs and Capabilities

The annotation processor has access to various utilities via the `javax.annotation.processing.ProcessingEnvironment` class:

- `javax.annotation.processing.Filer` [12] – allows creating new java source and class files as well as auxiliary resource files
- `javax.annotation.processing.Messager` [13] – allows registering errors, warnings and informational messages to elements of a type
- `javax.lang.model.util.ElementUtils` [14] – contains utility methods for operating on program elements (types, methods, fields, variables, etc.)
- `javax.lang.model.util.TypeUtils` [15] – contains utility methods for operating on types

3 Implementation

At a high level, the implementation for the type-safe Java bytecode modification library depends on the following stages:

- Mirror class generation - creating a class based on the original class with the final modifiers removed and visibility modifiers relaxed for the methods, fields and constructors of the original class
- Extension class validation - additional type-safety verification of the extension class
- Wiring code generation - generating Javassist wiring code for the original class that redirects method execution/field access to the extension class
- Runtime lookup and wiring code execution - executing the Javassist wiring code and loading the extension class alongside the original class at runtime

The annotation processor that implements this is located in the class `TypesafeBytecodeModificationProcessor.java` (Appendix D)

3.1 Mirror class generation

The goal of the type-safe Javassist library is to allow developers to write an `extension class` that subclasses naturally from the `original class`. This allows the developer to use IDE-provided code completion and refactoring tools for overriding methods of the `original class` in a type-safe manner – with the `@Override annotation` –, as well as to invoke methods on the `original class` from the `extension class` as usual when writing a subclass. One of the use-cases demonstrated in the introduction to Javassist (listing 9) demonstrated modifying the body of a method of the `original class` to include a null check before the method body. A natural way for a Java developer to implement the above use-case would be to extend from the `original class`, override the original method, add the null check and then call the `super.methodName()` to return execution the original method.

An example for the above use-case using class extension would look like the following:

```java
public class SampleClassExtension extends SampleClass {
  @Override
  public String publicMethod(String input) {
    if (input == null) {
      return null;
    }
    return super.publicMethod(input);
  }
}
```

The code above works with public, protected or package-private methods – if the extension class is placed in the same package as the original. Javassist, however, also supports scenarios that cannot be implemented with an `extension class`. For example, modifying the private methods of the `original class`, or calling private methods of the `original class` from overridden methods of an `extension class` is not possible. Furthermore, private classes, fields, methods and constructors pose problems for this approach. Also, final classes cannot be subclassed and final fields and methods could only be used from other methods but not overriden with a different implementation.

These problems could be overcome by generating a `mirror class` for the `original class`. This `mirror class` would include all the fields, method declarations and constructors similar to the original class, but with the final modifiers removed and the private modifiers changed to protected (for override and access from an `extension class`). The `mirror class` is only required during compilation – it will be discarded later on – and only needs to adhere to the signature of the `original class`. This means that the `mirror class` should extend from the parent class of the `original class` – this is possible since the `mirror class` will be generated into the same package as the `original class` – and implement the interfaces that the `original class` implements, however it can discard the logic inside the constructors, methods or field declarations by just returning null (or a primitive type default value).

For example, the mirror for SampleClass with the private modifiers of methods set to protected and the final modifiers removed for the method `instanceMethod(String input)` would look like the following:

```
public class SampleClass_Mirror {
  protected String instanceField = null;

  protected String instanceMethod(String input) {
    return null;
  }

  public String publicMethod(String input) {
    return null;
  }
}
```

Generating this mirror class is achieved by annotating the `extension` `class` with a special `@org.zeroturnaround.javassist.Patches(Class<?> value)` annotation (Appendix A) and supplying the class under modification as an argument to it. This requires the original class to be on the class path during compilation and makes use of a custom annotation processor that is registered to processes classes marked with the `@Patches` annotation. During the annotation processing step of the compilation cycle, the compiler detects that the `extension class` is annotated with `@Patches`, reads the class from the compiler classpath, inspects the class structure with Javassist and generates the `mirror class` for the `original class` using the `javax .annotation.processing.Filer` API. This generated `mirror class` can then be used as the actual parent class for the `extension class`.

The mirror class generation is illustrated in the Figure 1.

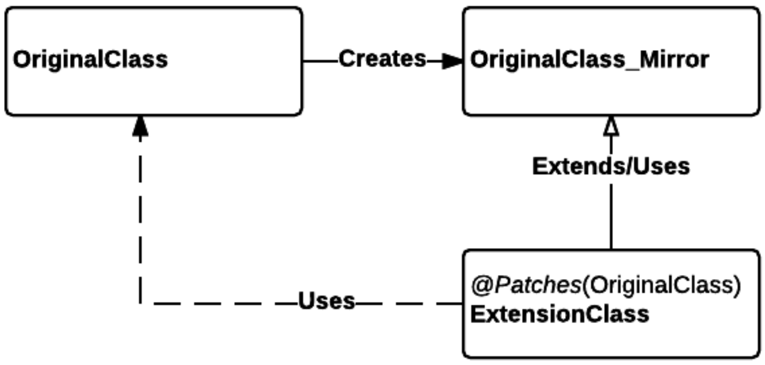


Figure 1: Generating a mirror class based on the original class.

For example, an `extension class` for SampleClass overriding the method `private final String instanceMethod(String input)` – instead of `public String publicMethod(String input)` as in the example above – would look like the following:

```java
@Patches(SampleClass.class)
public class SampleClassExtension2 extends SampleClass_Mirror {
  @Override
  protected String instanceMethod(String input) {
    if (input == null) {
      return null;
    }
    return super.instanceMethod(input);
  }
}
```

The mirror class source code generation logic is located in the class `MirrorClassGenerator` (Appendix E).

3.1.1 Namespacing and registry

A `mirror class` is generated at compile-time for an `original class` supplied as an argument to the `@Patches(Class<?> value)` annotation of an `extension class`. The `mirror class` is generated inside the same package as the `original class` to mimic the package visibility (and to a lesser extent, the private visibility) of the `original class`. The `mirror class` name is the same as the `original class` name, but with a suffix `_Mirror`. For example a mirror generated for the SampleClass listed above would be named `SampleClass_Mirror`. For the purposes of the prototype implementation, this gives us a sufficient uniqueness, however in the real world, the libraries under modification could include classes with the same `_Mirror` suffix. Generating a more unique suffix – or making it configurable – to help with those kinds of scenarios is not in the scope for this thesis and will be implemented at a later time.

Another consideration while generating a `mirror class` for an `original class` is that there could exist more than one `extension class` for the same `original class`. In order to avoid multiple `mirror classes` being created during compile-time, there has to exist some registry that holds the names of the `original classes` for which a `mirror class` has already been built.

3.1.2 Constructor/method/field generation considerations

All constructors, fields and methods that are available in the `original class` should also exist in the `mirror class` since the developer may wish to override/call those methods or access those fields in the `extension class` code. If a field of a class or a method/constructor argument is of a type for which a `mirror class` should be generated – due to some other `extension class` containing `@Patches(Class<?> value)` of that class, – the type of that field or argument is set to the `mirror class` of that type.

3.1.3 Superclasses without a default constructor

The `mirror class` is set to extend from the actual parent class of the `original class` so that the `extension class` extending from the `mirror class` could also see the methods defined in the parent of the `original class` – similar to extending from the `original class` itself. When the parent class of an `original class` that a mirror is generated for, includes a default no-args constructor, the constructors in the `mirror class` can be left without a body, or just include a call to `super()`.

However, if the parent of the original does not have a default no-args constructor, the parent class needs to be scanned for an eligible constructor to be callend in the constructors of the `mirror class` (otherwise the `mirror class` declaration would not be valid). Note that it is not possible to rely on the parent class having a constructor with the exact same argument list as the constructor of the `original class` – it may call `super()` with a different argument list.

Also, it is convenient to add a default no-args constructor inside the `mirror class` – even if one does not exist in the `original class` – so it would not be necessary to declare a constructor in the `extension class` just to call a non-default constructor in the `mirror class`.

3.1.4 Inner classes

An `original class` containing static or non-static inner classes pose some problems for the generation of the `mirror class`. First of all, the inner class itself could be private, which – if the inner class is used as a method/constructor argument, return type or a field declaration type in the `original class` – leads to visibility errors in the `mirror class` when using the inner class directly. For this reason, any inner classes defined inside the `original class` require a `mirror class` of their own and all method/constructor arguments, return types or field declaration types inside the `mirror class` of the `original class` should be substituted with the `mirror class` types of those inner classes.

3.1.5 Synthetic methods/constructors

A synthetic method/constructor is a method/constructor that does not exist in the Java source code for a class, but can be generated by the compiler during compilation. A private member of a class is not accessible by another class except when a class itself is either an inner class or an outer class of the other class – or more specifically, is in a hierarchy of inner/outer classes. This behavior is specified by the Java Language Specification [16], however the Java Virtual Machine Specification [6] does not support this. A synthetic method is a trick used by the Java compiler to bypass this visibility limitation for inner/outer classes and is generated into the bytecode of an inner or outer class during compile-time. Since a private field/method of a class is not directly accessible from another class, the compiler generates a method named `access${NUMBER}()` with package-level visibility to the class whose private member is accessed and the calls to the private member from the other class are replaced to call this synthetic method instead. The synthetic method is then able to access the private member on the class directly. A synthetic constructor is generated inside the bytecode of an inner class for all declared constructors of the inner class if the inner class includes a call to some private method of the outer class somewhere in its code. When the outer class instantiates the inner class, it will use the synthetic version of the constructor and supply an instance of itself as an additional argument to the constructor.

During `mirror class` generation the synthetic constructors and methods present in the bytecode of the `original class` can be ignored.

3.1.6 Library/class versions

The current implementation of the type-safe bytecode manipulation library supports generating a `mirror class` for a specific version of the `original class` which is found in the classpath of the application. If multiple versions of the same `original class` are present in the classpath, the specific version that will be used to generate the `mirror class` is unspecified. Furthermore, adding support for either a single `extension class` manipulating different versions of the same `original class`, or multiple `extension classes` manipulating different versions of the `original class`, is not in scope for this implementation and is something that could be researched in the future.

3.2 Extension class capabilities

The goal of the type-safe bytecode modification library is to provide a way for Java developers to modify the bytecode of a class in a way that is similar to writing a custom subclass for that class. This section presents examples of how the same code modification use-cases, which were introduced in the background section for Javassist, could be written using an `extension classes`.

3.2.1 Method modification

The first Javassist example demonstrated how a developer using Javassist would achieve replacing the method body of `SampleClass.instanceMethod(String input)`. In the context of a method override, replacing the body of the method comes naturally:

```java
@Patches(SampleClass.class)
public class OverwriteMethod extends SampleClass_Mirror {
  @Override
  String instanceMethod(String input) {
    return "Hello world!";
  }
}
```

Adding code that is executed just “before” a method could be achieved by overriding the method, modifying the method argument and then calling `super.instanceMethod(String input)` with it:

```
@Patches(SampleClass.class)
public class BeforeMethod extends SampleClass_Mirror {
  @Override
  String instanceMethod(String $1) {
    if ($1 == null) {
      return null;
    }
    return super.instanceMethod($1);
  }
}
```

Similarly, adding code that is executed “after” a method immediately before it returns could be achieved similarly. Modifying the return value of the method could be achieved by calling `super.instanceMethod(String input)` first and then appending the exclamation mark to the return value of that:

```
@Patches(SampleClass.class)
public class AfterMethod extends SampleClass_Mirror {
  @Override
  String instanceMethod(String input) {
    return super.instanceMethod(input) + "!";
  }
}
```

Adding code both before and after a method (such as a try-catch block) with Javassist was verbose and required creating a copy of the method and overwriting the original body to include a method call to the copy when required. With the extension class approach, the same kind of code modification could be achieved by:

```
@Patches(SampleClass.class)
public class AroundMethod extends SampleClass_Mirror {
  @Override
  String instanceMethod(String $1) {
    try {
      return super.instanceMethod($1);
    } catch (Exception e) {
      e.printStackTrace();
      return null;
    }
  }
}
```

3.2.2 Static methods

The above examples all demonstrated how a Java developer would consider extending the functionality of the instance methods of the `original class`. Modifying an existing static method call could look similar, with the exception that the IDE does not provide code completion capabilities for overwriting static methods in a subclass. Furthermore, the compiler does not enforce that the signature of a static method declaration in the `extension class` matches the static method declaration in the parent class, when using the `@Override` annotation – and actually registers an error during compile-time if one is added. To ensure compile-time type-safety, a new annotation `org.zeroturnaround.javassist.annotation.Modify` is introduced (Appendix B). The existence of this annotation and the existence of a matching static method inside the `original class` can now be verified by the annotation processor.

For all methods (both instance and static) of the `extension class` annotated with the `@Modify` annotation, the annotation processor should verify that the `original class` does indeed include a method with the same signature. If such a method does not exist, it should register a compilation error for the method declaration in the `extension class`. Similarly, if a method of the `extension class` does not have the `@Modify` annotation, a method with the same signature must not exist in the `original class`. If such a method does exist, the annotation processor should register a compilation error for the method declaration in the `extension class`. In this case the override relationship is made explicit and the annotation processor can notify the developer when he happens to make a mistake.

If either the overridden static method itself or some other static method of the `original class` needs to be called from a static method override of the `extension class`, it should be done as usual – by calling the static method on the parent of the `extension class`, i.e. the `mirror class`. Using the `@Modify` annotation introduced above, an example of a static method override that adds an exclamation mark to the return value of the original method should look like the following:

```java
@Patches(SampleClass.class)
public class StaticMethod extends SampleClass_Mirror {
  @Modify
  public static String staticMethod(String $1) {
    return SampleClass_Mirror.staticMethod($1) + "!";
  }
}
```

3.2.3 Field modification

Java does not provide a mechanism for modifying the field types/values of a class from a subclass. If a field with the same name and type is defined both in a local class and its parent, then the methods defined in the local class use the field of the local class whereas the methods defined in the parent class use the field of the parent class. Since Javassist however provides some support for field modification, the `extension class` mechanism should also emulate that.

