How to use Javassist for Polymorphism detection

Bachelor's Anleitung
at the
Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch/

by
Michael Morelli
July 2013

led by
Prof. Dr. Oscar Nierstrasz

Abstract

This document is an introduction to the Javassist bytecode manipulation library and shows how to use it for polymorphism detection.

Contents

1 Bytecode manipulation with Javassist 4
2 Implementation 5
2.1 Class Overview 6
2.1.1 UML 6
2.1.2 Classes and responsibility 7
2.2 Static detection 9
2.3 Dynamic detection 16
2.3.1 Reflection 16
2.3.2 Make classes reflective 17
2.3.3 Collect accesses at runtime 18
3 Sources 19

1 Bytecode manipulation with Javassist

There are many good tools to manipulate and extract information from Java bytecode (*.class files). The most known are BCEL of Apache Commons, AspectJ and Javassist.

For this very project we decided to use Javassist, because it should make Java bytecode manipulation simple. It enables Java programs to define a new class at runtime and to modify a class file when the JVM (Java Virtual Machine) loads it. The library provides two levels of API: source level and bytecode level. In the former case the user is able to edit a class file without knowledge of the specifications of the Java bytecode. And that is the API level we are using in our project.

Another plus of Javassist is it has implemented runtime reflection which enables Java programs to use a metaobject that controls method calls on base-level objects. This will be very useful for the Dynamic Polymorphism Detection case where we are heading to get field accesses at runtime. So in fact for the Dynamic Polymorphism Detection implementation all classes are loaded from the class files to the new Javassist JVM and then have a runtime.

The library itself has been written by Shigeru Chiba and is open-source.

2 Implementation

The main workflow of the polymorphism detection implementation is structured like this:

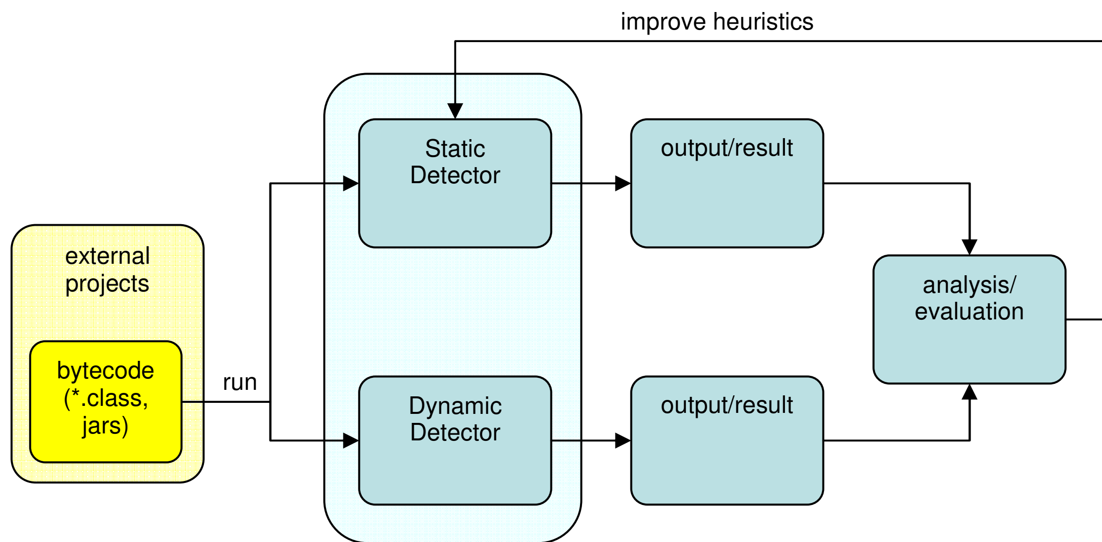

Figure 1: Workflow of the project

Basically Apache Commons libraries serve as external source projects (see chapter Analysis). In fact all class files and jar files of the external project serve as the input to our Detector. Based on these sources the Static- and Dynamic Detector separately and independently produce their output and save them to a separate text file. The output format of both outputs of field accesses is defined as follows:

```
KEY:  [enclosing package].[enclosing class]:
      [package of field type].[field type]:[field name]

VALUE(S):  [package of field type].[field type]
```

Every `KEY` is a certain field in a certain class. And the VALUES are the assigned fields to the KEY-field. If a `KEY` has multiple field types (VALUES), it is polymorphic by definition.

For example a field named `afield` of type `B` inside package `bPackage`, declared inside class `A` of package `aPackage` would appear as the `KEY`:

`[aPackage].[A]:[bPackage].[B]:[aField]`.

