TTC'2015 Case: Refactoring Java Programs using Spoon

Gérard Paligot
gerard.paligot@inria.fr
Inria

Nicolas Petitprez
nicolas.petitprez@inria.fr
Inria

Martin Monperrus
martin.monperrus@univ-lille1.fr
University of Lille

Abstract

TTC'2015 is the 8th Transformation Tool Contest for users and developers of transformation tools. In this paper, we present the use of Spoon, an open-source library to transform and analyze Java source code for the code refactoring track of TTC'2015. We use Spoon to implement *pull-up-method* and *create super-class* refactorings. The implementation uses an unmodified revision of Spoon and is done in 125 lines.

1 Introduction

Spoon[7] is an open-source library that enables you to transform and analyze Java source code. Spoon provides a complete and fine-grained Java metamodel where any program element (classes, methods, fields, statements, expressions...) can be accessed both for reading and modification. Spoon takes as input source code and produces transformed source code ready to be compiled.

For now, Spoon has been used in many different contexts: program analysis and transformation in Java[6], automatic repair of buggy if conditions[4] or fault injection [3] but nobody has ever studied the use of Spoon for refactoring Java programs. To explore this new usage of Spoon, we have implemented the two types of refactoring asked by Transformation Tool Contest 2015 [2].

Our solution is publicly available on Github:
`https://github.com/GerardPaligot/ttc-competition/`

The paper reads as follows. Section 2 gives a description of the chosen case study. Section 3 present the chosen solution. Section 4 explains how we validate our solution. Section 5 give some discussions of design decisions and perspectives for our work. Section 6 concludes this paper.

2 Background

2.1 Case study

The chosen case study is "Object-oriented Refactoring of Java Programs using Graph Transformation" [5], it proposes two object-oriented program refactorings. It consists of implementing two refactorings, namely *pull-up-method* and *create super-class*.

First, we explain how *pull-up-method* works. Before the refactoring, the Java code must have methods with identical signatures (name and parameters) and equivalent behaviors. These methods are then moved to the superclass. After the refactoring, the method is a member of the superclass and deleted from the subclasses. This operation is depicted in Figure 1.

We consider the following conditions to apply the *pull-up-method* refactoring:

1. Each child class of class `ParentClass` has at least one common method signature with the corresponding method definitions having equivalent functionality [5].

