The SDMLib solution to the Java Refactoring case for TTC2015

Olaf Gunkel, Matthias Schmidt, Albert Zündorf
Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73, 34121 Kassel, Germany

olaf.gunkel|matthias.schmidt|zuendorf@cs.uni-kassel.de

The Solution is hosted under https://bitbucket.org/mschmidt987/java-refactoring-case-ttc-2015-solution-fg-se-uni-kassel

This paper describes the SDMLib solution to the Java Refactoring case for TTC2015 [2]. SDMLib provides a mechanism for generating an abstraction model of a provided java program. In addition, SDMLib provides code generation that transforms the whole model or parts of it into java code. Thus, for the Java Refactoring case we just added a *Refactorer* that reads a java project and transforms the program graph according to the intended refactorings. These transformations are collected and applied to the source code by the SDMLib generator afterwards.

1 Introduction

Two of our studentical assistants found this case very interesting, because they plan to realize a related case in their master thesis. Their idea is to find bad smells and other structures that should be replaced by a design pattern implementation. After the detection of such places, the replacement should be applied by an automatic refactoring. The implementation of the TTC 2015 refactoring case gave them the chance to have a look on implementing refactorings and estimate the complexity of such code replacement operations.
Furthermore, our team gives a lecture in Graph Engineering at the University of Kassel in which we teach master grade students about the theoretical definition of graphs and practical approaches of graph matching and transformation operations. In addition, we teach them to implement a graph matching algorithm to perform transformations on the previously implemented generic graph. So we are familiar with several graph transformation techniques and interested in tasks that can be solved with them.
In previous work we already addressed the problem of parsing and generating java source code. To solve this, we added some features to our tool SDMLib. It is able to represent parsed code into a class model, that holds enough information to generate updated code afterwards (without changing the present code where it is not needed). We expected that to be a benefit for us when solving this case.
To address the Java Refactoring case, we used the introduced parser of SDMLib to create the program graph before the refactoring. Then we have built a new component to realise the refactorings in the graph. This component uses property change mechanisms to record the changes of the program graph and refactors the source code by calling the SDMLib generator afterwards.

2 SDMLib support for source code abstraction and generation

Transforming java source code into an abstract model is a complex task that can be accomplished by using a powerful parser. To solve this, SDMLib provides a recursive descent parser that is able to analyze java source code files and create an abstract graph model. Using the parser is really easy due to the fact that, as shown in Listing 1, only the source folder and the package name (where the program lies that should be

Submitted to:
TTC 2015

abstracted) is required.

```
public void updateFromCode(String srcFolder, String packageName){...}
```

Listing 1: Signature of the method that calls the parser for java programs

After parsing the source code, SDMLib provides a model that contains all information required for the refactoring case. The parts of the model, which we use to solve the case, can be seen in Figure 1.

The SDMLib model provides nearly every information that we need for the case. The only missing information, which is still missing in the solution, is the *access*-assoziation of class *TMember* as shown in figure 2 of the case description[1]. Despite the fact that the whole model represents complex program structures, it is comfortable, easy to use and enabled us to fullfill the requirements of the given tasks rapidly.