And if this field has a `VALUE` of type `valueType` within the package `valuePackage` the value would appear as follows:

```
[value package].[value type]
```

The produced output of the Static- and Dynamic Detector are compared and analyzed in chapter Analysis. Based on the analysis of the heuristics, the Static Detector is improved and with the new heuristics implemented the evaluation of the Static Detector starts again (see Figure 1).

2.1 *Class Overview*

2.1.1 UML

In the following UML-diagram all the classes (without helper classes) along with their public interfaces are drawn:

Controller
controller
Controller()

DynamicDataContainer
dynamicDetection
getFieldWriteTraps():MultiMap
setPool(ClassPool):void
getFieldType(String,String):String
isFieldTypePrimitve(String,String):boolean
-instance
0..1

StaticDataContainer
staticDetection
addCastAtLine(String):void
addReturnType(String,String):void
addFieldKey(String,String):void
addFieldValue(String,String):void
getMultiMap():MultiMap
-instance
0..1

Detector
controller
Detector(String)

MetaClass
dynamicDetection
MetaClass(Object,Object[])
trapMethodcall(int,Object[]):Object
trapFieldRead(String):Object
trapFieldWrite(String,Object):void

DynamicDetector
dynamicDetection
DynamicDetector(String)
run(String,String[]):void
getResult():MultiMap

StaticDetector
staticDetection
StaticDetector(String)
run():void
getResult():MultiMap

StaticFieldReader
staticDetection
StaticFieldReader()
edit(FieldAccess):void
edit(MethodCall):void
edit(Cast):void

FieldNameCollector
dynamicDetection
FieldNameCollector(String,MultiMap)
edit(FieldAccess):void

ModifierRewriter
dynamicDetection
ModifierRewriter()
edit(FieldAccess):void

Metaobject
javassist.tools.reflect
Metaobject(Object,Object[])
getClassMetaobject():ClassMetaobject
getObject():Object
setObject(Object):void
getMethodName(int):String
getParameterTypes(int):Class[]
getReturnType(int):Class
trapFieldRead(String):Object
trapFieldWrite(String,Object):void
trapMethodcall(int,Object[]):Object

ExprEditor
javassist.expr
ExprEditor()
doit(CtClass,MethodInfo):boolean
edit(NewExpr):void
edit(NewArray):void
edit(MethodCall):void
edit(ConstructorCall):void
edit(FieldAccess):void
edit(Instanceof):void
edit(Cast):void
edit(Handler):void

Figure 2: UML of the applications main classes

Additional helper classes (containers):