Field declarations in the `original class` can be modified with Javassist by calling `CtField.setName(String newName)` and `CtField.setType(CtClass clazz)`. Field values, however, cannot be modified directly on the `CtField` object, but instead should use a `CtConstructor` object as the initialisation is moved to the constructor by the Java compiler. Updating the values of those fields however is not straightforward since the constructor logic and the interdependencies between field initialisations can be quite complex. For example, the following examples produce the same bytecode:

```java
public class FieldInit {
  private String first = "First";
  private String second = "Second";
}
```

```java
public class FieldInitInConstr {
  private String first;
  private String second;
  public FieldInitInConstr() {
    first = "First";
    second = "Second";
  }
}
```

Things get difficult when one of the fields depends on the value of the other during initialisation. For example:

```java
public class FieldInitDep {
  private String first = "First";
  private String second;
  public FieldInitDep() {
    String s = first + ",";
    second = s + "Second";
  }
}
```

In this case, since the value of the second field depends on the value of the first (and not vice versa) an `extension class` implementation attempting to provide a correct extension/replacement mechanism for field values of the `original class` – for example, to change the value of the field named “first” and still have the value of the field named “second” be calculated based on that – requires extensive analysis of the interdependencies between the fields in the constructor call. This is even more difficult if the fields are initialised (or reinitialised) inside different constructors. Due to the complexity of implementing a correct solution for all cases, the solution outlined in this thesis will simplify the problem scope so that if an `extension class` defines a field that exists in the `original class` and initialises it to some value other than `super.fieldName`, the field value will be initialised to this value after the constructor in the `original class` has finished executing. If the value of this field is used to calculate the value of another field in the constructor of the `original class`, then the old value of the field will be used.

Modifying the value of the field of an `original class` inside the `extension class` depends on the field type and name matching between the original and the extension, as well as the field being annotated with `@Modify`. The semantics of using the `@Modify` annotation is the same as when used with static methods and the annotation processor should verify that a matching field either exists or does not exist in the `original class`. An example modifying the value of `SampleClass.instanceField` would look like the following:

```
@Patches(SampleClass.class)
public class OverwriteField extends SampleClass_Mirror {
  @Modify
  String instanceField = "updatedField";
}
```

3.2.4 Constructor modification

There are two types of initialisers in Java – class initialisers (also called static initialiser) and instance initialisers (also called constructors). Javassist provides capabilities for modifying these initialisers similar to what can be done with methods (with some limitations) and the type-safe library should emulate this.

The static initialiser of a class is run when a Java classloader creates a class definition object from class bytecode for the first time. The static initialiser initialies the static fields defined in the class and runs the code inside a `static {}` block (if one exists). The JVM ensures that all classes (up to `java.lang.Object`) are loaded by the time the class loads and that the static initialisers of those classes are executed. The class can access static fields and methods of the parent class (based on visibility rules) during its initialisation. In the context of an `extension class`, adding "after" advice to a static initializer of an `original class` could be done simply by adding a static block to the `extension class`. Based on the static initialiser execution order, the static initialiser of the `original class` is run first and then the static initialiser of the `extension class` is given a chance to run – optionally reading from or writing to static field values defined by the parent class. Adding "before" advice to a static initialiser does not make sense – there are no arguments that the static initializer is run with and if a "before" advice would set a value to some static field, then that field would be overwritten by the static initialiser of the `original class`. Also, entirely replacing the static initializer of the `original class` is currently not supported due to the difficulty of implementing this while being a rare use-case and is left as an opportunity for future enhancement.

Instance initialiser/Constructor modification is more difficult and designing the API for this use-case in the context of an `extension class` requires some consideration due to a class having potentially multiple constructors with different argument lists and these can call either an another constructor of the same class via `this(args)` or a constructor on the parent class via `super(args)`. Calling `this(args)` or `super(args)` as the first statement in the constructor is required and enforced by the compiler. If the parent class has a sufficiently visible default no-args constructor, an explicit call to `super()` can be omitted from a constructor, but the compiler will still add an implicit `super()` call as the first statement of a constructor during compilation.

For example, a class with 2 constructors – `ExampleClass()` and `ExampleClass(String)` – and a parent with a single constructor – `ParentClass(String)` – where `ExampleClass()` calls `this(String)` which in turn calls `super(String)`, the following chain of events can be observed when `new ExampleClass()` is called:

1. `ExampleClass()` starts executing, delegates to `ExampleClass(String)` via `this(String)`

2. `ExampleClass(String)` starts executing, delegates to `ParentClass(String)` via `super(String)`

3. `ParentClass(String)` calls some `this` or `super` of its own, after which it completes and returns execution to `ExampleClass(String)`

4. `ExampleClass(String)` runs the statements after `super(String)` and returns execution to `ExampleClass()`

5. `ExampleClass()` runs the statements after `this(String)`. The object is now fully constructed

Based on the above, every constructor invocation eventually ends up in a single `super(args)` call and that `super(args)` constructor is the first thing that is run during object initialisation – after the redirects via `this(args)`. Only after the first `super(args)` is called and the parent class initialised, can the object itself be initialised. In the context of an `extension class` constructor it is natural to expect that the `original class` constructor will always be run first, and the constructor override in the `extension class` forms an "after" advice to original constructor.

Modifying a constructor of an `original class` requires that the constructor in the `extension class` is marked with the `@Modify` annotation similar to method and field modifications. If the `original class` is constructed via a constructor that does not include an override in the `extension class`, the default constructor of the `extension class` will be invoked after the constructors of the `original class` have finished. If the `extension class` does not include a default no-args constructor, one will be generated for it during compilation by the annotation processor.

A subclass constructor can redirect execution to a different parent constructor either directly via calling `super(args)` with a different argument list, or indirectly via a chain of calls to `this(args)` which eventually calls some `super(args)`. Allowing constructor redirection in the context of the `extension class` is out of scope for this thesis and left as an opportunity for further enhancement.

An example `extension class` modifying the constructor of the `original class` and adding code "after" it looks like the following:

```java
@Patches(ConstructorClass.class)
public class AfterConstructor extends ConstructorClass_Mirror {
  @Modify
  public AfterConstructor(String $1) {
    super($1);
    System.out.println("Constructing");
  }
}
```

3.2.5 Adding new methods

Adding new methods with Javassist was verbose and required multiple lines of code. A Java developer writing an `extension class` would expect to add a new method to a subclass of the `original class` by just adding a method definition:

```
@Patches(SampleClass.class)
public class InsertMethodUseLocally extends SampleClass_Mirror {
  public String trim(String input) {
    if (input == null) {
      return null;
    }
    return input.trim();
  }

  @Modify
  String instanceMethod(String $1) {
    $1 = trim($1);
    return super.instanceMethod($1);
  }
}
```

Using an instance method added to some `original class` from another class, the object of the `original class` should be cast to the `extension class` type and the added method callend on that:

```
@Patches(SampleClass.class)
public class InsertMethodUseExternally extends SampleClass_Mirror {
  public String trim(String input) {
    if (input == null) {
      return null;
    }
    return input.trim();
  }
}

@Patches(OtherClass.class)
class UseAddedMethodInOtherClass extends OtherClass_Mirror {
  @Modify
  public String otherMethod(String $1) {
    $1 = ((InsertMethodUseExternally) new SampleClass()).trim($1);
    return super.otherMethod($1);
  }
}
```

For static methods, adding the method can be done in a similar manner and can be called on the `extension class` directly.

3.2.6 Adding new fields

Adding new fields to an `extension class` is done similarly to new methods, by just adding a field definition:

```
@Patches(SampleClass.class)
public class InsertFieldUseLocally extends SampleClass_Mirror {
  private String addedField = "addedField";

  @Modify
  String instanceMethod(String $1) {
    $1 = $1 + addedField;
    return super.instanceMethod($1);
  }
}
```

Accessing a field added to `original class` from another class can be done by casting an instance of the `original class` to the `extension class` type and invoking the field to the extension type directly. For static fields, the fields can be accessed on the `extension class` type directly.

```
@Patches(SampleClass.class)
public class InsertFieldUseExternally extends SampleClass_Mirror {
  public String addedField = "addedField";
}

@Patches(OtherClass.class)
class UseAddedFieldInOtherClass extends OtherClass_Mirror {
  @Override
  public String otherMethod(String $1) {
    $1 = ((InsertFieldUseExternally) sampleClass).addedField;
    return super.otherMethod($1);
  }
}
```

3.2.7 Adding new constructors

Adding new constructors to an `extension class` is done similarly to new methods, by just adding a new constructor declaration. However, in the context of a new constructor, the constructor must call an existing constructor of the `original class` by means of `super(args)`. It is currently not possible to call a parent class constructor of the `original class` directly.

Calling a constructor added to `original class` from another class can be done by calling `new ExtensionClassName(args)` directly. This call will be translated to a constructor call on the original class.

3.2.8 Modifying access modifiers

The access modifiers of a field or method could be overridden by adding that field/method to the `extension class`, annotating it with the `@Modify` annotation and listing the changed access modifiers as normally. The field or method for which access modifiers should be changed in the `original class` will be looked up based on the rules outlined above for field and method modification. An example adding synchronized to `SampleClass .instanceMethod` and making `SampleClass.instanceField` public would look like the following:

```java
@Patches(SampleClass.class)
public class UpdateFieldAndMethodModifiers extends
    SampleClass_Mirror {
  @Modify
  public String instanceField;

  @Modify
  synchronized String instanceMethod(String $1) {
    return super.instanceMethod($1);
  }
}
```

3.2.9 Method and Constructor instrumentation

Due to the complexity of implementing a proper API for method and constructor body instrumentation similarly to what is possible with Javassist `CtMethod.instrument(ExprEditor)`, it is out of scope for this thesis and left as an opportunity for future enhancement.

3.3 Compile-time type safety checking

Compile-time syntax and type safety checking relies on the regular Java compiler rules with some additions from the annotation processor to handle `@Modify` annotations.

For every field of the `extension class` annotated with the `@Modify` annotation, there must exist a matching field in the `original class` with the exact same type and name. If such a field does not exist, then the compilation fails and a compilation error is registered for the field declaration in the `extension class`. Similarly, if a field of the `extension class` does not have the `@Modify` annotation, a field with the same signature must not exist in the `original class`. If such a field does exist, the annotation processor registers a compilation error for the field declaration in the `extension class`. This applies to both static and instance fields.

For every method of the `extension class` annotated with the `@Modify` annotation, there must exist a matching method in the `original class` with the same signature – i.e. method name, return type and argument types. If such a method does not exist, the compilation fails and a compilation error is registered for the method declaration in the `extension class`. Similarly, if a method of the `extension class` does not have the `@Modify` annotation, a method with the same signature must not exist in the `original class`. If such a method does exist, the annotation processor registers a compilation error for the method declaration in the `extension class`. This applies to both static and instance methods.

For every constructor of the `extension class` annotated with the `@Modify` annotation, there must exist a matching constructor in the `original class` with the same signature – i.e. argument types. If such a constructor does not exist, the compilation fails and a compilation error is registered for the constructor declaration in the `extension class`. An additional check is made to validate that the constructor declaration in the `extension class` calls `super(args)` with the same argument types. If it calls `this(args)` or `super(args)` with different argument types, a compilation error is registered for the constructor declaration in the `extension class`. Similarly, if a constructor of the `extension class` does not have the `@Modify` annotation, a constructor with the same signature must not exist in the `original class`. If such a constructor does exist, the annotation processor registers a compilation error for the constructor declaration in the `extension class`.

The constructors of the `extension class` must be declared with at least package-private visibility so the constructors of the `original class` could call them. If a constructor of the `extension class` defines private level visibility, an error should be registered for the constructor declaration in the `extension class`.

The additional compile-time checks listed above are implemented in class `ExtensionClassValidator` (Appendix F)

3.4 Wiring code generation

After the `extension class` has been checked for syntax errors and type-safety based on the rules above, some mechanism has to be implemented which actually realizes the extension of the `original class`. When a call is made to a member of the `original class`, which has been modified in the `extension class`, the logic in the `extension class` should be applied based on the rules defined above. One of the ways this could be achieved is by modifying the entire codebase and replacing calls to the `original class` with calls to the `extension class`. This, however, requires scanning through all the classes available in the application and modifying all the classes that are found to use the `original class` and could result in performance degradation and problems with tooling integration (debugger for example). With Javassist however, the call sites to the `original class` are left unchanged and the `original class` methods/constructors/fields themselves are modified. This approach is preferrable since it requires modifying only the `original class` itself.

One approach on how to achieve modifying the `original class` is to merge the logic from the `extension class` directly onto the fields/methods/constructors of the `original class`. Another approach is to transform the `original class` and the `extension class` to collaborate via delegation and create the wiring code required to emulate this. The present implementation uses the second approach by transforming the `extension class` into an `inner class` of the `original class` and modifying the `original class` to call methods/constructors/fields of the `inner class` as required.

3.4.1 Extension class transformation

To transform the `extension class` into an inner class of the `original class`, there are some general transformations that have to be applied to the `extension class`.

First of all, during compile-time, the `extension class` is extending the `mirror class` and all calls to `super` constructors, fields and methods use the `mirror class` as the target type. These targets have to be replaced by the `original class` type instead.

Secondly, the `mirror class` was generated in order to bypass some visibility and finality constraints on the members of the `original class` – so that final and non-visible methods could be overridden. However, these visibility constraints still apply in the `original class` and need special handling in the `extension class` to be accessible. The standard approach how the Java compiler allows an inner class access to private members of the containing class (and vice-versa) is by generating synthetic methods into the containing class with package-private visibility during compilation, and rewiring the calls to private members of the other class to call the synthetic accessor instead. This approach is emulated so that the `inner class` that the `extension class` is transformed into, calls the private constructors, fields and methods of the `original class` via synthetic methods generated to access them.

