markfactory: Translation-based automatic exam evaluation for mass education

Boldizsár Németh
nboldi@elte.hu

Máté Tejfel
matej@elte.hu

ELTE Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary

Abstract

Mass education is an efficient way to train large number of students, and is applied on many universities across the world. The teaching process consists of many stages from lectures to evaluation. Automatic evaluation of exam results for courses on programming languages can be difficult when the evaluation has to assign scores for incomplete solutions.

This paper presents a method designed to let the educator easily write robust tests for evaluating student assignments. The evaluation code is transformed by a source-to-source transformation to use reflection when accessing classes written by the student. This enables to gracefully handle student errors, provide meaningful messages and estimate the scores for imperfect solutions. The transformation is implemented as an Eclipse plugin, and enables to automatically generate the automatic tester as the test code is modified. The plugin also generates a handout test, that can be used by students to test their solution on their own machines.

The plugin is used since 2014 for automatic evaluation of assignments on a beginner java course attended by approximately 250 students each semester.

1 Introduction

These days computer science education goes toward producing even greater numbers of students. This is understandable since the demand for programmers is greater than the supply. Online courses and universities thrive, and traditional universities work with large number of students. [1]

In our university it is common to have 12-14 classes, 20 student each for courses that are mandatory for all specializations. Having this many students with multiple assignments and exams results in a large number of programming solutions that have to be tested and evaluated. When only correct solutions are accepted we can write automatic tests for evaluating the solutions, but that is not possible when partial solutions must also be evaluated.

In: E. Vatai (ed.): Proceedings of the 11th Joint Conference on Mathematics and Computer Science, Eger, Hungary, 20th – 22nd of May, 2016, published at http://ceur-ws.org

2 Requirements

We were searching for methods to automatically test partial solutions for Java exams. The students are required to hand in solutions that can be compiled in itself (without test code). Otherwise no automatic testing is possible.

Our requirements for an evaluation method are the following:

I. The evaluation method must be able to evaluate solutions that are not complete (classes, methods, etc. might be missing).

II. We must handle exceptions that are raised from the solution methods, otherwise the evaluation would stop on the first exception.

III. We must handle exceptions that are raised from the test code caused by incorrect behavior of the test code. (We can't assume that the test code handles all cases perfectly.)

IV. The writer of the test should be able to assign numerical values to different exercises and these scores must be automatically given and summed when the test is run.

V. Writing automatic tests should not be more demanding than writing unit tests for the functionality of the solution.

VI. The test code must be compiled without the student solution or the sample solution.

VII. When there is an error or missing functionality in the solution it should be reported with a customized error message.

VIII. We should be able to give the students a hand-out that is understandable to them (no high-level Java concepts) and which they can use to validate their own solutions.

IX. Solving tests that are automatically evaluated should not increase the work that had to be done by the students.

3 Testing methods

We tried multiple methods of automatically evaluating partial solutions.

3.1 Pre-compiling the test

Thanks to the architecture of the Java Virtual Machine, classes are loaded on-demand. It is possible to compile the exam test with the sample solution and compile the students solution and then run the test that will load the students compiled solution.

This solution does not address the problem of exceptions raised during the testing nor does it handle the case when some functionality is missing from the student solution. In that case a rather cryptic error will be raised that might not be evident to the students.

3.2 Bigger problem granularity

Another possibility is to divide the exam into parts that are self-contained. In Java, a class is a self-contained unit of the program, that can be compiled in itself. In this case, the tests can check if these self-contained units exist and are implemented correctly. Being unable to compile a test means that the given part of the solution is not correctly implemented.

While the solution is good for catching exceptions and missing functionality, it does increase the work of the student (have to separate the solution into self-contained parts), and also increase the work of the instructor, who is preparing the test.

3.3 Automatic evaluation using reflection

The idea here is to use the Java Reflection API [2] to test the students solution. The reflection API allows runtime checking the existence of classes, methods, etc. as well as using them. The Reflection API can be used effectively with the annotations to mark certain program elements. By using reflections it is easy to catch exceptions in the invoked code, to compile the test once and run it for every student solution.

However using the reflection API is not simple. Writing a test that uses reflection to test the solution will be much more complicated that one that is not using that API. And more importantly, the resulting test cannot be given out to students, because they are not required to understand the Reflection API.

4 Automatic conversion of the test

Of all the possible solutions for the automatic evaluation of exam solutions, the last was the most successful, however it lacks a few important properties: It cannot catch exceptions that arise in the test code, it cannot be written easily and it cannot be given out to students.