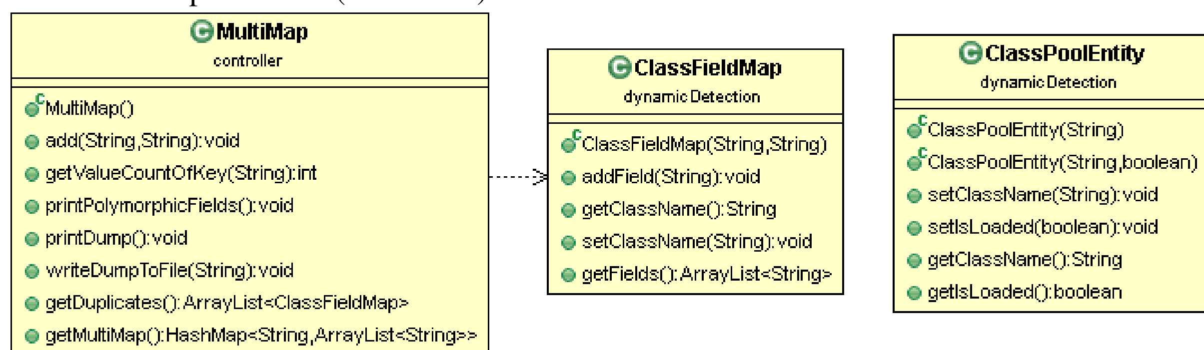

Figure 3: Helper classes

2.1.2 Classes and responsibility

Package	Class	Responsibility
`controller`	`Controller`	Main-class and application runner. Defines the absolute path to the bytecode of the external project.
	`Detector`	Parent class of the Detectors (Static and Dynamic). Has the knowledge of the external projects file system and offers package name of a class. Loads needed libraries of the external project.
	`MultiMap`	Container of the field accesses. Knows which of them are polymorphic. Can print the dump to a file. Detects duplicates inside the dump.
	`InterfaceField-Container`	Container of all field-writes of type Interface. (Used to evaluate heuristic). Writes output to staticInterfaceFields.txt.
`dynamic-Detection`	`DynamicDetector`	Prepares and runs the simulative run to save all field accesses at runtime.
	`DynamicDataContainer`	Container of field accesses at runtime. Uses a `MultiMap` for storage. Has direct access to the pool to get the type of a field, trapped by the `MetaClass`.
	`FieldNameCollector`	A derived class of `Javassist.ExprEditor`. Instruments a class and saves the field accesses in a `MultiMap` container.
	`ModifierRewriter`	A derived class of `Javassist.ExprEditor`. Sets the field modifier of a field to public.
	`MetaClass`	A derived class of `Javassist.Metaobject`. Traps field-writes at runtime and saves the field accesses to the `DynamicDataContainer`.
	`ClassFieldMap`	Container of class name and its fields. A class has many fields where a field can not occur more than once.
	`ClassPoolEntity`	Container of class name and its loading-state. A class is loaded when the `classLoader` has loaded it successfully. Needed to ensure that all classes are loaded.
`static-Detection`	`StaticDetector`	Parses all bytecode files of the external project and saves the result.
	`StaticDataContainer`	Container of field accesses at compile time. Uses a `MultiMap` for storage.
	`StaticFieldReader`	A derived class of `Javassist.ExprEditor` and overrides the `edit()` methods to get all non-primitive field accesses while parsing the external project bytecode.

The class `MultiMap` is an important container to store all field accesses received from Static or Dynamic Detector.
The class is a wrapper around a `HashMap<String, ArrayList<<String>>` with some additionally functionalities. The `add()` method adds a `KEY <String>` with its corresponding `VALUE<String>` to the HashMap `multiMap`. If the key already exists, the value is added to this existing KEY. Otherwise a new KEY is added with the associated value.
The public method `getValueCountOfKey()` returns the number of (distinct) VALUES of a specific KEY. If there is no such KEY zero is returned.
The object `MultiMap` is able to print its content/dump to a .txt file or alternative to the console. The map is also capable to detect duplicates and return them as an `ArrayList<ClassFieldMap>` object. In reference to the unit tests there is a getter which returns a reference to the internally field named `multiMap`.

2.2 Static detection

The Static Detection implementation should give us a good guess at compile time of which fields will be polymorphic at runtime.
To do so, we "parse" the bytecode of the external project. In other words, we are scanning the bytecode directly and save all field accesses to the container `MultiMap`.

But before getting started with the implementation of the Static Polymorphism Detector we have to introduce some definitions concerning the Javassist library:

`Javassist.CtClass` (compile time class):
A compile time class is an abstract representation of a class file. The `CtClass` object is a handle for dealing with a class file (bytecode). We can load a `CtClass` from the class file and then manipulate or inspect this class by the Javassist API. `CtClass` wraps a class file.
Analogous to the `CtClass` there are `CtField` and `CtMethod`.

`Javassist.ClassPool`:
The Javassist `ClassPool` is a container of `CtClasses`. A `CtClass` object must be obtained from this object. If the `ClassPool` method `get(CtClass)` is called, the object searches various sources represented by `ClassPath` to find a class file and then creates a `CtClass` object. The paths to the bytecode class files have to be set before being able to load the compile time classes from the `ClassPool`. Important to mention is that the `ClassPool` holds all the created `CtClasses`. If there are many, the `ClassPool` will consume a huge amount of memory. To avoid this, `CtClasses` of the Pool should be recreated.

To implement the Static Detector we use the `Javassist.ExprEditor`. This class is able to instrument either a compile time class (`CtClass`) or a compile time method (`CtMethod`).

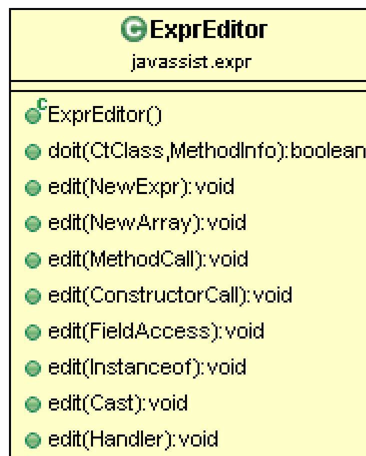

Figure 4: Javassist ExprEditor API

The class `StaticFieldReader` derives from the `Javassist.ExprEditor` and overwrites the methods `edit(FieldAccess)`, `edit(MethodCall)` and `edit(Cast)`. Since the `Javassist.ExprEditor` architecture is similar to the Strategy Pattern we can define the exact behaviours of the `edit()` methods in the derived class `StaticFieldReader`. In our case we instrument a compile time class (`CtClass`) which we get from the `ClassPool`. So every time a `CtClass` is instrumented by our `StaticFieldReader` the `edit(FieldAccess)` method is called for every field access in this class. For every method call in this class the `StaticFieldReader.edit(MethodCall)` is called and for every casting the method `StaticFieldReader.edit(Cast)`.