Lastly, since the `extension class` is transformed into an `inner class` of the `original class`, the `mirror class` that the `extension class` extends from at compile-time is removed from the `extension class`.

3.4.2 Original class transformation

The `original class` also needs to be transformed to support the field, method and constructor modifications specified in the `extension class`.

3.4.3 Extension class lifecycle

The `extension class` is transformed into an inner class of the `original class` and the class and instance lifecycle of the `extension class` is tied to the class and instance lifecycle of the `original class`. When the `original class` is loaded into a classloader, the `extension class` should also be loaded into the same classloader. When an instance of the `original class` is created by calling a constructor on it, an instance of the `extension class` will also be created and a reference to it saved into a field named `__extension` of the `original class`. During instantiation, the instance of the `extension class` will hold a soft reference to the `original class` in a field named `__original` so the `extension class` could also access the `original class` instance.

3.4.4 Field modification

For all fields that the `extension class` overrides via the `@Modify` annotation, the modifiers of the field of the `original class` with the same signature are updated to match. An exception to this rule is that if the field declaration in the `extension class` declares the field to be final, then this is ignored. The field declarations are removed from the `extension class` and all constructors of the `extension class` are updated to modify the field in the `original class` if the `extension class` modifies the field value. Furthermore, other members of the `extension class` using an overridden field are rewired to access the field in the `original class`. For private fields of the `original class`, a synthetic accessor is generated as described above.

3.4.5 Method modification

For all methods that the `extension class` overrides via the `@Override` or `@Modify` annotation, a copy is created in the `original class`. The original method is then rewired to invoke the override method inside the `extension class` via `__extension.methodName(args)`, which is itself rewired to call the copy of the original method via `__original.methodName(args)`, when it needs to call the original method on `super`.

The method delegation approach is illustrated with the diagram in the Figure 2.

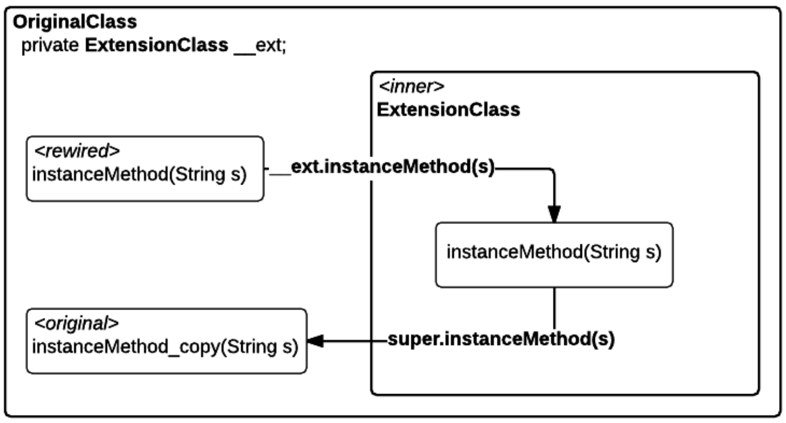


Figure 2: Method rewiring between the original class and the extension class.

3.4.6 Constructor modification

For all constructors that the `extension class` overrides via the `@Modify` annotation, the modifiers of those constructors are updated to the modifiers of the constructor declaration in the `extension class`. In addition, if a constructor of the `original class` is called which has a matching constructor in the `extension class`, then this constructor must be the one constructing the instance of the `extension class` and saving that instance into the `__extension` field of the `original class`. To implement this an int field of `__constrCounter` is added to the `original class`. The constructors of the `original class` are updated to increment the counter before the constructor calls `this(args)` or `super(args)` to delegate to another constructor and to decrement the counter after the constructor finishes and returns. If during this decrement, the constructor counter reaches zero, then the constructor that is finishing has to be the last constructor of the `original class` before returning the instance to the call site which called `new` on the class. In this case, this constructor needs to create an instance of the `extension class` – either by calling the matching `@Modify` annotated constructor, or the default no-args constructor – and save it into the `__extension` variable before returning. The constructors in the `extension class` that include the `@Modify` annotation

3.4.7 Added interfaces

If the `extension class` implements any interfaces, that the original class does not, these interfaces are added to the `original class` and all methods defined in those interface are created inside the `original class`. These methods act as proxies to the `extension class` and call the implementation method inside the `extension class` directly.

3.4.8 Added superclasses

The `extension class` cannot define an extension from a class since it is already extending from the `mirror class` of the `original class`. Modifying the class hierarchy of the `original class` with the `extension class` is out of scope for this thesis.

3.4.9 Added fields

The fields that are added inside the `extension class` do not need special handling, these can be accessed directly by any methods/field declarations in the `extension class` itself. If an `extension class` for some other class wants to use fields added to some `original class`, and uses it by casting the `original class` to an instance of the `extension class`, the `extension class` of that other class is rewired to access the field via the `__extension` field of the `original class`.

3.4.10 Added methods

The methods that are added inside the `extension class`, and do not implement an added interface as described above, do not need special handling, these can be accessed directly by any other added or overridden methods inside the same `extension class` itself. If an `extension class` for some other class wants to use methods added to some `original class`, and uses it by casting the `original class` to an instance of the `extension class`, the `extension class` of that other class is rewired to access the method via the `__extension` field of the `original class`.

3.4.11 Added constructors

For all constructors that are added inside the `extension class` and that do not have the `@Modify` annotation – except for the default no-args constructor – a matching constructor is created inside the `original class`. The added constructor in the `original class` is wired to call an existing constructor of the `original class` via `this(args)` that matches the `super(args)` of the constructor declaration in the `extension class`. Creating an instance of the `extension class` is done based on the rules described above in the constructor modification section. If an `extension class` for some other class wants to use a constructor added to some `original class`, and uses it by casting the `original class` to an instance of the `extension class`, the `extension class` of that other class is rewired to invoke the added `constructor` on the `original class` directly.

3.4.12 Transformation implementation

The above transformation logic is achieved by generating a wiring class that uses Javassist to transform the `original class` and the `extension class` during load-time. The wiring class is annotated with `org.zeroturnaround.javassist.annotation.Transformer` (Appendix C) with the value being the fully qualified name of the class that the transformer is capable of transforming, and implements a single interface – `org.zeroturnaround.javassist.annotation.processor.wiring.JavassistClassBytecodeProcessor`. This interface declares a single method – `public void process(ClassPool cp, ClassLoader cl, CtClass ctClass) throws Exception` – which includes the Javassist statements to modify both the `original class` and the `extension class`. Applications using this implementation to modify some class should look up the `@Transformer` classes capable of modifying the class bytecode, and call the `process()` method with the `javassist.CtClass` of the class and the `javassist.ClassPool` and `java.lang.ClassLoader` instances.

The rewiring logic described above is implemented in the class `WiringClassGenerator` (Appendix G). The `WiringClassGenerator` uses the Apache Velocity [17] templating engine for generating the transformer class from template `cbp.vtl` (Appendix H).

3.5 Runtime lookup and execution of the wiring code

The result of the rewiring step is a class file implementing the transformation logic of the `original class` with Javassist. To apply the transformation to some `original class`, an application needs to look up the class containing the transformation logic, instantiate it, and call the `process()` method. Implementations for runtime lookup of the transformation classes can vary based on the requirements of application developers and the applications they write. As an example, the simplest implementation could use classpath scanning to iterate through the class files found on the classpath of the application classloader, check if they implement the `@Transformer` annotation for some class, instantiate those and call the transformer with a `CtClass` instance of that class. Specific implementations of any lookup mechanisms are not in the scope of this thesis.

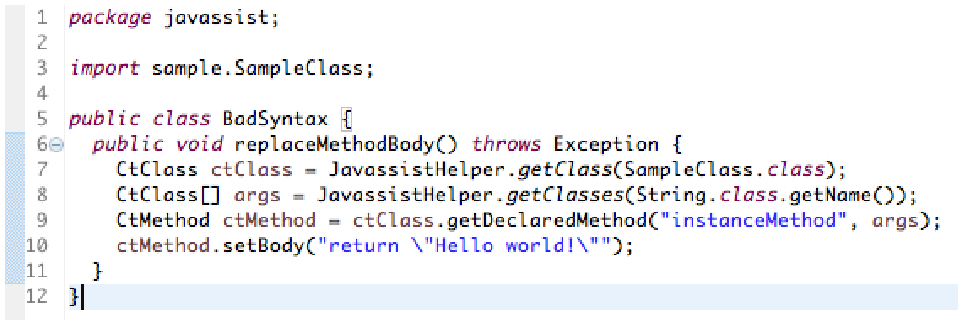
4 Analysis

It is difficult to quantitatively analyse what kind of benefit the approach presented in this thesis would bring – to a developer or a team of developers – over using Javassist directly for class bytecode modification. It could be argued that the developers using the outlined implementation should make less mistakes when writing class bytecode modification logic and should be able to more easily reason about `extension class` behavior, which in turn should lead to more maintainable code. The effects of this, however, varies – depending on the custom tooling already in place to alleviate some of the limitations with Javassist, and also on how experienced a developer is and how many mistakes he makes.

In the absence of quantitative analysis, this section presents some insights on how a developer working on a bytecode modification class would experience the development in his IDE, as well as some of the benefits and trade-offs of using the type-safe bytecode modification library while comparing to Javassist.

The use-case presented to demonstrate the comparison between bytecode modification, by a developer working directly with Javassist and by a developer working with the type-safe bytecode modification library implementation, is for a scenario where the developer needs to replace the method `SampleClass.instanceMethod(String input)` to return "Hello world!" instead of the current implementation.

To demonstrate the problem, the developer will make two errors in the codebase – a syntax error by simply omitting the semicolon in the return statement of the method, and a type error by using `(nstanceMethod` as the method name instead of `instanceMethod`. The developer is using Eclipse IDE with "Build Automatically" enabled.



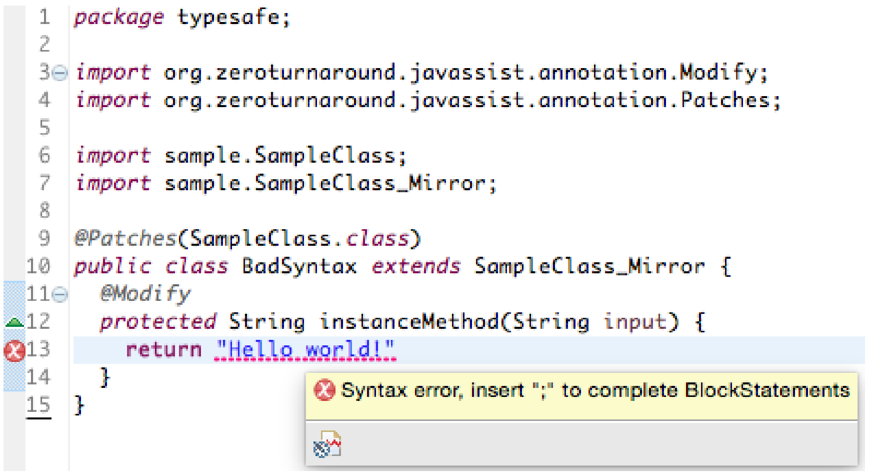
```
 1 package javassist;
 2 
 3 import sample.SampleClass;
 4 
 5 public class BadSyntax {
 6   public void replaceMethodBody() throws Exception {
 7     CtClass ctClass = JavassistHelper.getClass(SampleClass.class);
 8     CtClass[] args = JavassistHelper.getClasses(String.class.getName());
 9     CtMethod ctMethod = ctClass.getDeclaredMethod("instanceMethod", args);
10     ctMethod.setBody("return \"Hello world!\"");
11   }
12 }
```

Figure 3: A developer using Javassist for Java bytecode modification. A syntax error introduced due to missing semicolon on line 10.

Figure 3 demonstrates a developer creating an error in his bytecode modification code by omitting the semicolon in the argument to `CtMethod.setBody(String)`. As the compiler only checks that the argument to the

`CtMethod.setBody(String)` call is of type string without any further analysis on the content of that string, no errors are reported.

However, a developer using the type-safe Java bytecode modification library in a similar scenario on Figure 4 would get an appropriate compiler error informing the developer of the missing semicolon.

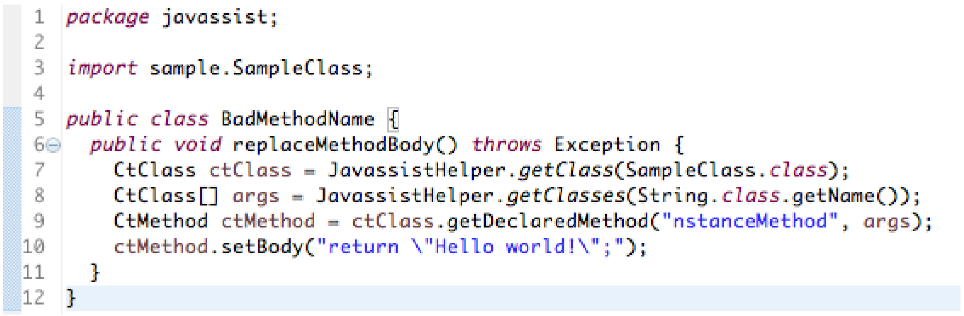


```java
1  package typesafe;
2
3  import org.zeroturnaround.javassist.annotation.Modify;
4  import org.zeroturnaround.javassist.annotation.Patches;
5
6  import sample.SampleClass;
7  import sample.SampleClass_Mirror;
8
9  @Patches(SampleClass.class)
10 public class BadSyntax extends SampleClass_Mirror {
11   @Modify
12   protected String instanceMethod(String input) {
13     return "Hello world!"
14   }
15 }
```

Syntax error, insert ";" to complete BlockStatements

Figure 4: A developer using the type-safe library for Java bytecode modification. A type error introduced due to missing semicolon on line 13.