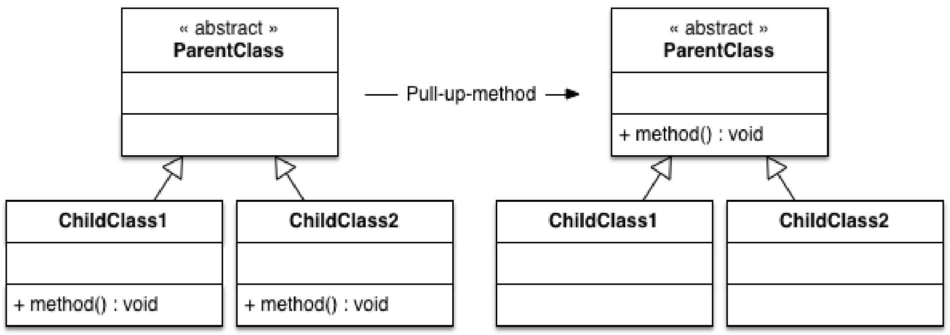

Figure 1: Illustration of a Pull-up-method

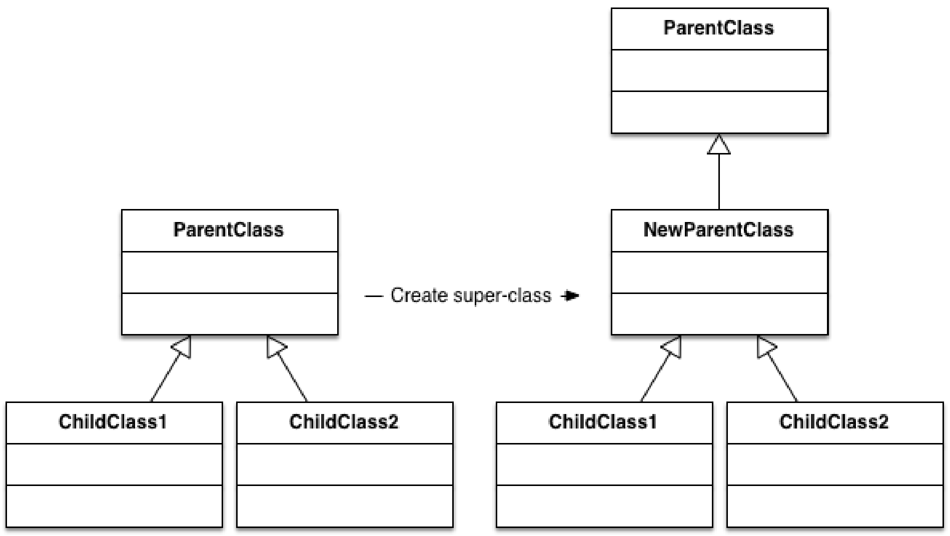

Figure 2: Illustration of a Create a super class

2. Each method in the child classes only accesses methods and fields accessible from `ParentClass`.

3. The `ParentClass` does not belong to a library and is editable. [5]

Second, we explains how the *create super-class* refactoring works. This kind of refactoring is useful when we have a set of classes with similar features. As a first step towards an improved program structure, a new common superclass of these classes is created. When we have subclasses with a parent class, it creates a new parent class and this new class extend the old one (the previous parent). This operation is depicted in the Figure 2.

In this case, we have one precondition: the classes are extending the same superclass. The precondition is always met is the default case since classes with no explicit inheritance in Java are all implementing `java.lang.Object`. [5]

This refactoring has the following post-conditions:

1. Each class has an inheritance to the new super class [5].

2. When the classes had an explicit inheritance relation to a superclass before the refactoring, their new superclass has an inheritance reference to the old super class [5].

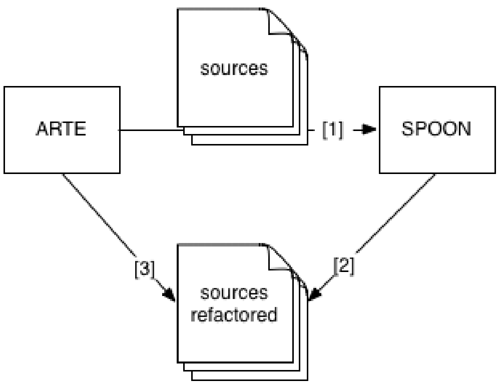

Figure 3: Transformation chain of the study case

2.2 Spoon

Spoon provides a Java abstract syntax tree (AST) designed to be understandable by developers. With this AST, developers can analyze or transform the source code. In our solution, we use two concepts of Spoon: `Factory` and `Query`.

`Factory` creates new elements or retrieves some specific elements.For instance, the factory is used to create the new super class for the *create super-class* refactoring.

`Query` makes complex queries on a AST. If you would like to retrieve all methods, fields, class or any elements of the meta-model, you execute a query to get them. This concept is used to retrieve all methods concerned by the *pull-up-method* refactoring.

2.3 Test infrastructure

ARTE is a Java program which executes test cases specified in a Domain Specific Language (DSL) for validating the solutions of the OO Refactoring Case Study of the Transformation Tool Contest 2015. A test case comprises a sequence of refactoring operations on a Java program as well as the expected results. The tests aim at checking the correct analysis of pre- and postconditions for refactorings and the execution of these refactorings [5].

This test framework defines specific command line arguments. When you execute the jar file, you launch a custom terminal where you execute these commands. From this terminal, you load your solution and execute the ARTE tests. If you execute all tests, it executes public tests and hidden tests. Input source code and assertions are public for public tests but only input sources are public for the hidden tests.

We give an overview of the transformation chain in Figure 3. First, we see than ARTE loads sources and gives them to our solution based on Spoon. Second, our implementation refactors the Fava source code given to print sources refactored. Third, ARTE uses sources refactored by our solution to check assertions on our results.

3 Presentation of the Solution

3.1 Pull-up-method

Algorithm 1 shows the pseudo-code of our implementation. In input, we have the method to be refactored and the superclass element where we should put the refactor method. These objects are given as parameter of the refactoring method. As output, a boolean tells whether the refactoring is done or not. Let's explain this algorithm step by step:

1. We check that the superclass given as parameter exists. If it doesn't exist, we are not allowed to pull up the method. The refactoring fails.