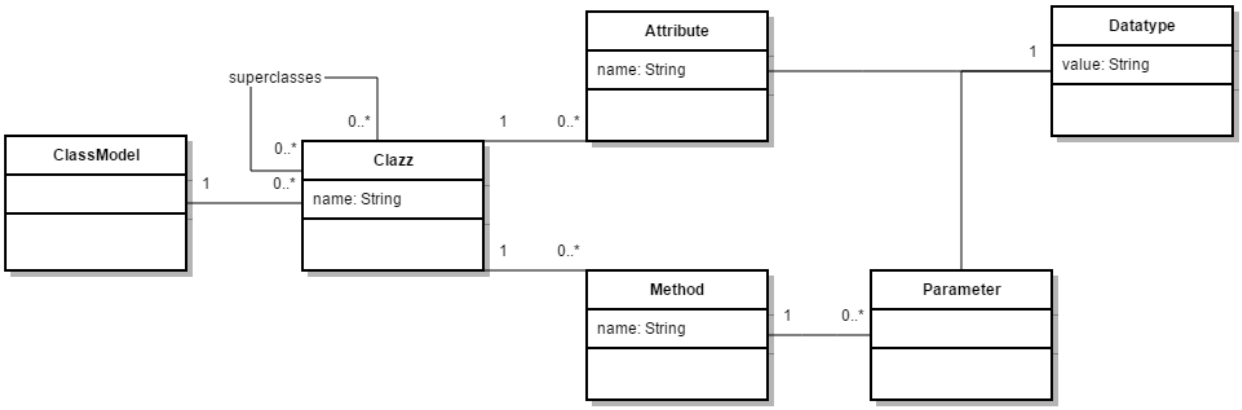

Figure 1: Cut of the source code abstraction model

To push our graph changes into the code, SDMLib supports us with its generator, that updates the parsed code. After creating a *ClassModel* by parsing a java project, every included class has its own parser instance, held by the *ClassModel*. The parsers are holding all relevant information about their class. For example, they have symbol tables in which, for every member, information about its position in the sourcecode are stored. By using this position information, its possible to extract, replace and insert parts of the sourcecode. Because of this relation, we can use the symbol table to delete, move or insert members in the source code. Listing 2 shows how to delete a member from the source file of a class. After replacing entries in the class, we set the boolean field *fileChanged* to *true* and commit the changes to the generating class *CGUtil*. Its *printFile(Parser)* Method writes the changes into the source code files.

```
1   SymTabEntry memberToDelSTE = clazzParser.getSymTabEntry(delMember);
2
3   clazzParser.replace(memberToDelSTE.getStartPos(),
4           memberToDelSTE.getEndPos()+1, "");
5
6   clazzParser.withFileChanged(true);
7
8   CGUtil.printFile(clazzParser);
```

Listing 2: How to push changes to the source code with SDMLib

3 Solving the Java refactoring case with SDMLib

Our solution covers the three major transformation steps (code to program graph, program graph refactoring and program graph to code) with support for create class-, pull up method-, pull up field- and extract superclass refactoring.
SDMLib already contains a mechanism to transform code into a program graph. So this part was quite easy to implement. The method *createModelFromSource* in Listing 3 shows how SDMLib can be used to generate a model out of given java source code. Just the path to the project is necessary.

```
1 public ClassModel createProgrammGraph(String pathToProject)
2 {
3
4    return refactorer.createModelFromSource(pathToProject);
5
6 }
```

Listing 3: Creating a object model from source code in a given package path

The resulted program graph now must be transformed according to the intended refactoring. Our algorithm is split into two parts here. The first part validates that the refactoring can be applied on the given object structure. For example a pull up method refactoring requires, that all child classes contain the method with the right signature. This requirement is checked for a valid match. The second step executes the graph transformation for the refactoring. Figure 2 shows an example situation for the pull up method refactoring. The method of the first child that should be pulled up gets his class relation changed to the parent. Furthermore we remove the matching methods of all other kids from the graph.
To complete the last step, we decided to add property change listeners to all relevant members of the object model. These are the methods, classes and attributes, because the refactorings cause changes to them. Our aim was to trace the changes. After the refactoring, the generator of SDMLib applies the traced transformations to the source code. For example our so called *ClazzSuperClassPropertyFileListener* reacts on changes of the inheritance field of a class. If a new superclass is set, this listener saves an object of the *ClazzSuperClazzPropertyFileChangeStep* Class in a *Queue*. This queue contains all events with their relevant information. To synchronize model and code, we execute all source code transformations according to the previous done model transformations. In this example, the generator changes the *extends* clauses of the affected classes or generates a new superclass.