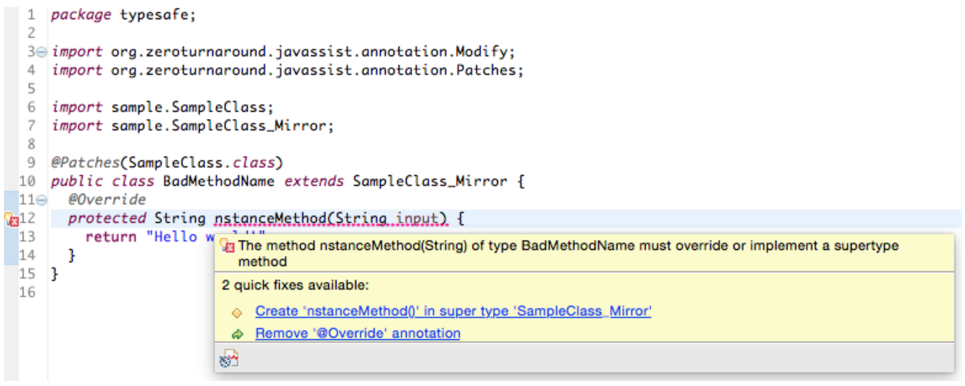
Figure 5 demonstrates a developer creating an error in his bytecode modification code by mistyping the method name for `SampleClass.instanceMethod(String input)`. Similarly to the above, since the compiler only checks that the supplied method name is of type string, no errors are reported.



```java
1  package javassist;
2
3  import sample.SampleClass;
4
5  public class BadMethodName {
6    public void replaceMethodBody() throws Exception {
7      CtClass ctClass = JavassistHelper.getClass(SampleClass.class);
8      CtClass[] args = JavassistHelper.getClasses(String.class.getName());
9      CtMethod ctMethod = ctClass.getDeclaredMethod("nstanceMethod", args);
10     ctMethod.setBody("return \"Hello world!\";");
11   }
12 }
```

Figure 5: A developer using Javassist for Java bytecode modification. A type error introduced due to mistyped method name on line 9.

However, a developer using the type-safe Java bytecode modification library in a similar scenario on Figure 6 would get an appropriate compiler error informing the developer that the method name has been mistyped.



```
 1 package typesafe;
 2
 3 import org.zeroturnaround.javassist.annotation.Modify;
 4 import org.zeroturnaround.javassist.annotation.Patches;
 5
 6 import sample.SampleClass;
 7 import sample.SampleClass_Mirror;
 8
 9 @Patches(SampleClass.class)
10 public class BadMethodName extends SampleClass_Mirror {
11   @Override
12   protected String nstanceMethod(String input) {
13     return "Hello w
14   }
15 }
16
```

The method nstanceMethod(String) of type BadMethodName must override or implement a supertype method

2 quick fixes available:

- Create 'nstanceMethod()' in super type 'SampleClass_Mirror'
- Remove '@Override' annotation

Figure 6: A developer using the type-safe library for Java bytecode modification. A type error introduced due to mistyped method name on line 12.

4.1 Benefits

The main benefit of using the type-safe Java bytecode modification library is compile-time type-safety. A developer writing bytecode modification logic for existing classes could detect syntax and type errors early in development and could fix those right away instead of either building custom tooling to detect those, or only finding out about them during runtime.

Second, with this approach, the IDE could provide code completion capabilities for calling existing fields and methods of the `original class` and for overriding existing methods. Java developers have come to expect such support from their IDEs and rely on the increased productivity and type-safety this provides.

Third, the proposed API uses subclassing which is a familiar construct for overriding some functionality of the `original class` and should be easier to learn for developers who are unfamiliar with Java bytecode modification libraries, as well as easier to reason about for seasoned developers who are already familiar with the concepts.

4.2 Trade-offs

The type-safe Java bytecode modification library has some trade-offs/limitations that developers should consider before integrating it to their toolchains.

First, the class under modification has to be on the classpath of the application during compile-time. For applications that need to change classes for which either the source code of the class or the class file itself is available, this does not pose a problem. However, if the class file is not available/obtainable then the type-safety of the bytecode modification cannot be enforced. For example, a performance tool which modifies all method invocations of an application (regardless of the method signatures or the classes they are defined in) to calculate and save the execution times for subsequent performance analysis. This kind of tool would not be able to use the type-safe Java bytecode modification library due to the unlikelyhood of acquiring all user classes that would need to be instrumented, or furthermore, all the classes for all tool users.

Second, it is not possible to perform bulk updates of methods/constructors/fields of a given class (or a group of classes) with a single statement. With Javassist, the `CtClass` object could be asked for a list of methods with the same method name, and the Java bytecode modification logic could be added inside an iterator. However with the approach outlined in this thesis this would be achieved by requiring `all` of the methods, that require the same modification, to be overridden in the `extension class`.

Third, the current implementation of the type-safe Java bytecode modification library does not support multiple versions of the same class under modification. With Javassist it is possible to differentiate between runtime versions by querying for the existence of some class or some method, and if the class or method exists, then to alter the bytecode modification logic to fit the different versions.

Fourth, due to the way the type-safe Java bytecode modification library is implemented – creating an inner class to the `original class` and delegating method execution to the `inner class` instead of modifying the method/field/constructor definitions directly – this incurs a slight performance penalty and a memory/permgen consumption overhead since the `extension classes` needs to be loaded into some classloader and instantiating the `original class` also creates an instance of the `extension class`.

4.3 Summary

The focus of this work is on general bytecode modification use-cases that can be expressed in terms of subclassing, with some additional support for common scenarios that cannot be expressed via a plain Java subclass. The supported use-cases and the application of those in combination with the usage of Javassist API is expected to cover most of the possible scenarios for Java bytecode modification. The present implementation does not aim to provide a complete alternative for Javassist, but instead focuses on the most important areas where it is possible to guarantee the type-safety of instrumentation.

5 Related work

Java class bytecode modification has been around for a long time. Due to maturity of the topic, the available tools implementing this are quite numerours. This section attempts to outline the major libraries providing Java bytecode modification, give a short overview of how each of those work and compare these to the implementation outlined in this thesis.

5.1 ObjectWeb ASM

ASM [18] is a Java class manipulation tool designed to dynamically generate and manipulate Java classes, which are useful techniques to implement adaptable systems. ASM uses the "visitor" design pattern without explicitly representing the visited class structure as an object tree.

An example ASM transformer class for modifying the bytecode of `SampleClass.instanceMethod(String string)` to print "Hello world!" instead would look like the following:

```java
package asm;

import java.io.ByteArrayInputStream;

import javassist.ClassPool;

import org.junit.Assert;
import org.junit.Test;
import org.objectweb.asm.ClassReader;
import org.objectweb.asm.ClassVisitor;
import org.objectweb.asm.ClassWriter;
import org.objectweb.asm.MethodVisitor;
import org.objectweb.asm.Opcodes;

import sample.SampleClass;

public class ASMTransformer {
  @Test
  public void replaceMethodBody() throws Exception {
    ClassWriter writer = new ClassWriter(ClassWriter.COMPUTE_MAXS);
    ClassVisitor visitor = new ClassVisitor(Opcodes.ASM4, writer) {
      @Override
      public MethodVisitor visitMethod(int access, String name,
          String desc, String signature, String[] exceptions) {
        MethodVisitor mv = super.visitMethod(access, name, desc,
            signature, exceptions);
        if (name.equals("instanceMethod") &&
            desc.equals("(Ljava/lang/String;)Ljava/lang/String;")) {
          mv.visitCode();
          mv.visitLdcInsn("Hello world!");
          mv.visitInsn(Opcodes.ARETURN);
          mv.visitMaxs(2, 2);
          mv.visitEnd();
          return null;
        }

        return mv;
      }
    };
    ClassReader reader = new ClassReader(
      this.getClass().getClassLoader()
      .getResourceAsStream("sample/SampleClass.class"));
    reader.accept(visitor, 0);
    final byte[] modifiedBytecode = writer.toByteArray();

    SampleClass sampleClass = (SampleClass)
        toClass(modifiedBytecode);
    Assert.assertEquals("Hello world!",
        sampleClass.publicMethod("Test"));
  }

  private Object toClass(byte[] bytecode) throws Exception {
    ClassPool cp = new ClassPool();
    return cp.makeClass(new
        ByteArrayInputStream(bytecode)).toClass().newInstance();
  }
}
```

Due to the light weight of the library and its extensive capabilities, it is used in many different tools ranging from programming languages to database persistence frameworks. Notable tools/frameworks using ObjectWeb ASM include Groovy, Clojure, JRuby, AspectJ, Apache OpenEJB and Oracle TopLink.

ASM is a versatile tool, but its low-level API requires an extensive knowledge of Java bytecode and may not be suitable for unexperienced programmers.

5.2 CGLib

CGLib [5] is a Java class bytecode manipulation tool built on top of ObjectWeb ASM which uses different interceptors to define the bytecode modification logic. It creates a proxy class for the original class during runtime and rewires all constructor invocations and member accesses through this generated proxy class

An example CGLib transformer class for modifying the bytecode of `SampleClass.instanceMethod(String string)` to print "Hello world!" instead would look like the following:

```java
package cglib;

import java.lang.reflect.Method;

import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

import org.junit.Assert;
import org.junit.Test;

import sample.SampleClass;

public class CGLibTransformer {
  @Test
  public void testReplaceHelloWorld() throws Exception {
    Enhancer enhancer = new Enhancer();
    enhancer.setSuperclass(SampleClass.class);
    enhancer.setCallback(new MethodInterceptor() {
      @Override
      public Object intercept(Object target, Method method, Object[]
          args, MethodProxy proxy) throws Throwable {
        if ("publicMethod".equals(method.getName())) {
          return "Hello world!";
        } else {
          return proxy.invokeSuper(target, args);
        }
      }
    });

    SampleClass proxy = (SampleClass) enhancer.create();
    Assert.assertEquals("Hello world!", proxy.publicMethod("Test"));
  }
}
```

Notable tools/frameworks using CGLib include Spring, Hibernate and Guice.

5.3 AspectJ

AspectJ [19] is an aspect-oriented programming extension to Java for implementing and modularizing cross-cutting concerns such as logging, security or transaction demarcation. It uses a notion of pointcuts – declarative way of defining the program elements that are to be modified – and advice – the method/block of code that describes the modification logic to be applied to a subset of pointcuts. The pointcut declaration syntax uses xml-based configuration or annotations such as `@Aspect`, `@Around`, `@Pointcut`

An example AspectJ transformer class for modifying the bytecode of `SampleClass.publicMethod(String string)` to print “Hello world!” instead would look like the following:

```java
package aspectj;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class AspectJTransformer {
  @Around("execution(* sample.SampleClass.publicMethod(..))")
  public String publicMethod(final ProceedingJoinPoint pjp) {
    return "Hello world!";
  }
}
```

AspectJ is somewhat similar to the development model outlined in this thesis since the code contained in an aspect and its corresponding advice is structured as a Java class. Furthermore, if the libraries containing classes that are under modification are on the compiler classpath, the aspect code itself can be type-safe and report errors during compilation.

However, its reliance on stringly-typed syntax for declaring pointcuts, overreliance on different annotations for defining different types of advice and the advice method signatures requiring AspectJ-specific arguments lists for more complex tasks – such as looking up original method argument values – keep AspectJ from being as type-safe as the approach outlined in this thesis.

Notable tools/frameworks using AspectJ include Spring.

5.4 Apache BCEL

The Byte Code Engineering Library (Apache Commons BCEL$^{\text{TM}}$) [4] is one of the oldest Java bytecode modification libraries and with the advance of newer tools like ASM and Javassist, is not used as much as it used to. Notable tools/frameworks using Apache BCEL include JBoss, FindBugs, Xalan XSLT Compiler and AspectJ.

6 Conclusion

This thesis investigated the limitations of Java bytecode modification using Javassist and proposed an alternative solution which alleviates some of these limitations – compile-time type-safety of the modification code and IDE support for code completion.

First, the thesis presented Java bytecode modification use-cases and demonstrated the limitations developers face while using Javassist. As a prerequisite to the proposed solution, an overview of type systems and the advantages of static typing over dynamic typing was provided. A brief introduction to Java annotation processing, essential for the proposed implementation of the proof of concept, was also given.

Second, the application programming interface and the annotations introduced to support the type-safe bytecode modification were detailed in the implementation part of the thesis. The described approach relied on writing the bytecode modification code as a direct subclass of the original class and depended on the compiler taking care of syntax checking and type validation of the subclass. Since in some cases it was impossible to depend from the original class directly – if the original class was final, for example – a concept of *mirror class* was introduced. A mirror class is generated based on the original class during compilation by the Java annotation processor and has the same signature as the original class, with visibility modifiers of its members set to protected or higher and "final" keywords removed. The thesis described the additional validation checks introduced to handle overriding static methods, constructors and fields in a type-safe way. Finally, the mechanism through which the subclass modification is applied to the original class during runtime was described.

Next, the development experience for a developer using Javassist for Java bytecode modification was compared to the type-safe approach. The analysis presented benefits and drawbacks a developer interested in adding the type-safe Java bytecode modification library to his toolset should consider.

Finally, the thesis gave an overview of other major Java bytecode modification libraries and compared them briefly with the proposed type-safe approach.

In conclusion, Java bytecode modification is a non-trivial task and a number of tools already exist to solve this task, yet without any type-safety guarantees. This thesis described an approach to solve this problem by utilizing Java annotation processing and Javassist to provide a working alternative to those tools providing the type-safety.