Since the `StaticDetector` extends the class `Detector` we get the exact absolute path to the bytecode of the external project. The super class `Detector` is responsible for preparing and managing the directory path to the external projects bytecode. The class has the knowledge about the path and can offer the package name of a class.
By the convention of the Eclipse IDE the package name begins after the binary (short: bin) directory in the file system. But there is no way to get the package name of a class directly. So we have to concatenate the package name while traversing the file system. The package name is build from the end of the root directory to the beginning of the class file name.

For example if we have a Windows OS path like:

```
../bin/org/apache/tests/exceptionTests.class
```

Since `bin` is the root directory (the parent directory of the root package `org`), the package name of `exceptionTest.class` would be:

```
org.apache.tests
```

The knowledge of package name is important for loading compile time classes from the `ClassPool` respectively `ClassLoader`. To load or get the class `exceptionTest.class` from the `ClassPool` we have to assign “`org.apache.tests.exceptionTest.class`” as an argument. If we would assign “`exceptionTest.class`” a NotFoundException would occur.
The reason for this behaviour lies in the different class name structures of bytecode and Java code. The class `exceptionTest.java` holds the class `exceptionTest` and imports the package `org.apache.tests` in the file header, but in its bytecode the class is defined as `org.apache.tests.exceptionTest` without importing the package name in the header.

To be able to evaluate the types of the field accesses we have to append the needed libraries to the `ClassPool`. Otherwise we would get an exception by the `StaticFieldReader` while accessing a field’s type which is defined externally and not known to the project. So for every external project we analyse/run with the aid of the `StaticFieldReader`. We have to ensure that all needed types are known. This is reached by the method `StaticDetector.appendLibrariesToPool()`, which sets the search path of the pool to the needed jar-files where the types are declared.

Finally we can iterate through the external projects bytecode. This happens via recursive traversing through the file system to load all class files to a `CtClass` one after the other. Every loaded `CtClass` can now be instrumented by our `StaticFieldReader` object.

As mentioned the `StaticFieldReader` is derived from the `Javassist.ExprEditor` and saves the field accesses, method calls and casts to the `StaticDataContainer`.

Note: The `StaticDataContainer` is a Singleton, since for every `CtClass` a new instance of the `StaticFieldReader` is instantiated; we have to ensure that the written data is saved in the same singleton object.

The overridden method `edit(FieldAccess)` first checks if the field access is declared as `static`. If that is the case, we return and get to the next field access, because `static` fields should be ignored (see chapter Heuristics). Further there is checked if the field type is primitive it should be ignored too.

Basically a field access can be either a writer or a reader.

```
1
2  public class MainClass
3  {
4      private Object objectA;
5      private Object objectB;
6
7      public static void main(String[] args)
8      {
9          MainClass main = new MainClass();
10         main.objectA = main.objectB;
11     }
12 }
```

Code segment 1: Class MainClass with field access at line 10

In this foobar example above we have the declared fields: `objectA` and `objectB`. Further we have two field accesses at line 10: `objectA` which is in this case the writer, since a value is assigned to this field, and the field `objectB` which is a reader access, because the field is only read and assigned to `objectA`.

Suppose we would instrument this example class: `MainClass`. Our implementation would save `objectA` as the KEY and the type of `objectB` as the VALUE to the object `StaticDataContainer.MultiMap`.
In detail the `StaticDataContainer` class has internally two distinct Hashmaps for saving the field-writers and the field-readers separately and for later merging them to the `MultiMap <KEYS, VALUES>`. This is clarified by the following diagram:

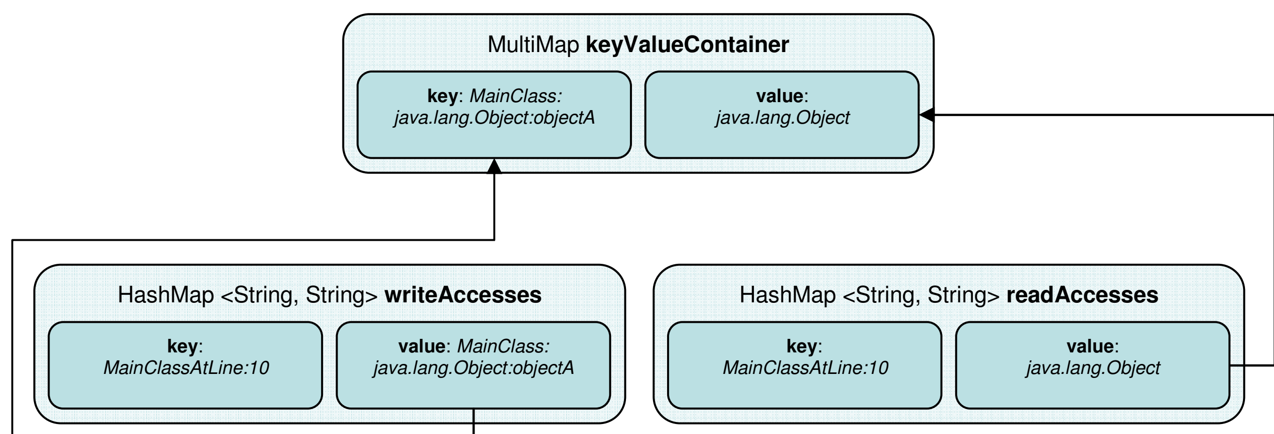

Figure 5: Merging internal hashmaps to MultiMap

A single Hashmap would not suffice, because we have not the context knowledge between field-writers and the field-readers inside the method `StaticFieldReader.edit(FieldAccess)`.
Every field-writer is saved in the Hashmap `writeAccesses`, where the key is the string concatenation of class name and the line number at which the access appears in the class file. The corresponding value is the field type at this LOC. In our example the `writeAccesses` content would look like the string as follows:

```
MainClassAtLine:11 - MainClass:Java.lang.Object:objectA
```

Where the key is `MainClassAtLine`:11 (enclosing class and line of occurrence concatenated) and the value `MainClass:Java.lang.Object:objectA.`

The field readers are saved in the second internally Hasmap `readAccesses`. This map contains the key like in the Hashmap `writeAccesses` and the value is the string-concatenation of the class name in which this field write occurs, the field type and the field name.

We need to save the accesses in separate Hashmaps, since we have not the context knowledge while parsing the external bytecode. For example we do not know if the correct read-access is the value type or the return type of read-accesses method.

The merge-process can differ in three cases:

1. merge `writeAccesses` with `readAccesses`
2. merge `writeAccesses` with `returnTypes`
3. merge `writeAccesses` with `casts`

1) `writeAccesses` with `readAccesses`
At the end of the `StaticDetector` we merge the two internal Hashmaps after the key and we get the `MultiMap` dump:

```
KEY: MainClass:Java.lang.Object:objectA
VALUE(S): Java.lang.Object
```

2) `writeAccesses` with `returnTypes`
Field accesses can appear in different ways like:

```
4  public class MainClass
5  {
6      private classA objectA = new classA();
7      private classB objectB = new classB();
8
9      public static void main(String[] args)
10     {
11         MainClass main = new MainClass();
12         main.objectA = main.objectB.getClassA();
13     }
14 }
```

Code segment 2: Class MainClass with field access. ObjectB with method call.

In this case the field `objectA` at line 12 has the value type `classA` and not `classB`, since the method `getClassA()` returns a class A instance. If we would only merge the writer- and reader access as in the later example we would get a wrong result. So we implemented this case by instrumenting the method calls in a class. This happens via the overridden method `edit(MethodCall)`. Inside this method we evaluate the return type of the method and save the type to the `StaticDataContainer` inside the `HashMap` called `returnValues`. The key of the `HashMap` is the location of the occurrence of the method call and the value is the return type of the called method.

The format is:

```
HashMap<[EnclosingClassName]AtLine[LineNumberOfTheCall],
          [return-type of the called Method]>
```

So in this example we get the merged `MultiMap` dump:

```
KEY: MainClass:classA:objectA
VALUE(S): classA
```

So if we have a method call we take the method-return-type as the “field reader” (here `classA`) and not the read field type itself (here `classB`).

