Case Study: Object-oriented Refactoring of Java Programs using Graph Transformation

Géza Kulcsár, Sven Peldszus, and Malte Lochau

TU Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt
{geza.kulcsar@es|sven.peldszus@stud|malte.lochau@es}.tu-darmstadt.de

Abstract. In this case study for the transformation tool contest (TTC), we propose to implement object-oriented program refactorings using graph transformation. The case study possesses two major challenges to be solved by solution candidates: (1) bi-directional synchronization between source/target program source code and abstract graph-based program representations, and (2) program graph transformation rules for graph-based program refactorings. To keep those challenges manageable within the scope of TTC, we limit our considerations to a restricted sub-language of *Java 1.4*, comprising major object-oriented building blocks, i.e., declarations of classes and inheritance, methods with overloading/overriding, as well as field accesses. In particular, we require solutions to implement at least two prominent refactorings, namely *pull-up-method* and *create super-class*, thus demanding both of the aforementioned challenges to be equally tackled. Our evaluation framework for candidate solutions consists of collections of sample programs comprising both positive and negative cases, as well as an automated before-after testing procedure.

1 Introduction

The fact that software systems are repeatedly changed, updated, refined, and enhanced – in other words, *evolve* throughout their entire life-cycle – in order to be improved, or to meet ever-changing requirements lies at the very core of software engineering. This continuous evolution also means that systems are prone to internal decay and the incremental nature of these changes may cause software systems to arrive at inconsistent, or even unwanted states. This gradual erosion of the original design might also make the code base incomprehensible.

The *object-oriented* (OO) programming paradigm has been an important step towards software system modularity. The OO programming paradigm allows for enforcing essential program/data structures, e.g., by means of design patterns. In this regard, Opdyke was the first to propose *refactoring* as a countermeasure for those negative consequences of software evolution, by defining 23 possible refactorings towards patterns in a human-readable form [3]. Thereupon, Fowler expanded the catalog of refactorings (with retaining its informal nature) in his seminal work [1], which serves as a *de facto* standard by now.

Program refactoring aims at high-level restructuring of OO programs at class/method level to fit previously defined structural patterns without altering its observable behavior. Most recent implementations usually rely on ad-hoc program transformations directly applied to the AST (Abstract Syntax Tree). A promising alternative to tackle the challenge of identifying those (possibly concealed) program parts being subject to structural improvements is *graph-based refactoring*. Here, the program is transformed into a abstract and custom-tailored program graph representation that (i) only contains relevant program elements, and (ii) making explicit static semantic cross-AST dependencies, being crucial to reason about refactorings. Nevertheless, certain language constructs of more sophisticated programming languages pose severe challenges for a correct execution of refactorings, especially for detecting refactoring possibilities and for verifying their feasibility. As a consequence, the correct specification and execution of refactorings for OO languages like Java have been extensively studied for a long time in the literature and, therefore, can not serve as scope for a TTC case study to their full extent. In our case study we, therefore, propose the challenge of graph-based refactorings to be considered on a restricted sub-language of Java 1.4, further being limited to core OO constructs being of particular interest for the respective structural patterns.

In particular, we require a candidate solution for our case study to do the following:

1. The solution takes as input a program (source code) written in Java, conforming to the Java 1.4 major version with some additional restrictions (detailed further in Section 2.1).
2. The solution transforms the input program into an abstract representation, i.e., a (Typed) Program Graph (PG), which has a predefined format (i.e., conforms to formally specified meta-model that is part of the case study).
3. The solution checks whether a given refactoring operation can be performed on the PG; in case the program transformation defined by the refactoring operation is syntactically applicable and meets semantic preconditions for behavior preservation, the PG is transformed to a refactored PG, and an appropriate message is shown otherwise.
4. If the refactoring was possible, the solution synchronizes the refactored PG and the Java code such that those program parts unaffected by the transformation remain unchanged.
5. The solution produces a refactored Java program (source code) as output, whose observable behavior is identical to the input program.

This procedure comprises two major challenges:

I **Bidirectional and incremental synchronization of the Java source code and the PG.** This dimension of the case study requires special attention when it comes to maintaining the correlation between different kinds of program representation (textual vs. graphical) and different abstraction levels. Additionally, the code and the graph representation differ significantly w.r.t. the type of information that is displayed explicitly, concerning, e.g.,

method calls, field accesses, overloading, overriding etc. As the (forward) transformation of a given Java program into a corresponding PG representation necessarily comes with loss of information, the backward transformation of (re-)building behavior-preserving Java code from the refactored PG cannot be totally independent from the forward transformation – a correct solution for this case study has to provide some means of restoring those parts of the input program which are not mapped to, or reflected in the PG.