2. We retrieve all methods candidates for the refactoring with the same name and type parameters and we store them in a list named `candidates`.
3. For each candidate, we check that the superclass of the declaring class of the current candidate method exists. If this superclass doesn't exist, the refactoring fail.
4. For each candidate, we check that the body of the current candidate method does not try to access fields of the declaring class. If it tries, the refactoring fails.
5. For each candidate, we check that the body of the current candidate method does not try to access methods of the declaring class. If it tries, the refactoring fails.
6. We retrieve all subclasses of the superclass and for each subclass, we check that the method exists in it. If not, the refactoring fails.
7. When all previous conditions are passed, we apply the refactoring. For each candidate, we remove it from the declaring class and we set the method asked in the superclass.

```
Data: superclass element and method element
Result: true if the refactoring is done
if superclass doesn't exist then
    fail
end
candidates ← all methods candidates for refactoring;
foreach candidate in candidates do
    if superclass of candidate doesn't exist then
        fail
    end
    if body of candidate try to access fields of declaring class then
        fail
    end
    if body of candidate try to access methods of declaring class then
        fail
    end
end
foreach subclass in subClasses of superclass do
    if method to refactor isn't present in subclass then
        fail
    end
end
foreach candidate in candidates do
    Removes candidate method from declaring class;
end
Adds method in superclass;
```
Algorithm 1: Pull up methods in their superclass

3.2 Create super-class

Algorithm 2 shows the pseudo-code of our implementation of *create super-class*. As input, we have a set of children and a superclass element. These objects are given as parameter of the refactoring method. As output, a boolean tells whether the refactoring is done or not. Let's explain this algorithm step by step:

1. We check that the superclass does not already exist. If yes, the refactoring fails because we are not allowed to create a superclass on an existing class.
2. We create the new superclass from the superclass.

pub pum2 1	0,063 seconds
pub pum1 1 paper1	0,018 seconds
pub pum1 2	0,002 seconds
pub csc1 1	0,136 seconds
pub csc1 2	0,002 seconds
pub pum3 1	0,005 seconds
hidden csc1 1	0,003 seconds
hidden csc1 2	0,002 seconds
hidden pum1 1	0,003 seconds
hidden pum1 2	0,003 seconds
hidden csc2 1	0,003 seconds
hidden pum2 1	0,005 seconds
hidden pum2 2	0,003 seconds
hidden csc3 1a	0,009 seconds
hidden csc3 1	0,004 seconds

Table 1: Execution time measurements

3. We collect all super-classes of children and we check that there are all the same superclass. If yes, new superclass extends this superclass. Otherwise, the refactoring fails.
4. For each child in set of children, we set its superclass with the new superclass.

```
Data: set of children and superclass element
Result: true if the refactoring is done
if superclass already exists then
    fail
end
Create newsuperclass from superclass;
Set superclass of newsuperclass from superclasses of children;
foreach child in children do
    Set superclass of child with newsuperclass;
end
```

Algorithm 2: Creates and sets the new superclass for all children

3.3 Execution time measurements

When we execute a test in ARTE, we see in output the name of the test case, the executed refactoring, results of assertions and the execution time. Table 1 shows execution time measured by ARTE when we execute all tests (execution time measurements are different when we execute one by one).

We see that the performances are good. The worst execution time is the test case pub csc1 1. This test case applies the refactoring *create super-class* on an example with two child class and a super class for these subclasses.

3.4 Architecture

Our solution is available on Github [1]. It is a Maven project in Java 8 with only one "compile" dependency: spoon. The solution has 2 "provided" dependencies: TTCTestInterface and EMF. TTCTestInterface is a Jar file given by the case study and versioned in the project. We have created a local maven repository in the project to save all versions of TTCTestInterface jar file updated by organizers. TTCTestInterface contains an interface which returns objects with EMF objects, like `EList`. To manipulate objects like `EList`, we need the EMF dependency. Finally, there are 2 "test" dependencies: junit and mockito which are used to test our solution.