The most recent source code for the proof of concept can be found online at: `https://bitbucket.org/svenlaanela/thesis-prototype/src`

6.1 Future work

Based on some of the remaining limitations for the type-safe Java bytecode modification library which were outlined in the trade-offs section (4.2), here follows a list of proposed future improvements that would alleviate some of the limitations of the implementation and expand upon the work to provide even more value to developers using the library.

First of all, to smoothen the transition for existing developers working with Javassist to the type-safe Java bytecode manipulation library, the implementation could include a capability for including custom Javassist code directly inside the body of the `extension class`. A developer could then gradually transition an existing bytecode modification code-base written in Javassist and would not have to rewrite everything at once. Furthermore, since Javassist supports more exhaustive bytecode modification use-cases, this would avoid excluding the type-safe approach in those scenarios.

Second, this thesis left the instrument API (`using CtMethod.instrument()`) as an opportunity for future improvement. This use-case is often used by developers working with Javassist and would also benefit from a type-safe alternative.

Third, multiple versions of the same class or library cannot currently be modified with the type-safe Java bytecode modification library. A tool which functionality depends on modifying an application class usually has to support multiple versions of that application and multiple versions of the class. The main question here is whether it is possible to provide compile-time type safety by including multiple versions of the same `original class` in the compiler classpath. Alternatively, it may require custom IDE tooling to have two different `extension classes` depend – and provide code completion and type-safety for – different versions of a single `original class`.

Fourth, when compared to Javassist, the type-safe bytecode modification library provides a more structured view of the extension code. The current implementation depends on an `extension class` extending from a `mirror class` of the original class, however it would be preferable to view the `extension class` as extending from the `original class` directly. This would be useful for seeing a unified view of how the `extension class` actually fits into the original class code and with some custom IDE tooling could allow the user to click through the method invocations between the `extension class` and the `original class`.

Finally, currently none of the bytecode modification libraries (except for AspectJ) have debugger integration that would provide a unified debugging experience for developers working on bytecode modification code for existing classes. When stepping through a `original class` which is modified by some bytecode modification library, the debugging view does not account for the bytecode modification logic and does not allow to "stepping" into the `extension class` or the bytecode modification statements. It should be possible to build a custom IDE debugger integration on top of the unified view of the `original class` and its `extension class` which would behave as if the extension class actually exists inside the original codebase.

References

[1] "Introduction to programming using java, seventh edition, version 7.0, august 2014 by david j. eck section 1.3 the java virtual machine." `http://math.hws.edu/javanotes/c1/s3.html`.

[2] "Objectweb asm." `http://asm.ow2.org/`. [Online; accessed 13-May-2015].

[3] S. Chiba, "Javassist—a reflection-based programming wizard for java," in *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, p. 174, 1998.

[4] "Bcel, the byte code engineering library." `http://commons.apache.org/bcel/`. [Online; accessed 13-May-2015].

[5] "cglib." `https://github.com/cglib/cglib`. [Online; accessed].

[6] F. Yellin and T. Lindholm, "The java virtual machine specification," *Addison-Wesley*, 1996.

[7] B. C. Pierce, *Types and Programming Languages.* Cambridge, MA, USA: MIT Press, 2002.

[8] "Javassist tutorial 2." `http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/tutorial/tutorial2.html`. [Online; accessed 29-March-2015].

[9] "Javassist api: javassist.expr.expreditor." `http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/html/javassist/expr/ExprEditor.html`. [Online; accessed 2-April-2015].

[10] "Processor (java platform se 8)." `http://docs.oracle.com/javase/8/docs/api/javax/annotation/processing/Processor.html`. [Online; accessed 20-March-2015].

[11] "Compilation overview." `http://openjdk.java.net/groups/compiler/doc/compilation-overview/`. [Online; accessed 23-May-2015].

[12] "Filer (java platform se 8)." `http://docs.oracle.com/javase/8/docs/api/javax/annotation/processing/Filer.html`. [Online; accessed 23-May-2015].

[13] "Messager (java platform se 8)." `http://docs.oracle.com/javase/8/docs/api/javax/annotation/processing/Messager.html`. [Online; accessed 23-May-2015].

[14] "Elements (java platform se 8)." `http://docs.oracle.com/javase/8/docs/api/javax/lang/model/util/Elements.html`. [Online; accessed 23-May-2015].

[15] "Types (java platform se 8)." `http://docs.oracle.com/javase/8/docs/api/javax/lang/model/util/Types.html`. [Online; accessed 23-May-2015].

[16] K. Arnold, D. Holmes, T. Lindholm, F. Yellin, *et al.*, "Java language specification," 2000.

[17] "Apache velocity." `http://velocity.apache.org/`. [Online; accessed 24-May-2015].

[18] E. Bruneton, R. Lenglet, and T. Coupaye, "Asm: a code manipulation tool to implement adaptable systems," *Adaptable and extensible component systems*, vol. 30, 2002.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *ECOOP 2001—Object-Oriented Programming*, pp. 327–354, Springer, 2001.

Appendix A   Patches.java

```
package org.zeroturnaround.javassist.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

/**
 * Defines the class to patch
 */
@Target(value={ElementType.TYPE})
public @interface Patches {
  public Class<?> value();
}
```

Appendix B Modify.java

```java
package org.zeroturnaround.javassist.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * An annotation marking a field, method or constructor of an extension class
 * as replacing some field, method or constructor of the original.
 */
@Retention(RetentionPolicy.SOURCE)
@Target(value={ElementType.FIELD, ElementType.METHOD, ElementType.CONSTRUCTOR})
public @interface Modify {}
```

Appendix C   Transformer.java

```
package org.zeroturnaround.javassist.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

/**
 * An annotation marking a class bytecode transformer class for a given class
 * specified by the String value. After the annotation processor has constructed
 * a Transformer class for a given original class and its extension class, the
 * Transformer class is marked with the @Transformer annotation.
 */
@Target(value={ElementType.TYPE})
public @interface Transformer {
  String value();
}
```

Appendix D TypesafeBytecodeModificationProcessor.java

```java
package org.zeroturnaround.javassist.annotation.processor;

import java.io.BufferedWriter;
import java.io.PrintWriter;
import java.util.Set;

import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.ElementKind;
import javax.lang.model.element.TypeElement;
import javax.lang.model.type.MirroredTypeException;
import javax.lang.model.type.TypeMirror;
import javax.tools.JavaFileObject;

import org.zeroturnaround.javassist.annotation.Patches;
import org.zeroturnaround.javassist.annotation.processor.mirror.MirrorClassGenerator;
import org.zeroturnaround.javassist.annotation.processor.util.IOUtil;
import org.zeroturnaround.javassist.annotation.processor.validator.ExtensionClassValidator;
import org.zeroturnaround.javassist.annotation.processor.wiring.WiringClass;

@SupportedAnnotationTypes("org.zeroturnaround.javassist.annotation.Patches")
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class TypesafeBytecodeModificationProcessor extends AbstractProcessor {

  @Override
  public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
    try {
      for (Element element : roundEnv.getRootElements()) {
        if (element.getKind() == ElementKind.CLASS && element.getAnnotation(Patches.class) != null) {
          doProcess((TypeElement) element);
        }
      }
    }
    catch (RuntimeException e) {
      System.err.println("Generic error running annotation processor " + e);
    }
    return true;
  }

  private void doProcess(TypeElement extensionClass) {
    TypeMirror originalClassType = getPatchedClassType(extensionClass);
    generateMirrorClass(extensionClass, originalClassType);
    validateExtensionClass(extensionClass, originalClassType);
    generateWiringClass(extensionClass, originalClassType);
  }

  private TypeMirror getPatchedClassType(Element extensionClass) {
    Patches annotation = extensionClass.getAnnotation(Patches.class);
    try {
      Class<?> value = annotation.value();
      return null; // should throw exception instead of returning here;
    }
    catch (MirroredTypeException e) {
      return e.getTypeMirror();
    }
  }

  private void generateMirrorClass(Element extensionClass, TypeMirror originalClass) {
    System.out.println("Generating mirror class for " + extensionClass);
    PrintWriter w = null;
    try {
      MirrorClassGenerator mirrorClass = new MirrorClassGenerator(originalClass.toString());

      JavaFileObject mirrorClassObject = processingEnv.getFiler().createSourceFile(mirrorClass.getName(),
            extensionClass);

      w = new PrintWriter(new BufferedWriter(mirrorClassObject.openWriter()));
      w.print(mirrorClass.generateSource());
      w.flush();
    } catch (Exception e) { // TODO: proper error handling
      System.out.println("Failure generating mirror class" + e);
    } finally {
      IOUtil.closeQuietly(w);
    }
  }

  private void validateExtensionClass(TypeElement extensionClass, TypeMirror originalClassType) {
    new ExtensionClassValidator(extensionClass, originalClassType, processingEnv.getMessager()).validate();
  }

  private void generateWiringClass(Element extensionClass, TypeMirror originalClass) {
    System.out.println("Generating wiring class for " + originalClass + " (" + extensionClass + ")");

    PrintWriter w = null;
    try {
      WiringClass wiringClass = new WiringClass(originalClass.toString(), extensionClass.toString());

      JavaFileObject cbpClass = processingEnv.getFiler().createSourceFile(wiringClass.getName(), extensionClass);

      w = new PrintWriter(new BufferedWriter(cbpClass.openWriter()));
      w.print(wiringClass.generateSource());
      w.flush();
    } catch (Exception e) {
      System.out.println("Failure generating mirror class" + e);
      e.printStackTrace();
    } finally {
      IOUtil.closeQuietly(w);
    }
  }
}
```