To solve this problem we designed a method to automatically generate the test that uses Reflection from the normal java test code. We planned a mapping of Java program elements into a subset of elements that are using the Reflection API, only load a class if it is needed and does it in a checked fashion where the lack of a solution class can be handled gracefully.

We use three annotations to control the transformation of the program.

- The `@ExamTest` marks the class that will be used to test the students solution. The whole transformation is only done on classes that are marked with this annotation.
- The `@ExamExercise` marks a method that is used to test a given functionality of the students solution. The body of these methods must be protected so every exception can be caught before it stops the test.
- The `@TestSolution` annotation marks the classes that belong to the sample solution for the exam. The transformation must know that because they will not be present when the test is run, so every reference to them must be checked.

Finally, by generating a version of the tests where all the annotations and references to the translation API are removed and all the actual test code is commented out, we have a version of the test that can be given out to the students and does exactly what the automated version does.

Using automatically generated reflection code to evaluate the solutions satisfies all requirements that are defined above.

5 Mapping of Java elements

This section describes how different Java program elements are mapped into a subset of Java program elements that avoid early class loading and use the solution classes in a checked manner.

5.1 Imports

The transformation should remove all imports that are importing test solution classes. This is necessary to be able to compile the test independently of the sample solution.

5.2 Class declarations

For practical reasons, classes that are marked with `@ExamTest` will get a direct superclass that adds a few utility methods to the class. These utility methods will be used by the code that is generated by the transformation. This way we can prevent the use of the methods of the auto-tester in the pure test code, while enable it in the transformed code. However as a result, test classes must not subclass anything but `Object`.

5.3 Method declarations

Method declarations that are marked with `@ExamExercise` will be protected from any kind of exception raised while executing their body. The idea is that the instructor should write exercise test methods that test the given feature and at the end they increase the number of points earned by the student. This way if an exception occurs while executing the body of the method the cause of the exception can be printed out, and points will not be earned since the execution cannot continue to the end of the method body. To do this, the body of each exercise test method must be wrapped in a try-catch block that simply prints out the exception if one is raised.

5.4 Method and constructor calls

Method calls on objects that belong to a test solution are replaced by calls using reflection. These consist of first fetching of the correct method (possibly dealing with overloading), and invoking it by giving all arguments and calling `Method.invoke` and casting the result to the expected type. Constructor calls are similarly transformed when they construct object of a test solution class.

5.5 Array creation

When creating arrays we must take care, because arrays once they are created with a certain element type are usually not interchangeable. If we would simply create arrays of `Object` elements each time an array of `A` elements (a solution class) is created, and pass it to a method (with reflection) that takes an array of `A` elements, it would fail, since an array of `Object` elements cannot be cast to an array of `A` elements.

So for array creation (whether an empty array created with the given dimensions or an array of the given elements is created) the generated code will contain a call to the built-in method `Array.newInstance` that takes the runtime representation of a class and a size to construct an array with the given element type and size.

5.6 Field accesses

Getting and setting fields should also use the reflection API, by the `Field.get` and `Field.set` methods. In case of field sets the whole assignment expression is transformed, for example `a.fld = expr` becomes `fldObj.set(a, expr)`. For composite operators the left operand is repeated on the right-hand side, so `a.fld += expr` becomes `fldObj.set(a, fldObj.get(a)+expr)`.

5.7 Type literals

Type literals (for example `A.class`) are replaced with checked class loading when the value of the type literal refers a class that is a test solution.

5.8 Type casts and `instanceof` expressions

Existing type casts in the test code will not be replaced (except for the specified class), but new ones will be introduced at multiple points, described above. The `instanceof` expressions are replaced by calls for `Class.isInstance` that are the reflection equivalents of them.

5.9 Types

The Java Virtual Machine usually loads a class when it is executing an instruction that references the given type. However when in class `A` there is a static reference to class `B`, loading class `A` causes to load class `B` as well. Static references are static attributes, signatures of static functions and static method calls. [2]

When transforming the types that occur in the test, our goal is to delay the loading of the student test solution until the test actually creates such an object or invokes a static method. This requires of removing all static references. The generated test should must be able to compile without the sample solution.

The transformation rules are the following:

- Any type that is marked with `@TestSolution` should be replaced by its closest ancestor that is not marked with the annotation.
- In array types the element type should be transformed.
- In parameterized types, each type argument should be transformed.
- In wildcard types the bounds must be transformed.