We generate the solution in a jar file with the maven command:

```
$ mvn clean assembly:assembly
```

This command compiles the project, launches all Junit test cases and builds the final jar file with dependencies in the target directory of the project. After that, this jar file is used on ARTE to launch all tests of this last tool. According to our experience, it isn't possible to integrate ARTE in the maven process because ARTE must be launched as command line.

All Junit tests are available in the folder `src/test/java` and correspond to public and hidden test cases given by organizers executed in ARTE. All Java source code used by ARTE has been copied in `src/test/java/resources` and used by test cases in `src/test/java/fr/inria`. So, when we compile the project with the command line given in the next section, we execute the same tests than the tests executed in ARTE.

The implementation asked of TTCTestInterface is SpoonTtc. This class retrieves the Java source code in the method `createProgramGraph` and builds the Spoon AST. This AST is used on methods to apply refactorings with 2 stages: First, refactoring methods check whether we must apply the refactoring. Second, refactoring methods apply the refactoring on the Spoon AST. The Java source code refactored is printed in the method `synchronizeChanges` in the source directory of the original Java program.

4 Validation

For *pull-up-method*, it has 9 tests corresponding to the ARTE public. There is on parameterized test class to launch 6 tests on all examples available in resources. For *create super-class*, it has 6 tests corresponding to the ARTE public and hidden tests and has a test class parameterized to launch 6 tests on examples.

For each test case, we make some assertions on the Spoon AST and the boolean result of the refactoring method. We make pre-conditions to know if the Spoon AST is in a correct state. We check that the refactoring method returns the expected boolean value. Finally, we make post-condition on the Spoon AST to know if the refactoring is applied or not, according to the boolean result of the refactoring method.

When we call refactoring methods, its parameters has a dependency to EMF. So, we add the mockito dependency to simulate these objects and test our implementation in a controlled environment. We build mocked objects and we add them in parameter of refactoring methods. For example, Listing 1 shows a mocked object given at the method `applyPullUpMethod`.

Listing 1 Test case for the pull-up-method

```
@Test
public void testPullUpMethod11() throws Exception {
  spoonTtc.createProgramGraph("./src/test/resources/paper-example01/");
  // Pre-asserts on the Spoon AST.
  assertTrue(spoonTtc.applyPullUpMethod(
    getPullUpRefactoringMocked(
      "example01.ParentClass", "foo", "java.lang.String", "int")));
  // Post-asserts on the Spoon AST.
}
```

This example calls the method `applyPullUpMethod` to apply the refactoring of the same name. As parameter, we give the mocked object. To build this last object, we give the super class where the method will be pulled up and the method concerned by the refactoring with type of its parameters. In this case, the refactoring is possible so we check than the result is `true`.

5 Discussions

To our opinion, Spoon was well suited for this case study. Its understandable AST and its capability to transform Java programs were appropriate. We realized the refactorings quickly and within a few lines. Implementing the refactoring case study with Spoon took 80 lines for *pull-up-method* and 21 lines for *create super-class*.

Spoon has no module to refactor Java source code. This case study was a great opportunity for us to start such a module. All contributions here will be integrated in Spoon in a next release and will be improved with some new refactorings in the future.

6 Conclusion

We have presented a solution for the for this edition of Transformation Tool Contest based on Spoon. It has validated the idea of using Spoon for implementing refactorings of Java source code.

References

[1] GitHub repository of our submission. `https://github.com/GerardPaligot/ttc-competition/tree/master/lib/fr/inria/TTCTestInterface/`.

[2] Transformation Tool Contest 2015. `http://www.transformation-tool-contest.eu/`.

[3] Benoit Cornu, Lionel Seinturier, and Martin Monperrus. Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions. *Information and Software Technology*, 57(0):66 – 76, 2015.

[4] Favio Demarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. Automatic Repair of Buggy If Conditions and Missing Preconditions with SMT. In *CSTVA'2014*, Hyderabad, India, 2014.

[5] Sven Peldszus Géza Kulcsàr and Malte Lochau. Case Study: Object-oriented Refactoring of Java Programs using Graph Transformation. 2015.

[6] Renaud Pawlak, Carlos Noguera, and Nicolas Petitprez. Spoon: Program Analysis and Transformation in Java. Research Report RR-5901, Inria, 2006.

[7] Spoon. Spoon project on GitHub. `https://github.com/INRIA/spoon`.