Appendix E MirrorClassGenerator.java

```java
package org.zeroturnaround.javassist.annotation.processor.mirror;

import java.util.Arrays;

import javassist.CannotCompileException;
import javassist.ClassClassPath;
import javassist.ClassPool;
import javassist.CtClass;
import javassist.CtConstructor;
import javassist.CtField;
import javassist.CtMethod;
import javassist.Modifier;
import javassist.NotFoundException;
import javassist.bytecode.AccessFlag;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.zeroturnaround.javassist.annotation.MethodCall;

/**
 * MirrorClass encapsulates the logic of generating a mirror class (and required hierarchy) for a given original
       class.
 *
 */
public class MirrorClassGenerator {
  private static final Logger logger = LoggerFactory.getLogger(MirrorClassGenerator.class);
  private final String originalClassName;

  public MirrorClassGenerator(String originalClassName) {
    this.originalClassName = originalClassName;
  }

  public String getOriginalClassName() {
    return originalClassName;
  }

  public String getName() {
    // TODO: Mirror classes could use different suffixes?
    return originalClassName + "_Mirror";
  }

  public String getSimpleName() {
    return getName().substring(getName().lastIndexOf('.'));
  }

  public String generateSource() throws Exception {
    return generateSource(originalClassName);
  }

  private String generateSource(String originalClassName) throws Exception {
    ClassPool classPool = ClassPool.getDefault();
    classPool.insertClassPath(new ClassClassPath(this.getClass()));

    CtClass originalClass = classPool.get(originalClassName);
    return generateSource(originalClass);
  }

  private String generateSource(CtClass originalClass) throws Exception {
    StringBuilder result = new StringBuilder();
    result.append("package " + originalClass.getPackageName() + ";\n");
    result.append("\n");
    result.append(generateBodySource(originalClass));
    return result.toString();
  }

  private String generateBodySource(CtClass ctClass) throws Exception {
    StringBuilder result = new StringBuilder();

    String mirrorClassName = ctClass.getSimpleName() + "_Mirror";
    if (mirrorClassName.contains("$")) {
      mirrorClassName = mirrorClassName.substring(mirrorClassName.lastIndexOf('$') + 1);
    }

    // add class declaration
    {
      int modifiers = ctClass.getModifiers();
      if (Modifier.isPrivate(modifiers) || Modifier.isPackage(modifiers)) {
        modifiers = Modifier.setProtected(modifiers);
      }
      modifiers = Modifier.clear(modifiers, Modifier.FINAL);
  if (Modifier.isAbstract(modifiers)) {
    modifiers = Modifier.clear(modifiers, Modifier.ABSTRACT);
  }

  if (!Modifier.isInterface(modifiers)) {
    result.append(Modifier.toString(modifiers) + " class " + mirrorClassName); // public class className
  } else {
    result.append(Modifier.toString(modifiers) + " " + mirrorClassName);
  }
  CtClass superClass = ctClass.getSuperclass();
  if (superClass != null && !"java.lang.Object".equals(superClass.getName())) {
    String superClassName = superClass.getName().replace("$", ".");
    result.append(" extends " + superClassName); // extends parentClassName
  }
  CtClass[] interfaceClasses = ctClass.getInterfaces();
  if (interfaceClasses.length > 0) {
    result.append(" implements ");
    for (CtClass interfaceClass : interfaceClasses) {
      result.append(interfaceClass.getName().replace("$", "."));
    }
  }

  result.append(" {\n");

  if (!Modifier.isInterface(modifiers)) {
    result.append(" protected final void instrument(" + MethodCall.class.getName() + " call) {}\n");
  }
}

// add nested classes
for (CtClass nestedClass : ctClass.getDeclaredClasses()) {
  String innerClassName = nestedClass.getName().substring(ctClass.getName().length() + 1);
  logger.debug("Nested class: " + innerClassName);
  try {
    Integer.parseInt(innerClassName); // anonymous inner class
  }
  catch (NumberFormatException e) {
    String nestedClassSrc = generateBodySource(nestedClass);
    result.append(nestedClassSrc);
  }
}

// add constructors, convert to public for access/override
for (CtConstructor constructor : ctClass.getDeclaredConstructors()) {
  logger.debug("Adding constructor for " + mirrorClassName + " " + constructor.getModifiers() + " " +
         Modifier.toString(constructor.getModifiers()));
  logger.debug("Checking for synthetic");
  if ((constructor.getModifiers() & AccessFlag.SYNTHETIC) != 0 || constructor.getModifiers() == 0) { // wtf
    logger.debug("is synthetic");
    continue;
  }
  String s = "public " + mirrorClassName + "(\n";
  for (int i = 0; i < constructor.getParameterTypes().length; i++) {
    int modifiers = constructor.getModifiers();
    if (Modifier.isPrivate(modifiers) || Modifier.isPackage(modifiers)) {
      modifiers = Modifier.setPublic(modifiers);
    }

    CtClass parameter = constructor.getParameterTypes()[i];
    if (i != 0)
      s += ", ";
    String parameterName = toMirrorSafeName(ctClass, parameter);
    s += parameterName + " $" + (i + 1);
  }
  s += ") {";

  // find eligible super constructor
  for (CtConstructor superConstructor : ctClass.getSuperclass().getDeclaredConstructors()) {
    if (Modifier.isPublic(superConstructor.getModifiers())
        || Modifier.isProtected(superConstructor.getModifiers())
        || Modifier.isPackage(superConstructor.getModifiers())
           && ctClass.getSuperclass().getPackageName().equals(ctClass.getPackageName())) {
      s += " super(";
      for (int i = 0; i < superConstructor.getParameterTypes().length; i++) {
        if (i != 0)
          s += ", ";
        s += "null";
      }
      s += ");";
      break;
    }
  }

  s += "}\n";
```

```
    result.append(s);
  }

  // convert all class fields to public for access
  for (CtField field : ctClass.getDeclaredFields()) {
    int modifiers = field.getModifiers();
    if (Modifier.isPrivate(modifiers) || Modifier.isPackage(modifiers)) {
      modifiers = Modifier.setProtected(modifiers);
    }
    modifiers = Modifier.clear(modifiers, Modifier.FINAL);

    String typeName = toMirrorSafeName(ctClass, field.getType());

    result.append(Modifier.toString(modifiers) + " " + typeName + " " + field.getName() + " = " +
           getDefaultValue(field.getType()) + ";\n");
  }

  // convert all methods to public non-final for access/override
  for (CtMethod method : ctClass.getDeclaredMethods()) {
    logger.debug("Adding method for " + mirrorClassName + " " + method.getModifiers() + " " +
           Modifier.toString(method.getModifiers()));
    logger.debug("Checking for synthetic");
    if ((method.getModifiers() & AccessFlag.SYNTHETIC) != 0 || method.getName().startsWith("access$")) { // wtf
      logger.debug("is synthetic");
      continue;
    }
    String methodSrc = addMethod(ctClass, mirrorClassName, method);
    result.append(methodSrc);
  }

  result.append("}\n");
  return result.toString();
}


private String addMethod(final CtClass ctClass, final String mirrorClassName, final CtMethod method) throws
      NotFoundException, CannotCompileException {
  StringBuilder result = new StringBuilder();
  int modifiers = method.getModifiers();
  if (Modifier.isPrivate(modifiers) || Modifier.isPackage(modifiers)) {
    modifiers = Modifier.setProtected(modifiers);
  }
  modifiers = Modifier.clear(modifiers, Modifier.FINAL);

  String returnType = toMirrorSafeName(ctClass, method.getReturnType());
  result.append(Modifier.toString(modifiers) + " " + returnType + " " + method.getName() + "(");
  for (int i = 0; i < method.getParameterTypes().length; i++) {
    CtClass parameter = method.getParameterTypes()[i];
    if (i != 0)
      result.append(", ");
    String parameterName = toMirrorSafeName(ctClass, parameter);
    result.append(parameterName + " $" + (i + 1));
  }
  result.append(")");

  if (Modifier.isAbstract(modifiers)) {
    result.append(";\n");
  }
  else {
    result.append("{\n");
    String defaultValue = getDefaultValue(method.getReturnType());
    if (defaultValue != null) {
      result.append("return " + defaultValue + ";\n");
    }
    result.append("}");
  }
  return result.toString();
}

private String toMirrorSafeName(CtClass containingClass, CtClass type) throws NotFoundException {
  if (Arrays.asList(containingClass.getDeclaredClasses()).contains(type)) {
    return type.getName().substring(type.getName().lastIndexOf('$') + 1) + "_Mirror";
  }
  else {
    return type.getName();
  }
}

private String getDefaultValue(CtClass clazz) {
  String name = clazz.getName();
  if ("void".equals(name)) return null;
  else if ("char".equals(name)) return "'x'";
  else if ("short".equals(name)) return "0";
  else if ("int".equals(name)) return "0";
```

```
    else if ("long".equals(name)) return "0";
    else if ("boolean".equals(name)) return "false";
    else if ("float".equals(name)) return "0.0";
    else if ("double".equals(name)) return "0.0";
    else return "null";
  }

}
```

Appendix F ExtensionClassValidator.java

```java
package org.zeroturnaround.javassist.annotation.processor.validator;

import java.util.List;

import javassist.ClassClassPath;
import javassist.ClassPool;
import javassist.CtClass;
import javassist.CtField;
import javassist.CtMethod;

import javax.annotation.processing.Messager;
import javax.lang.model.element.Element;
import javax.lang.model.element.ElementKind;
import javax.lang.model.element.ExecutableElement;
import javax.lang.model.element.TypeElement;
import javax.lang.model.element.VariableElement;
import javax.lang.model.type.DeclaredType;
import javax.lang.model.type.TypeMirror;
import javax.tools.Diagnostic;

import org.zeroturnaround.javassist.annotation.Modify;

/**
 * Validator for extension class
 */
public class ExtensionClassValidator {

  private final TypeElement extensionClass;
  private final TypeMirror originalClass;
  private final Messager messager;

  public ExtensionClassValidator(TypeElement extensionClass, TypeMirror originalClass, Messager messager) {
    this.extensionClass = extensionClass;
    this.originalClass = originalClass;
    this.messager = messager;
  }

  public void validate() {
    System.out.println("Validating extension class: " + extensionClass.toString());
    try {
      ClassPool classPool = ClassPool.getDefault();
      classPool.insertClassPath(new ClassClassPath(this.getClass()));
      CtClass ctClass = classPool.get(originalClass.toString());
//      CtConstructor[] existingConstructors = ctClass.getDeclaredConstructors();

      for (Element element : extensionClass.getEnclosedElements()) {
        System.out.println("Validating existence of: " + element.toString());
        boolean shouldExist = false;
        if (element.getAnnotation(Modify.class) != null || element.getAnnotation(Override.class) != null) {
          shouldExist = true;
        }
        boolean doesExist = false;
        // check if field exists in the original class
        if (element.getKind() == ElementKind.FIELD) {
          VariableElement variable = (VariableElement) element;
          for (CtField existingField : ctClass.getDeclaredFields()) {
            if (existingField.getName().equals(variable.getSimpleName())) {
              doesExist = true;
              break;
            }
          }
          for (CtField existingField : ctClass.getFields()) {
            if (existingField.getName().equals(variable.getSimpleName())) {
              doesExist = true;
              break;
            }
          }
        }

        // check if method exists in the original class hierarchy
        if (element.getKind() == ElementKind.METHOD) {
          ExecutableElement method = (ExecutableElement) element;
          for (CtMethod existingMethod : ctClass.getDeclaredMethods()) {
            if (existingMethod.getName().equals(method.getSimpleName().toString()) &&
                compareTypes(existingMethod.getParameterTypes(), method.getParameters())) {
              doesExist = true;
              break;
            }
          }
          for (CtMethod existingMethod : ctClass.getMethods()) {
```

```
          if (existingMethod.getName().equals(method.getSimpleName().toString()) &&
                compareTypes(existingMethod.getParameterTypes(), method.getParameters())) {
            doesExist = true;
            break;
          }
        }

        // check if method exists in the interface
        for (TypeMirror implementedInterface : extensionClass.getInterfaces()) {
          DeclaredType actualInterface = (DeclaredType) implementedInterface;
          for (Element interfaceElement : actualInterface.asElement().getEnclosedElements()) {
            if (interfaceElement.getKind() == ElementKind.METHOD) {
              ExecutableElement interfaceMethod = (ExecutableElement) interfaceElement;
              if (interfaceMethod.getSimpleName().equals(method.getSimpleName()) &&
                    compareTypes(interfaceMethod.getParameters(), method.getParameters())) {
                doesExist = true;
                break;
              }
            }
          }
//          annotation.getAnnotationType().
//          for (implementedInterface.) {
//            System.out.println("IFACE METHOD: " + interfaceMethod.getName());
//            if (interfaceMethod.getName().equals(method.getSimpleName().toString()) &&
       compareTypes(interfaceMethod.getParameterTypes(), method.getParameters())) {
//              doesExist = true;
//              break;
//            }
//          }
        }
      }

      if (shouldExist && !doesExist) {
        System.out.println("Element " + element.toString() + " does not exist in the original class hierarchy
              or an interface!");
        messager.printMessage(Diagnostic.Kind.ERROR, "The method/field/constructor "+element.toString()+" of
              type ... must override or implement a supertype method/field/constructor", element);
      }
      if (!shouldExist && doesExist) {
        System.out.println("Element " + element.toString() + " already exists in the original class hierarchy
              or an interface!");
        messager.printMessage(Diagnostic.Kind.ERROR, "The method/field/constructor "+element.toString()+"
              exists in the original class and must declare @Modify or @Override", element);
      }
    }
  } catch (Exception e) {

  }
}

/**
 * Returns true if all types represented as CtClass[] elements exist in the list of VariableElement elements.
     Compares by name
 *
 * @return
 */
private boolean compareTypes(CtClass[] first, List<? extends VariableElement> second) {
  if (first == null && second == null) {
    return true;
  }
  if (first == null || second == null) {
    return false;
  }
  if (first.length != second.size()) {
    return false;
  }
  for (int i = 0; i < first.length; i++) {
    System.out.println("FIRST: " + first[i].getName() + ", SECOND: " + second.get(i).asType().toString());
    if (!first[i].getName().toString().equals(second.get(i).asType().toString())) {
      return false;
    }
  }
  return true;
}

private boolean compareTypes(List<? extends VariableElement> first, List<? extends VariableElement> second) {
  if (first == null && second == null) {
    return true;
  }
  if (first == null || second == null) {
    return false;
  }
  if (first.size() != second.size()) {
    return false;
    }
    for (int i = 0; i < first.size(); i++) {
      System.out.println("FIRST: " + first.get(i).asType() + ", SECOND: " + second.get(i).asType());
      if (!first.get(i).asType().toString().equals(second.get(i).asType().toString())) {
        return false;
      }
    }
    return true;
  }

}
```