Another access-example via method call of the project Apache Commons JXPath:

```
NodePointer org.apache.commons.jxpath.ri.axes.AncestorContext.currentNodePointer
77      public boolean nextNode() {   EvalContext org.apache.commons.jxpath.ri.EvalContext.parentContext
78          if (!setStarted) {
79              setStarted = true;    parent context
80              currentNodePointer = parentContext.getCurrentNodePointer();
81              if (includeSelf && currentNodePointer.testNode(nodeTest)) {
82                  position++;          NodePointer org.apache.commons.jxpath.ri.EvalContext.getCurrentNodePointer()
83                  return true;         Returns the current context node. Undefined before the beginning of the iteration.
84              }                        Returns:
85          }                                NodePoiner
```

Code segment 3: Project JXPath - Method AncestorContext.nextNode

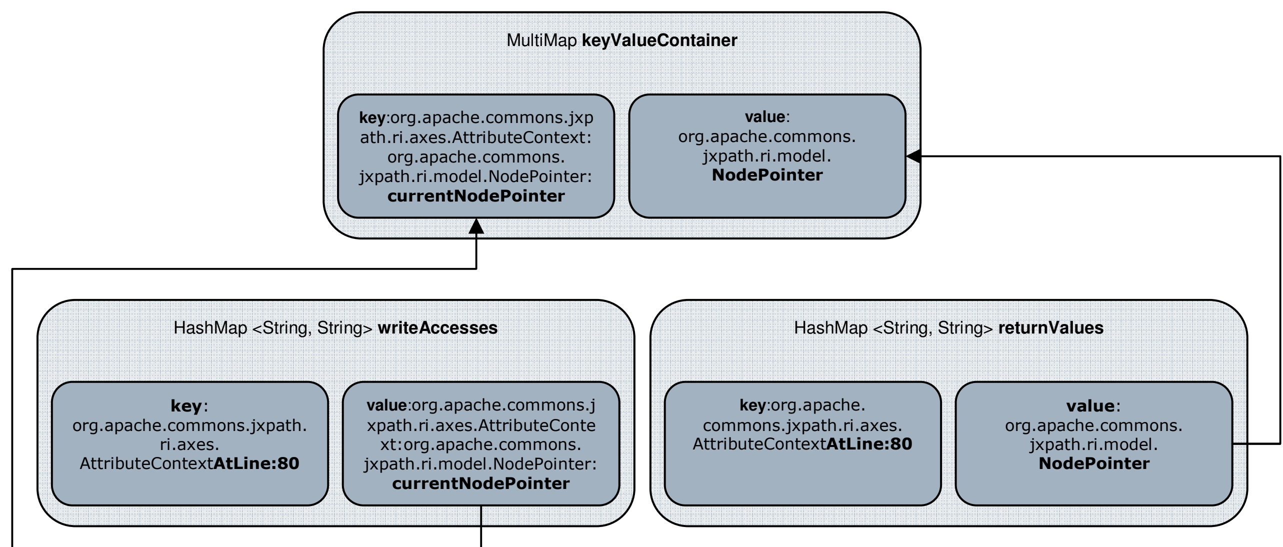

Figure 6: Project JXPath - The value of the writeAccesses Hashmap is merged with the value of HashMap returnValues, since both have the same key and occur therefore at the same LOC.

But as you can imagine we can have multiple method calls at one line:

```
main.objectA = main.objectB.getClassA().getClassC();
```

Since the method `edit (MethodCall)` of the `Javassist.ExprEditor` is called for each method call from left to right we can simply override the method return types from left to right. That means we get the return type of the method `getClassC()` after the return-type of the method `getClassA()`.
In detail we get two method calls at the same line. Therefore the enclosing class and the position are the same for both calls. And since every call is mapped to its position, we have inside the HashMap `returnValues` twice the same key. And by the definition of a Java.HashMap inserting a key which already exists will override the previous key and its value.
Since `Javassist` does not offer an API which returns the method calls of a specific field we had to implement it as mentioned - over the position/location.

Exactly the same happens if we have public field accesses like:

```
main.objectA = main.objectB.fieldOfObjectB;
```

Here `fieldB` is a public field of the `main` class and the `fieldOfObjectB` is a public field of `objectB`. The return type of the field access reader would be the declared type of field: `fieldOfObjectB`.

But if we link method calls with the field over the line we could get the following problem:

Assume we have in the code of the external project a field access and a method call like:

```
If (IsTrue()) fieldA = fieldB;
```

In this case we would not get the right result, since the if-condition is at the same LOC as the field-access. The result is correct if they are at separate LOC like:

```
If (IsTrue())
     fieldA = fieldB;
```

3) `writeAccesses` with `casts`
Another case of field access could be with casting:
We assume we have a Class A and a subclass B.

```
A classA = new A();
B classB = (B) classA;     // downcasting
```

That case would give us a field `classB` with value type A, but here the value type is B, since `classA` is down casted to an instance of B.
In this case we check if for the field `classB` is a field value and a casting at the same LOC. If that is the case we merge the field `classB` with the value type of the casting class (here class B). So as a result we get the correct value type of `classB`, which is B.