6 Implementation

The evaluation method is implemented as the markfactory Eclipse [3] plugin. It is integrated into the build toolchain, so the test file that uses reflection is generated every time the original test is saved. If the plugin is enabled for a given project it also adds the library needed for automated testing to the project classpath.

The JDT [5] library was used to analyze and transform the abstract syntax tree (AST) of the program. We implemented the transformation with an `ASTVisitor` that provides us methods to visit all the AST nodes in the syntax tree both top-down and bottom-up order. We designed a wrapper library around the JDT, because building elements of the syntax tree using the simple JDT API is both wordy and error-prone.

The plugin generates the automatic tester into a new project to prevent name collisions. It exports the automatic tester into a jar file that can be easily deployed. The plugin also generates the handout version of the tester that does not contain any reference to markfactory classes and can be compiled without markfactory. In the handout the imports of the solution classes and method bodies are commented out. This way the student can immediately compile it and later can comment out the parts of the test for which she already implemented the solution.

7 Example

In this section we present a simple exam test and how it is transformed to use the Reflection API. The example is a simplified version of an actual exam. The exercise was to implement some parts of a card game.

7.1 Test code

Original definition of the test class:

```
import hu.elte.markfactory.annotations.ExamExercise;
import hu.elte.markfactory.annotations.ExamTest;
import uno.Color;
import uno.card.NumberCard;

@ExamTest
public class Tester {

  private static int points = 0;

  private static NumberCard nc1;
  private static NumberCard nc2;

  public static void main(String[] args) throws IOException {
    testNumberCard(5);
    System.out.println("\n\nArchived score: " + points);
  }

  @ExamExercise
  private static void testNumberCard(int p) {
    nc1 = new NumberCard(Color.BLUE, 3);
    nc2 = new NumberCard(Color.BLUE, 8);
    assertTrue("Putting a BLUE 3 on a BLUE 8 should be possible", nc1.canPlaceOn(nc2));
    points += p;
  }
}
```

Definition of the solution class `Color`:

```
package uno;
import hu.elte.markfactory.annotations.TestSolution;

@TestSolution
public enum Color {
  GREEN, BLUE, YELLOW, RED
}
```

Definition of the solution class `NumberCard`:

```
package uno.card;

import hu.elte.markfactory.annotations.TestSolution;
import uno.Color;
import uno.Game;

@TestSolution
public class NumberCard extends ColorCard {
  // ...
}
```

7.2 Generated code

```
import hu.elte.markfactory.annotations.ExamExercise;
import hu.elte.markfactory.annotations.ExamTest;

@ExamTest
public class Tester extends hu.elte.markfactory.testbase.ReflectionTester {
```

Imports that reference the solution classes have been removed. The class now subclasses the utility class `ReflectionTester`.

```
  private static int points = 0;

  private static Object nc1;
  private static Object nc2;
```

The types of `nc1` and `nc2` have been replaced with `Object`, since `NumberCard` is a solution class and it has no non-solution ancestor other than `Object`.

```
  public static void main(String[] args) throws IOException {
    testNumberCard(5);
    System.out.println("\n\nArchived score: " + points);
  }

  @ExamExercise
  private static void testNumberCard(int p) {
```

The `testNumberCard` method is an `@ExamExercise` so it is wrapped in a try-catch block to catch all exceptions without stopping the testing process. In such a case however, `points` will not be incremented. As a design decision for this test, we collected the numbers of points in the `main` method.

```
    try {
      nc1 = ((Object) construct("uno.card.NumberCard",
                                new java.lang.String[] { "uno.Color", "int" },
                                new java.lang.Object[]
                                  { ((Enum) staticFieldValue("uno.Color", "BLUE")), 3 }));
      nc2 = ((Object) construct("uno.card.NumberCard",
                                new java.lang.String[] { "uno.Color", "int" },
                                new java.lang.Object[]
                                  { ((Enum) staticFieldValue("uno.Color", "BLUE")), 8 }));
```