Appendix G WiringClassGenerator.java

```java
package $original.packageName;

import java.util.*;
import javassist.*;
import javassist.expr.*;
import javassist.bytecode.*;
import org.zeroturnaround.javassist.annotation.*;

@Transformer("$original.name")
public class $original.cbpSimpleName implements
       org.zeroturnaround.javassist.annotation.processor.wiring.JavassistClassBytecodeProcessor {

  private static final String EXTENSION_FIELD = "__companion";
  private static final String ORIGINAL_FIELD = "__original";
  private static final String ORIGINAL_METHOD = "__original";
  private static final String ORIGINAL_AWARE_CLASS = "org.zeroturnaround.javassist.annotation.OriginalAware";

  public void process(final ClassPool cp, final ClassLoader cl, final CtClass originalClass) throws Exception {
    originalClass.addField(CtField.make("final $extension.name "+EXTENSION_FIELD+" = new $extension.name ();" ,
           originalClass));

    CtClass extensionClass = cp.get("$extension.name");
    extensionClass.addField(CtField.make("private $original.name "+ORIGINAL_FIELD+" = null;", extensionClass));
    extensionClass.addInterface(cp.get(ORIGINAL_AWARE_CLASS));
    extensionClass.addMethod(CtMethod.make("public void setOriginal(Object original) { "+ORIGINAL_FIELD+" =
           ($original.name) original; }", extensionClass));

    for (final CtMethod extensionMethod : extensionClass.getDeclaredMethods()) {
      // for all methods that are overriden in the companion class, create a copy of the original method
      // and redirect the original method to call the method defined in the companion class (which will call
      // the created copy of the original method if required.

      CtMethod originalMethod = null;
      CtMethod originalCopy = null;
      try {
        // throws NotFoundException if not a method override!
        originalMethod = originalClass.getDeclaredMethod(extensionMethod.getName(),
              extensionMethod.getParameterTypes());
        if (extensionMethod.hasAnnotation(Modify.class)) {
          originalMethod.setModifiers(extensionMethod.getModifiers());
        } else {
          //
        }
        originalCopy = CtNewMethod.copy(originalMethod, extensionMethod.getName() + ORIGINAL_METHOD,
              originalClass, null);
        System.out.println("Copy method: "+originalMethod.getName()+"() -> "+originalCopy.getName()+"()");
        originalClass.addMethod(originalCopy);
        String target = Modifier.isStatic(extensionMethod.getModifiers()) ? extensionClass.getName() :
              EXTENSION_FIELD;
        if ("void".equals(originalMethod.getReturnType().getName())) {
          originalMethod.setBody("{ "+target+"."+extensionMethod.getName()+"($$);}");
        } else {
          originalMethod.setBody("{ return "+target+"."+extensionMethod.getName()+"($$);}");
        }
      } catch (NotFoundException ignored) {} // expected, not all methods of the companion exist in the original

      // Find anonymous inner classes of types MethodCall instantiated within this method
      final List<String> instrumentClassNames = new ArrayList<String>();
      extensionMethod.instrument(new ExprEditor() {
        public void edit(final NewExpr c) throws CannotCompileException {
          try {
            String className = c.getClassName();
            if (Arrays.asList(cp.get(className).getInterfaces()).contains(
                cp.get(org.zeroturnaround.javassist.annotation.MethodCall.class.getName()))){
              instrumentClassNames.add(className);
            }
          } catch (NotFoundException ignored) { ignored.printStackTrace(); }
        }
      });

      // The companion class was compiled against a mirror of the original, overwrite all method and field
             accesses
      // to invoke the original
      final CtMethod finalOriginalCopy = originalCopy;
      extensionMethod.instrument(new ExprEditor() {
        public void edit(final MethodCall m) throws CannotCompileException {
          try {
          if (m.getClassName().equals(originalClass.getName() + "_Mirror")) {
              // Register a method instrument/edit for the body in methods that call Instrument
            if (m.getSignature().contains(org.zeroturnaround.javassist.annotation.MethodCall.class
      .getName().replace('.', '/'))) {
        /* extract method */
        for (final String instrumentClassName : instrumentClassNames) {
          System.out.println("Instrumenting method:" + finalOriginalCopy.getSignature() + ", class:"
               +instrumentClassName);
            CtClass instrumentClass = cp.get(instrumentClassName);
            for (final CtMethod instrumentMethod : instrumentClass.getDeclaredMethods()) {
              final String instrumentName = instrumentMethod.getName();
              final CtClass instrumentReturnType = instrumentMethod.getReturnType();
              final List<CtClass> instrumentParameterTypes = new
                     ArrayList<CtClass>(Arrays.asList(instrumentMethod.getParameterTypes()));
              final CtClass instrumentTargetType = instrumentParameterTypes.remove(0);
              // Add an instrumenter as defined in the anonymous class
              finalOriginalCopy.instrument(new ExprEditor() {
                public void edit(MethodCall m2) throws CannotCompileException {
                  try {
                    System.out.println("Replace: "+finalOriginalCopy.getSignature()+" " +
                           m2.getClassName() +", " + m2.getMethodName());
                    if (m2.getClassName().equals(instrumentTargetType.getName()) &&
                           m2.getMethodName().equals(instrumentName)
                        && Arrays.equals(m2.getMethod().getParameterTypes(),
                               instrumentParameterTypes.toArray())) {
                      //TODO: hacky solution, need to save the instance somewhere
                      //TODO: does not support redirecting method call to a method inside the same
                             class/inside a mirror
                      m2.replace("$_ = new "+instrumentClassName+"(__companion)."+instrumentName+"($0,
                             $$);"); // TODO: hacky solution, need to save the instance somewhere.
                    }
                  } catch (NotFoundException ignored) { ignored.printStackTrace(); }
                }
              });
            }
          }
        m.replace("");
      } else {
        /* extract method rewireMethodCallFromMirrorToOriginal */
          System.out.println("Rewiring method call in extension:
               "+m.getClassName()+"."+m.getMethodName()+"() -> "+
            originalClass.getName()+"."+m.getMethodName()+"__original()");

          CtMethod originalMethod = originalClass.getMethod(m.getMethod().getName(),
               m.getMethod().getSignature());

          // If the modifier for the original method is private, we need to create a synthetic
          // accessor for it. Ignore already existing synthetic methods and just create
          // new ones for things that we need.
          String syntheticMethodName = null;
          if (Modifier.isPrivate(originalMethod.getModifiers())) {
          syntheticMethodName = createSyntheticAccessor(originalClass, originalMethod);
          }

          if (syntheticMethodName != null) {
          if ("void".equals(m.getMethod().getReturnType().getName())) {
            m.replace("{ "+originalClass.getName()+"." + syntheticMethodName +"("+ORIGINAL_FIELD+", $$);
                 }");
          } else {
              m.replace("{ $_ = "+originalClass.getName()+"." + syntheticMethodName +"("+ORIGINAL_FIELD+",
                   $$); }");
            }
          } else {
            String targetMethodName = m.getMethodName();
            String target = Modifier.isStatic(m.getMethod().getModifiers()) ? originalClass.getName() :
                 ORIGINAL_FIELD;
            if ("void".equals(m.getMethod().getReturnType().getName())) {
              m.replace("{ "+target+"." + m.getMethodName() + ORIGINAL_METHOD+"($$); }");
            } else {
              m.replace("{ $_ = "+target+"." + m.getMethodName() + ORIGINAL_METHOD+"($$); }");
            }
          }
        }
  }
  } catch (NotFoundException e) { e.printStackTrace();}
}
public void edit(FieldAccess f) throws CannotCompileException {
  try {
  if (f.getField().getDeclaringClass().getName().equals(originalClass.getName() +"_Mirror")) {
    /* extract method rewireFieldAccessFromMirrorToOriginal */
      System.out.println("Rewiring field access in extension:
           "+f.getField().getDeclaringClass().getName()+"."+
        f.getFieldName()+" -> "+originalClass.getName()+"."+f.getFieldName());
      CtField originalField = originalClass.getDeclaredField(f.getFieldName());

      // If the modifier for the original field is private, we need to create a synthetic
      // accessor for it. Ignore already existing synthetic methods and just create
```

```
          // new ones for things that we need.
          String syntheticMethodName = null;
          if (Modifier.isPrivate(originalField.getModifiers())) {
            syntheticMethodName = createSyntheticAccessor(originalClass, originalField);
          }

          if (syntheticMethodName != null) {
            if (f.isReader()) {
              f.replace("{ $_ = "+originalClass.getName()+"."+syntheticMethodName+"("+ORIGINAL_FIELD+")}");
            }
            if (f.isWriter()) {
              f.replace("{ "+originalClass.getName()+"."+syntheticMethodName+"("+ORIGINAL_FIELD+", $1)}");
            }
          } else {
            String target = Modifier.isStatic(f.getField().getModifiers()) ? originalClass.getName() :
                   ORIGINAL_FIELD;
            if (f.isReader()) {
              f.replace("{ $_ = "+target+"." + f.getFieldName() +"; }");
            }
            if (f.isWriter()) {
              f.replace("{"+target+"."+f.getFieldName()+" = $1; }");
            }
          }
        }
      } catch (NotFoundException e) { e.printStackTrace(); }
    }
  });
}

// Static block of extension is executed after original static block
CtConstructor extStaticInitializer = extensionClass.getClassInitializer();
if (extStaticInitializer != null) {
  System.out.println("STATIC INIT: " + extStaticInitializer.toString());
  rewireExtensionClassMethodsToCallOriginal(extStaticInitializer, originalClass);
  CtConstructor origStaticInitializer = originalClass.getClassInitializer();
  if (origStaticInitializer == null) {
    originalClass.makeClassInitializer().setBody(extStaticInitializer, null);
  } else {
    origStaticInitializer.insertAfter(extStaticInitializer.toString());
  }
}



for (CtConstructor extensionConstructor : extensionClass.getDeclaredConstructors()) {

}
for (CtConstructor originalConstructor : originalClass.getDeclaredConstructors()) {
  originalConstructor.insertAfter("{ (("+ORIGINAL_AWARE_CLASS+")"+EXTENSION_FIELD+").setOriginal(this); }");
}

for (CtField extensionField : extensionClass.getDeclaredFields()) {
  if (extensionField.hasAnnotation(Modify.class)) {
    CtField originalField = originalClass.getField(extensionField.getName());
    originalField.setModifiers(extensionField.getModifiers());
  } else {

  }
}

extensionClass.toClass(); // TODO: NB! it might be incorrect to call this in the context of the
       ContextClassLoader of a given thread.
}

private void rewireExtensionClassMethodsToCallOriginal(final CtBehavior extensionClassExecutable, final
      CtClass originalClass) throws CannotCompileException, NotFoundException {
  System.out.println("Rewiring extension class executable to call original:
       "+extensionClassExecutable.getDeclaringClass().getName()+"."+
    extensionClassExecutable.getName()+extensionClassExecutable.getSignature());
  extensionClassExecutable.instrument(new ExprEditor() {
    public void edit(MethodCall m) throws CannotCompileException {
      try {
        if (m.getClassName().equals(originalClass.getName() + "_Mirror")) { // is a call on a mirror?
          System.out.println(originalClass.getName()+"."+m.getMethodName() + m.getSignature());
          String target = Modifier.isStatic(m.getMethod().getModifiers()) ? originalClass.getName() :
                ORIGINAL_FIELD;
          if ("void".equals(m.getMethod().getReturnType().getName())) {
            m.replace("{ "+target+"." + m.getMethodName() + ORIGINAL_METHOD+"($$); }");
          } else {
            m.replace("{ $_ = "+target+"." + m.getMethodName() + ORIGINAL_METHOD+"($$); }");
          }
        }
      } catch (NotFoundException e) {
        e.printStackTrace(); // TODO
      }
    }

    public void edit(FieldAccess f) throws CannotCompileException {
      try {
        if (f.getField().getDeclaringClass().getName().equals(originalClass.getName() +"_Mirror")) { // is a
               call on a mirror?
          System.out.println(originalClass.getName()+"."+f.getFieldName());
          String target = Modifier.isStatic(f.getField().getModifiers()) ? originalClass.getName() :
                 ORIGINAL_FIELD;
          if (f.isReader()) {
            f.replace("{ $_ = "+target+"." + f.getFieldName() +"; }");
          }
          if (f.isWriter()) {
            f.replace("{"+target+"."+f.getFieldName()+" = $1; }");
          }
        }
      } catch (NotFoundException e) {
        e.printStackTrace(); //TODO
      }
    }
  });
}

private String createSyntheticAccessor(CtClass originalClass, CtField originalField) throws NotFoundException,
       CannotCompileException {
  System.out.println("creating a synthetic getter/setter for: " + originalField.getType().getName() +
       originalField.getName());
  String syntheticMethodName = getUniqueSyntheticMethodName();
  String syntheticGetterDeclaration =
      "static " +originalField.getType().getName()+" "+syntheticMethodName+"("+originalClass.getName()+"
             instance) {"+
        "return instance."+originalField.getName()+";"+
      "}";
  System.out.println(syntheticGetterDeclaration);
  originalClass.addMethod(CtNewMethod.make(syntheticGetterDeclaration, originalClass));

  String syntheticSetterDeclaration =
      "static void "+syntheticMethodName+"("+originalClass.getName()+" instance,
             "+originalField.getType().getName()+" value) {"+
        "instance."+originalField.getName()+" = value;"+
      "}";
  System.out.println(syntheticSetterDeclaration);
  originalClass.addMethod(CtNewMethod.make(syntheticSetterDeclaration, originalClass));
  return syntheticMethodName;
}

private String createSyntheticAccessor(CtClass originalClass, CtMethod originalMethod) throws
       NotFoundException, CannotCompileException {
  System.out.println("creating a synthetic method for: " + originalMethod.getName() +
       originalMethod.getSignature());
  String syntheticMethodName = getUniqueSyntheticMethodName();
  String argsString = toArgsString(toClassNames(originalMethod.getParameterTypes()), true);
  String argsStringWithoutTypes = toArgsString(toClassNames(originalMethod.getParameterTypes()), false);
  String syntheticMethodArgs = originalClass.getName()+ " instance" + (!argsString.isEmpty() ? (", "
       +argsString) : "");
  String syntheticMethodDeclaration =
    "static "+originalMethod.getReturnType().getName()+" "+syntheticMethodName+"("+syntheticMethodArgs+") {"+
      "return instance."+originalMethod.getName()+ORIGINAL_METHOD+"("+argsStringWithoutTypes+");"+
    "}";
  System.out.println(syntheticMethodDeclaration);
  originalClass.addMethod(CtNewMethod.make(syntheticMethodDeclaration, originalClass));
  return syntheticMethodName;
}

private String[] toClassNames(CtClass[] ctClasses) {
  String[] result = new String[ctClasses.length];
  for (int i = 0; i < ctClasses.length; i++) {
    result[i] = ctClasses[i].getName();
  }
  return result;
}

private String toArgsString(String[] argClasses, boolean includeTypes) {
  if (argClasses == null || argClasses.length == 0) {
    return "";
  } else {
    String result = "";
    boolean addComma = false;
    for (int i = 0; i < argClasses.length; i++) {
      result += addComma ? ", " : "";
      result += includeTypes ? argClasses[i] + " " : "";
      result += "$synarg"+(i+1);
      addComma = true;
```

```java
      }
      return result;
    }
  }

  // should be good enough for avoiding collisions
  private int syntheticMethodCounter = 9999;
  private String getUniqueSyntheticMethodName() {
    return "access$" + syntheticMethodCounter--;
  }
}
```