As already mentioned the field-readers, field-writers, method calls and casts are saved individually to an internal HashMap.
These Hashmaps are merged differently before returned as the `MultiMap` result.
The different merging-processes are done inside the method `StaticDataContainer.mergeKeyAndValues ()`:

```
62      private void mergeKeyAndValues()
63      {
64          Set<String> set = writeAccesses.keySet();
65          for (String key : set)
66          {
67              String theKey = writeAccesses.get(key);
68
69              if (readAccesses.containsKey(key) && casts.containsKey(key))
70              {
71                  String value = casts.get(key);
72                  keyValueContainer.add(theKey, value);
73              }
74
75              else if (readAccesses.containsKey(key) && returnTypes.containsKey(key)
76                      && !casts.containsKey(key))
77              {
78                  String value = returnTypes.get(key);
79                  keyValueContainer.add(theKey, value);
80              }
81
82              else if (readAccesses.containsKey(key))
83              {
84                  String value = readAccesses.get(key);
85                  keyValueContainer.add(theKey, value);
86              }
87          }
88      }
89  }
```

Code segment 4: Static Detector - method StaticDataContainer.mergeKeyAndValues

In the method-body of `mergeKeyAndValues()` we iterate through the `writeAccesses` key set and check for each key if the same key exists inside the HashMap `readAccesses` (3rd if-condition in code segment 4). If that is true we merge the value of `writeAccesses` of this key with the value of `readAccesses` to the result buffer `keyValueContainer`. But if we have for a write-access at the same line a read-access and a cast, we merge the `writeAccesses` with the `casts` and not with the `readAccesses` (1st if-condition).
If we have a write-access with a read-access and a method call at the same line we merge the `writeAccesses` with the `returnTypes` of the method call (2nd if condition).

Note: Since Javassist offers no API which gives the value of a write-access we only merge field-write-accesses with field-read-accesses.
If we have a field access like:

```
fieldA = aLocalVariable;
```

Here the access would not appear in the static result.

2.3 *Dynamic detection*

The main idea of the Dynamic Detection implementation is to get all fields which are polymorphic at runtime. And by definition that is the case when a field has multiple types. That means the algorithm should collect all field accesses at runtime and write them to the output file.

2.3.1 Reflection

To implement the Dynamic Detector we use the `Javassist.tools.reflect` library. Reflection makes it possible to inspect classes, interfaces, methods ect. at compile time. It is also possible to initiate new objects, invoke methods and get/set field values using Java reflection.

Two main classes of this package `dynamicDetection` are important: the `ClassPool` (see chapter: Static detection) and the `ClassLoader`.

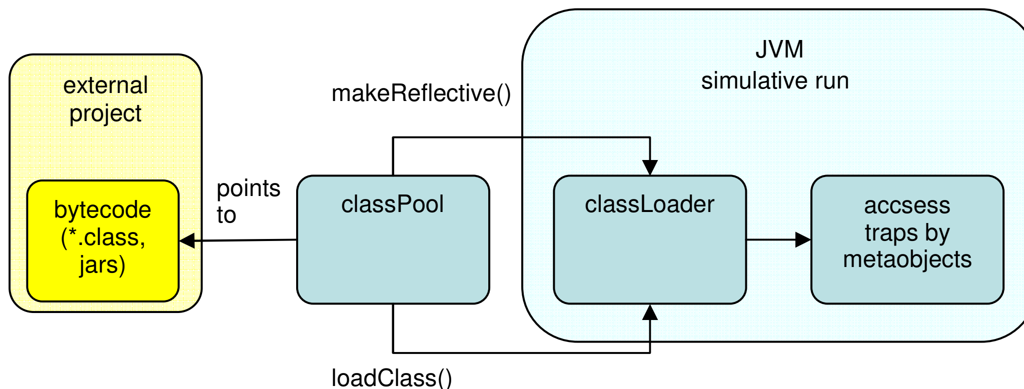

Figure 7: The classPool points to the bytecode root directory. The compile time classes are obtained by the classPool and loaded via methods loadClass() (for interfaces) and makeReflective to the classLoader. The classLoader runs the loaded classes in a simulative run (separate JVM). During the simulative runtime all reflective classes throw their access traps with help of their metaobject. The trapped accesses are saved to the MultiMap container.