Since `NumberCard` is a solution class, constructor calls of it have to be replaced with reflection code. The `ReflectionTester.construct` method is capable of constructing the given object type if the types and values of the arguments have been given. The types of the arguments are passed as strings containing the qualified name of the type. Since `Color` is an enumeration and a solution class, its casted to `Enum` (the base class for all enumerations in java). The `BLUE` instance of the enumeration `Color` is technically a static field in java, so the `ReflectionTester.staticFieldValue` method will be used that gets the value of a static field using reflection.

```
      assertTrue( "Putting a BLUE 3 on a BLUE 8 should be possible",
                  ((boolean) call("canPlaceOn", nc1,
                                  new java.lang.String[] { "uno.card.Card" },
                                  new java.lang.Object[] { nc2 })));
```

The `canPlaceOn` is a method in a solution class, so it is invoked by the `ReflectionTester.call` method that is capable of invoking a given method with the given parameter types and values that are given similar to the `construct` method.

```
      points += p;
    } catch (hu.elte.markfactory.testbase.MissingProgramElementException e) {
      output(e);
      return;
```

If we detect that some program element is missing, the error message can be more specific.

```

    } catch (java.lang.Throwable e) {
      e.printStackTrace();
      return;
    }
```

If there is an exception during the execution of the method a generic exception message is displayed.

```
  }
}
```

8 State of the art

While we were searching for solutions we encountered several articles on Java code transformation. Code migration is a task similar to transforming a program for automatic evaluation, because it also transforms the basic elements of the program code (for example method calls) [7]. Some refactoring operations also systematically rewrite the Java syntax tree [9]. Processing annotations is already used in tools that modify Java source code [8].

We also found papers on code transformations for educational purposes. One of these papers present a method to transform source code into pseudo-code [6].

While code transformation tools are widely used, they are rarely applied to problems in education. We did not find existing solutions for the exact problem presented in this paper.

9 Results

In this paper we presented a solution for automatic evaluation of the students' (possible erroneous or partial) solutions for a Java exam. The instructor have to write a sample solution and a test for the solution and the plugin will generate a version of the test code that uses the Reflection API to test the student's code. The paper identifies 9 requirement for such a system and provides a method that fulfills all of these requirements.

The system is implemented as an Eclipse plugin, and the source code is available on GitHub [4].

The tool had been put to use in a beginner Java course with approximately 250 students each semester. It had a good testing coverage, only about one in five assignment had to be tested manually. It integrated well with the assignment management software that we used for the course. More detailed measurements about the efficiency of this tool is the topic of further research.

The cases where manual check is needed are easy to find. In these cases the automatic examination assigned 0 points to the student because she put the source code in the wrong folder or haven't named the packages accordingly to our guidelines.

The running time of the transformed checker is not important in our situation. Even if the evaluation of the student solution is slower by one or two orders of magnitude, it is still insignificant related to the time it takes to evaluate a solution by hand. In our setup, the evaluation of student submissions happened right after the user uploaded the solution, so only one solution had to be tested at a time and the system could handle hundreds of students.

Some issues are yet to be addressed. The most significant one is when in the test we would like to use a class that has a test solution superclass. This is useful for example in case of testing the presence of an abstract class. Correct mapping for such classes is the topic needs further inspection.

9.0.1 Acknowledgements

I would like to thank all the instructors at ELTE for their help for designing this method of automatic testing and their patience when the tool was not working correctly.

References

[1] Sue Clegg, Alison Hudson, John Steel The Emperor's New Clothes: Globalisation and e-learning in Higher Education British Journal of Sociology of Education. Vol. 24, Iss. 1, 2003

[2] Tim Lindhol, Frank Yellin, Gilad Bracha, Alex Buckley. The Java Virtual Machine Specification. Java SE 8 Edition

[3] The Eclipse IDE. http://www.eclipse.org/

[4] The MarkFactory project on GitHub. https://github.com/nboldi/markfactory

[5] JDT Core Component. https://eclipse.org/jdt/core/

[6] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura Learning to Generate Pseudo-code from Source Code using Statistical Machine Translation 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)

[7] Zakarea Alshara, Abdelhak Seriai, Chouki Tibermacine, Hinde Bouziane, Christophe Dony, Anas Shatnawi. (2015). Migrating large object-oriented Applications into component-based ones: instantiation and inheritance transformation. ACM SIGPLAN Notices. 51. 55-64. 10.1145/2936314.2814223.

[8] Pawlak, Renaud. (2005). Spoon: annotation-driven program transformation—the AOP case. 10.1145/1101560.1101566.

[9] Hannes Kegel, Friedrich Steimann. (2008). Systematically refactoring inheritance to delegation in Java. Proceedings - International Conference on Software Engineering. 431-440. 10.1145/1368088.1368147.