Appendix H cbp.vtl

```
package $original.packageName;

import java.util.*;
import javassist.*;
import javassist.expr.*;
import javassist.bytecode.*;
import org.zeroturnaround.javassist.annotation.Modify;

public class $original.cbpSimpleName implements
      org.zeroturnaround.javassist.annotation.processor.wiring.JavassistClassBytecodeProcessor {

  private static final String EXTENSION_FIELD = "__companion";
  private static final String ORIGINAL_FIELD = "__original";
  private static final String ORIGINAL_METHOD = "__original";
  private static final String ORIGINAL_AWARE_CLASS = "org.zeroturnaround.javassist.annotation.OriginalAware";

  public void process(final ClassPool cp, final ClassLoader cl, final CtClass originalClass) throws Exception {
    originalClass.addField(CtField.make("final $extension.name "+EXTENSION_FIELD+" = new $extension.name ();" ,
          originalClass));

    CtClass extensionClass = cp.get("$extension.name");
    extensionClass.addField(CtField.make("private $original.name "+ORIGINAL_FIELD+" = null;", extensionClass));
    extensionClass.addInterface(cp.get(ORIGINAL_AWARE_CLASS));
    extensionClass.addMethod(CtMethod.make("public void setOriginal(Object original) { "+ORIGINAL_FIELD+" =
          ($original.name) original; }", extensionClass));

    for (final CtMethod extensionMethod : extensionClass.getDeclaredMethods()) {
      // for all methods that are overriden in the companion class, create a copy of the original method
      // and redirect the original method to call the method defined in the companion class (which will call
      // the created copy of the original method if required.

      CtMethod originalMethod = null;
      CtMethod originalCopy = null;
      try {
        // throws NotFoundException if not a method override!
        originalMethod = originalClass.getDeclaredMethod(extensionMethod.getName(),
              extensionMethod.getParameterTypes());
        if (extensionMethod.hasAnnotation(Modify.class)) {
          originalMethod.setModifiers(extensionMethod.getModifiers());
        } else {
          //
        }
        originalCopy = CtNewMethod.copy(originalMethod, extensionMethod.getName() + ORIGINAL_METHOD,
              originalClass, null);
        System.out.println("Copy method: "+originalMethod.getName()+"() -> "+originalCopy.getName()+"()");
        originalClass.addMethod(originalCopy);
        String target = Modifier.isStatic(extensionMethod.getModifiers()) ? extensionClass.getName() :
              EXTENSION_FIELD;
        if ("void".equals(originalMethod.getReturnType().getName())) {
          originalMethod.setBody("{ "+target+"."+extensionMethod.getName()+"($$);}");
        } else {
          originalMethod.setBody("{ return "+target+"."+extensionMethod.getName()+"($$);}");
        }
      } catch (NotFoundException ignored) {} // expected, not all methods of the companion exist in the original

      // Find anonymous inner classes of types MethodCall instantiated within this method
      final List<String> instrumentClassNames = new ArrayList<String>();
      extensionMethod.instrument(new ExprEditor() {
        public void edit(final NewExpr c) throws CannotCompileException {
          try {
            String className = c.getClassName();
            if (Arrays.asList(cp.get(className).getInterfaces()).contains(
                cp.get(org.zeroturnaround.javassist.annotation.MethodCall.class.getName()))){
              instrumentClassNames.add(className);
            }
          } catch (NotFoundException ignored) { ignored.printStackTrace(); }
        }
      });

      // The companion class was compiled against a mirror of the original, overwrite all method and field
            accesses
      // to invoke the original
      final CtMethod finalOriginalCopy = originalCopy;
      extensionMethod.instrument(new ExprEditor() {
        public void edit(final MethodCall m) throws CannotCompileException {
          try {
          if (m.getClassName().equals(originalClass.getName() + "_Mirror")) {
              // Register a method instrument/edit for the body in methods that call Instrument
            if
                  (m.getSignature().contains(org.zeroturnaround.javassist.annotation.MethodCall.class.getName().replace('.',
           '/'))) {
        /* extract method */
        for (final String instrumentClassName : instrumentClassNames) {
          System.out.println("Instrumenting method:" + finalOriginalCopy.getSignature() + ", class:"
                +instrumentClassName);
            CtClass instrumentClass = cp.get(instrumentClassName);
            for (final CtMethod instrumentMethod : instrumentClass.getDeclaredMethods()) {
              final String instrumentName = instrumentMethod.getName();
              final CtClass instrumentReturnType = instrumentMethod.getReturnType();
              final List<CtClass> instrumentParameterTypes = new
                    ArrayList<CtClass>(Arrays.asList(instrumentMethod.getParameterTypes()));
              final CtClass instrumentTargetType = instrumentParameterTypes.remove(0);
              // Add an instrumenter as defined in the anonymous class
              finalOriginalCopy.instrument(new ExprEditor() {
                public void edit(MethodCall m2) throws CannotCompileException {
                  try {
                    System.out.println("Replace: "+finalOriginalCopy.getSignature()+" " +
                          m2.getClassName() +", " + m2.getMethodName());
                    if (m2.getClassName().equals(instrumentTargetType.getName()) &&
                          m2.getMethodName().equals(instrumentName)
                        && Arrays.equals(m2.getMethod().getParameterTypes(),
                              instrumentParameterTypes.toArray())) {
                      //TODO: hacky solution, need to save the instance somewhere
                      //TODO: does not support redirecting method call to a method inside the same
                            class/inside a mirror
                      m2.replace("$_ = new "+instrumentClassName+"(__companion)."+instrumentName+"($0,
                            $$);");
                    }
                  } catch (NotFoundException ignored) { ignored.printStackTrace(); }
                }
              });
            }
          }
        m.replace("");
      } else {
        /* extract method rewireMethodCallFromMirrorToOriginal */
          System.out.println("Rewiring method call in extension:
                "+m.getClassName()+"."+m.getMethodName()+"() -> "+
            originalClass.getName()+"."+m.getMethodName()+"__original()");

          CtMethod originalMethod = originalClass.getMethod(m.getMethod().getName(),
                m.getMethod().getSignature());

          // If the modifier for the original method is private, we need to create a synthetic
          // accessor for it. Ignore already existing synthetic methods and just create
          // new ones for things that we need.
          String syntheticMethodName = null;
          if (Modifier.isPrivate(originalMethod.getModifiers())) {
          syntheticMethodName = createSyntheticAccessor(originalClass, originalMethod);
          }

          if (syntheticMethodName != null) {
          if ("void".equals(m.getMethod().getReturnType().getName())) {
            m.replace("{ "+originalClass.getName()+"." + syntheticMethodName +"("+ORIGINAL_FIELD+", $$);
                  }");
          } else {
              m.replace("{ $_ = "+originalClass.getName()+"." + syntheticMethodName +"("+ORIGINAL_FIELD+",
                    $$); }");
            }
          } else {
            String targetMethodName = m.getMethodName();
            String target = Modifier.isStatic(m.getMethod().getModifiers()) ? originalClass.getName() :
                  ORIGINAL_FIELD;
            if ("void".equals(m.getMethod().getReturnType().getName())) {
              m.replace("{ "+target+"." + m.getMethodName() + ORIGINAL_METHOD+"($$); }");
            } else {
              m.replace("{ $_ = "+target+"." + m.getMethodName() + ORIGINAL_METHOD+"($$); }");
            }
          }
        }
  }
  } catch (NotFoundException e) { e.printStackTrace();}
}
public void edit(FieldAccess f) throws CannotCompileException {
  try {
  if (f.getField().getDeclaringClass().getName().equals(originalClass.getName() +"_Mirror")) {
    /* extract method rewireFieldAccessFromMirrorToOriginal */
      System.out.println("Rewiring field access in extension:
            "+f.getField().getDeclaringClass().getName()+"."+
        f.getFieldName()+" -> "+originalClass.getName()+"."+f.getFieldName());
      CtField originalField = originalClass.getDeclaredField(f.getFieldName());

      // If the modifier for the original field is private, we need to create a synthetic
      // accessor for it. Ignore already existing synthetic methods and just create
```

```
          // new ones for things that we need.
          String syntheticMethodName = null;
          if (Modifier.isPrivate(originalField.getModifiers())) {
            syntheticMethodName = createSyntheticAccessor(originalClass, originalField);
          }

          if (syntheticMethodName != null) {
            if (f.isReader()) {
              f.replace("{ $_ = "+originalClass.getName()+"."+syntheticMethodName+"("+ORIGINAL_FIELD+")}");
            }
            if (f.isWriter()) {
              f.replace("{ "+originalClass.getName()+"."+syntheticMethodName+"("+ORIGINAL_FIELD+", $1)}");
            }
          } else {
            String target = Modifier.isStatic(f.getField().getModifiers()) ? originalClass.getName() :
                   ORIGINAL_FIELD;
            if (f.isReader()) {
              f.replace("{ $_ = "+target+"." + f.getFieldName() +"; }");
            }
            if (f.isWriter()) {
              f.replace("{"+target+"."+f.getFieldName()+" = $1; }");
            }
          }
      }
      } catch (NotFoundException e) { e.printStackTrace(); }
    }
  });
}

// Static block of extension is executed after original static block
CtConstructor extStaticInitializer = extensionClass.getClassInitializer();
if (extStaticInitializer != null) {
  System.out.println("STATIC INIT: " + extStaticInitializer.toString());
  rewireExtensionClassMethodsToCallOriginal(extStaticInitializer, originalClass);
  CtConstructor origStaticInitializer = originalClass.getClassInitializer();
  if (origStaticInitializer == null) {
    originalClass.makeClassInitializer().setBody(extStaticInitializer, null);
  } else {
    origStaticInitializer.insertAfter(extStaticInitializer.toString());
  }
}



for (CtConstructor extensionConstructor : extensionClass.getDeclaredConstructors()) {

}
for (CtConstructor originalConstructor : originalClass.getDeclaredConstructors()) {
  originalConstructor.insertAfter("{ (("+ORIGINAL_AWARE_CLASS+")"+EXTENSION_FIELD+").setOriginal(this); }");
}

for (CtField extensionField : extensionClass.getDeclaredFields()) {
  if (extensionField.hasAnnotation(Modify.class)) {
    CtField originalField = originalClass.getField(extensionField.getName());
    originalField.setModifiers(extensionField.getModifiers());
  } else {

  }
}

extensionClass.toClass(); // TODO: NB! it might be incorrect to call this in the context of the
       ContextClassLoader of a given thread.
}

private void rewireExtensionClassMethodsToCallOriginal(final CtBehavior extensionClassExecutable, final
      CtClass originalClass) throws CannotCompileException, NotFoundException {
  System.out.println("Rewiring extension class executable to call original:
       "+extensionClassExecutable.getDeclaringClass().getName()+"."+
  extensionClassExecutable.getName()+extensionClassExecutable.getSignature());
  extensionClassExecutable.instrument(new ExprEditor() {
    public void edit(MethodCall m) throws CannotCompileException {
      try {
        if (m.getClassName().equals(originalClass.getName() + "_Mirror")) { // is a call on a mirror?
          System.out.println(originalClass.getName()+"."+m.getMethodName() + m.getSignature());
          String target = Modifier.isStatic(m.getMethod().getModifiers()) ? originalClass.getName() :
                ORIGINAL_FIELD;
          if ("void".equals(m.getMethod().getReturnType().getName())) {
            m.replace("{ "+target+"." + m.getMethodName() + ORIGINAL_METHOD+"($$); }");
          } else {
            m.replace("{ $_ = "+target+"." + m.getMethodName() + ORIGINAL_METHOD+"($$); }");
          }
        }
      } catch (NotFoundException e) {
        e.printStackTrace(); // TODO
      }
    }

    public void edit(FieldAccess f) throws CannotCompileException {
      try {
        if (f.getField().getDeclaringClass().getName().equals(originalClass.getName() +"_Mirror")) { // is a
               call on a mirror?
          System.out.println(originalClass.getName()+"."+f.getFieldName());
          String target = Modifier.isStatic(f.getField().getModifiers()) ? originalClass.getName() :
                 ORIGINAL_FIELD;
          if (f.isReader()) {
            f.replace("{ $_ = "+target+"." + f.getFieldName() +"; }");
          }
          if (f.isWriter()) {
            f.replace("{"+target+"."+f.getFieldName()+" = $1; }");
          }
        }
      } catch (NotFoundException e) {
        e.printStackTrace(); //TODO
      }
    }
  });
}

private String createSyntheticAccessor(CtClass originalClass, CtField originalField) throws NotFoundException,
       CannotCompileException {
  System.out.println("creating a synthetic getter/setter for: " + originalField.getType().getName() +
       originalField.getName());
  String syntheticMethodName = getUniqueSyntheticMethodName();
  String syntheticGetterDeclaration =
      "static " +originalField.getType().getName()+" "+syntheticMethodName+"("+originalClass.getName()+"
             instance) {"+
        "return instance."+originalField.getName()+";"+
      "}";
  System.out.println(syntheticGetterDeclaration);
  originalClass.addMethod(CtNewMethod.make(syntheticGetterDeclaration, originalClass));

  String syntheticSetterDeclaration =
      "static void "+syntheticMethodName+"("+originalClass.getName()+" instance,
             "+originalField.getType().getName()+" value) {"+
        "instance."+originalField.getName()+" = value;"+
      "}";
  System.out.println(syntheticSetterDeclaration);
  originalClass.addMethod(CtNewMethod.make(syntheticSetterDeclaration, originalClass));
  return syntheticMethodName;
}

private String createSyntheticAccessor(CtClass originalClass, CtMethod originalMethod) throws
       NotFoundException, CannotCompileException {
  System.out.println("creating a synthetic method for: " + originalMethod.getName() +
       originalMethod.getSignature());
  String syntheticMethodName = getUniqueSyntheticMethodName();
  String argsString = toArgsString(toClassNames(originalMethod.getParameterTypes()), true);
  String argsStringWithoutTypes = toArgsString(toClassNames(originalMethod.getParameterTypes()), false);
  String syntheticMethodArgs = originalClass.getName()+ " instance" + (!argsString.isEmpty() ? (", "
       +argsString) : "");
  String syntheticMethodDeclaration =
    "static "+originalMethod.getReturnType().getName()+" "+syntheticMethodName+"("+syntheticMethodArgs+") {"+
      "return instance."+originalMethod.getName()+ORIGINAL_METHOD+"("+argsStringWithoutTypes+");"+
    "}";
  System.out.println(syntheticMethodDeclaration);
  originalClass.addMethod(CtNewMethod.make(syntheticMethodDeclaration, originalClass));
  return syntheticMethodName;
}

private String[] toClassNames(CtClass[] ctClasses) {
  String[] result = new String[ctClasses.length];
  for (int i = 0; i < ctClasses.length; i++) {
    result[i] = ctClasses[i].getName();
  }
  return result;
}

private String toArgsString(String[] argClasses, boolean includeTypes) {
  if (argClasses == null || argClasses.length == 0) {
    return "";
  } else {
    String result = "";
    boolean addComma = false;
    for (int i = 0; i < argClasses.length; i++) {
      result += addComma ? ", " : "";
      result += includeTypes ? argClasses[i] + " " : "";
      result += "$synarg"+(i+1);
      addComma = true;
      }
      return result;
    }
  }

  // should be good enough for avoiding collisions
  private int syntheticMethodCounter = 9999;
  private String getUniqueSyntheticMethodName() {
    return "access$" + syntheticMethodCounter--;
  }
}
```