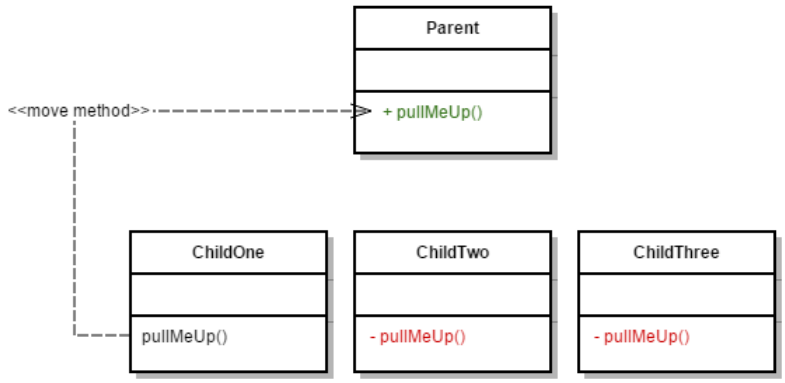

Figure 2: Example Graph Transformation for Pull Up Method Refactoring

Overall this case was made for us, because SDMLib already had many features to help us creating a program graph and updating the appropriate java source code. Especially the parser and the generator of SDMLib helped to complete these tasks. Furthermore the resulting program graph fullfilled all our needs for the refactorings.

4 Accomplished testcases

In Table 1 all execution times und the result of the given cases are presented. Except of one hidden case, our program succeeds in all tests. The one that fails contains a test where a method of two child classes should not be pulled up, because one of them is accessing a field, that the other one does not have. Our program fails here, because our tool does not analyse the semantic of method bodies. So there are no access edges in our program graph yet.
By writing additional tests, we made sure to cover many other cases. The pull up refactoring ensures that the parent class is available. Furthermore we detect whether the pull up method or field is already defined in it, that it has childs and that all childs own the method or field with the right set of parameters. The create superclass refactorer also filters out the corner cases. It ensures that the superclass is not already existing. In addition, the refactoring fails with a response if not all chosen classes have the same superclass.
In Table 1, all execution times und the result of the given cases are presented. Except of one hidden case, our program succeeds in all tests. The one that fails contains a test where a method of two child classes should not be pulled up, because one of them is accessing a field, that the other one do not have. Our program fails here, because our tool does not analyse the semantic of method bodies. So there are no access edges in our program graph.
By writing additional test cases, we make sure to cover many other cases. The pull up refactoring ensures that the parent class is available. Furthermore we detect whether the pull up method or field is already defined in it, that it has childs and that all childs own the method or field with the right set of parameters. The create superclass refactorer also filters out the corner cases. It ensures that the superclass is not already existing. In addition, the refactoring fails with a response if not all chosen classes have the same superclass.

Case	Time(s)	Result
pub_ pum3_ 1	0	SUCCESS
hidden_ csc3_ 1a	0,003	SUCCESS
hidden_ csc1_ 2	0,001	SUCCESS
pub_ pum1_ 1_ paper1	0,005	SUCCESS
hidden_ csc1_ 1	0,007	SUCCESS
pub_ csc1_ 2	0,003	SUCCESS
hidden_ pum1_ 2	0,001	SUCCESS
pub_ csc1_ 1	0,005	SUCCESS
hidden_ pum1_ 1	0,002	FAILURE
hidden_ csc2_ 1	0,002	SUCCESS
pub_ pum1_ 2	0	SUCCESS
hidden_ pum2_ 2	0	SUCCESS
hidden_ pum2_ 1	0,002	SUCCESS
hidden_ csc3_ 1	0,006	SUCCESS
pub_ pum2_ 1	0,001	SUCCESS

Table 1: Execution time of all given test cases

References

[1] M. L. Géza Kulcsár, Sven Peldszus. Case Study: Object-oriented Refactoring of Java Programs using Graph Transformation.

[2] Object-oriented Refactoring of Java Programs using Graph Transformation (TTC'2015). https://github.com/Echtzeitsysteme/java-refactoring-ttc, 2015.