`Javassist.tools.reflect.Loader`:
As mentioned the `Loader` enables to intercept the class loading mechanism and modify the loaded class. The `Loader` offers the methods `makeReflective()` and `loadClass()`. To get all field accesses of a class we have to make the class reflective. Since interfaces do not contain field accesses they do not have to be made reflective. We can load them directly to the loader via the method `loadClass (CtClassName)`. And in `Javassist.reflection` interfaces can not be made reflective at all. The `Loader` needs the paths received from the `ClassPool` for the localisation of the classes you attend to load.
So if we want to detect all field accesses of an external project we have to load all interfaces first and make all other classes reflective. Once this is done, the `Loader` runs the project in a simulative run and therefore in a separate JVM. We call this run “simulative”, since it is not the real run itself but one in a separate JVM.

The class `DynamicDetector` holds the fields: `ClassPool:pool` and `Loader:classLoader`. So in the constructor we have to initialize and link them to each other. The `ClassPool` pool contains the class path to the binary directory of the external project. So it points to the bytecode source directory of the external project we want to analyze. Then the pool is set to the `Loader` for having a reference to the `ClassPool` which is needed to find a class if we want to load it to the `Loader`.
Further we have to delegate the loading of the singleton class `DynamicDataContainer` to the `classLoader`. Otherwise a new instance of the `dynamicDataContainer` would be instantiated in the simulative run, since the instance is not in the scope of the simulative run. In other words the objects created in my program do not exist in the simulative run so we have to explicitly delegate the loading of the class we want to access at the simulative run.
And to gather the data created by the `MetaClass` we have to ensure that every `MetaClass` saves its field access traps in the same container, which is the reason for using the Singleton Pattern.

The main steps of the detector implementation are the following:

1) make all classes reflective
2) collect field accesses at simulative runtime

2.3.2 Make classes reflective

So in the first step every class of the external project has to be made reflective. A `Metaobject` is created for every object at the base level. A different reflective object is associated with a different `Metaobject`. During the simulative run every `Metaobject` intercepts field accesses on the reflective object at the base-level. To implement the behaviour of a field call we can derive from the `Metaobject` class. In our case, only the field access traps are of interest.
But there the first problem occurs. The `Metaobject` only intercepts field accesses if the fields are declared as `public`, field accesses with modifiers: `private`, `protected` and `package` are ignored. But the goal is to have field-access-traps at all fields, regardless of which modifier type they have. To reach that we have to set all non-`public` fields to `public` for getting all field-access-traps during the simulative run. But to do so, we have to detect and rename field names which are duplicated in a project. This is done by the class `ModifierRewriter`. For example a class A and its subclass B can have both a private field with the same name. After just setting the modifiers of these two fields to `public` the fields would be shared, which has to be avoided by any circumstances. So in our implementation duplicated fields are detected, but not renamed. The user gets informed about the duplication of fields, but the fields are not renamed to avoid sharing of fields. The reason lies in `Javassist`. The method `CtField.setName(String newName)` only renames the declared field name and not all occurrences of that field name in the whole class.
To detect the duplicated field names we have implemented the class `FieldNameCollector`, which collects via the overridden method `edit(fieldAccesss f)` all field names to an internal `MultiMap`. And as already mentioned the object `MultiMap` is able to detect duplicates.

By Javassist convention all classes we would like to make reflective has be made reflective in a specific order, otherwise an Exception would occur. We have to ensure that first the parent class of a class is made reflective before the class itself. To implement that we use a field called classNames, which is an Array of the object ClassPoolEntity. The ClassPoolEntity class is a helper class which binds a class name (string) to a load state (boolean).
The algorithm of the process is implemented as follows. First we load all classes whose parent class is Java.lang.Object. In other words, if the parent class is not in the class list we load it. If the parent class of a class is loaded, the class itself is loaded. This process is done till all classes are loaded/made reflective.

2.3.3 Collect accesses at runtime

Now every class of the external project has now a MetaClass. During the simulative run every field write is trapped by the MetaClass which holds this access. All the field writes are saved in the delegated singleton container DynamicDataContainer. The field reads are not for interest, since we can evaluate the accesses value type by the DynamicDataContainer and the field value is known in the field write method of the metaobject. So we do not have to merge field writers with readers as in the Static Detection case.

3 Sources

Javassist	http://www.csg.is.titech.ac.jp/~chiba/javassist/
Javassist Tutorial	http://www.csg.is.titech.ac.jp/~chiba/javassist/tutorial/tutorial.html
Javassist IBM Dokuments	http://www.ibm.com/developerworks/java/library/j-dyn0916/index.html
Javassist API	http://www.csg.is.titech.ac.jp/~chiba/javassist/html/
	Project at GitHub
Polymorphism Detector	https://github.com/mmorelli/PolymorphismDetection
External Projects	https://github.com/mmorelli/External-Projects