II **Program refactoring by PG transformation.** In our case study, refactoring operations are represented as rules consisting of a left-hand side and a right-hand side as usual. The left-hand side contains the elements which have to be present in the input and whose images in the input will be replaced by a copy of the right-hand side if the rule is applied. Therefore, the actual program refactoring part of our case study involves in any case (i) the specification of the refactoring rules are based on refactoring operations given in a semi-formal way, (ii) pattern matching (potentially including forbidden patterns, recursive path expressions and other advanced techniques) to find occurrences of the pattern to be refactored in the input program and (iii) a capability of transforming the PG in order to arrive at the refactored state. Note that the classical approach to program refactoring (which is used here) never goes deeper into program structure and semantics than high-level OO building blocks, namely classes, methods and field declarations; the declarative rewriting of more fine-grained program elements such as statements and expressions within method bodies is definitely out of scope of our case study for TTC.

We demand two exemplary refactoring operations to tackle when solving this case study. The first one, `Pull Up Method` is a classical refactoring operation – our specification follows that of [1]. `Pull Up Method` addresses Challenge II to a greater extent. The second one, `Create Superclass` is also inspired by the literature, but has been simplified for TTC. It can be considered as a first step towards factor out common elements shared by sibling classes into a fresh superclass. In contrast to `Pull Up Method`, new elements have to be created and appended to the PG. `Create Superclass`, therefore, comes with more difficulties regarding Challenge I. Nevertheless, the solution developer will face both challenges while implementing both refactoring operations.

In the following, we give a detailed description of the case study to be solved by specifying the constituting artifacts, (meta-)models and transformations in Section 2. The two sample refactoring operations mentioned above are elaborated (including various examples) in Section 3. The correctness of the solutions is tested concerning sample input programs together using an automated before-after testing framework containing executable program test cases. Some test cases are based on the examples of Section 3, while some of them are hidden from the user – these cases check if the refactorings have been carefully implemented such that they also handle more complex situations correctly. Further details about this framework, the additional evaluation criteria, and the solution ranking system can be found in Section 4.

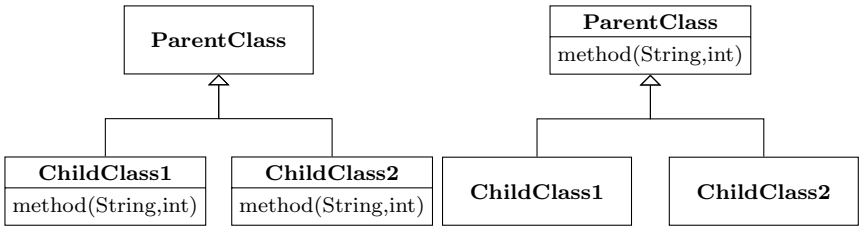
Based on the demanded functionality to be implemented by all solutions for the case study, further interesting extensions to those core tasks are mentioned in Section 5.

2 Case Description

Before diving into the details of the actual scenario to cope with, we motivate our case study once again by recalling the aim of refactorings. For this purpose, we use the very words of Opdyke, the godfather of refactorings, which say that refactoring is the same as "restructuring evolving programs to improve their maintainability without altering their (externally visible) behaviors" [3]. Hence, solutions of our case study have to (and, hopefully, want to) demonstrate the power of their chosen transformation tool by implementing graph-based program transformation and model-to-code incremental change propagation.

To describe the case study in a nutshell, we provide an intuitive example here, describing a program state where a natural need for restructuring arises.

Example. Refactoring Scenario 1 shows a basic example for a refactoring of a simple program called Java Program 1. In this case, we expect that a program transformation takes place which moves `method` from all child classes of the class `ParentClass` to this same superclass. (This is a classical refactoring which is called `Pull Up Method` and builds a significant part of our case study. `Pull Up Method` will be further specified and exemplified in Section 3.)



(a) Class Diagram of Source Code 1

(b) Class Diagram after the Refactoring of Source Code 1

Refactoring Scenario 1: Structure of the Java Program before and after the Application of the Refactoring *pum(ParentClass, method(String, int))*

To get a better understanding under which circumstances we except that a refactoring takes place and what exactly happens, we take a closer look on the corresponding Java source code. The source code for Refactoring Scenario 1 can be seen in Source Code 1.

We recognize that `ParentClass`, the common parent for `ChildClass1` and `ChildClass2` is empty, as shown in the class diagrams in 1. Moreover, the

`methods` in the child classes have the same signature (name and parameter list) - i.e., a `Pull Up Method` can take place if the two `method` implementations have an equivalent functionality.

In this example, it is obvious that the implementation of `method` in `ChildClass1` and `ChildClass2` is different but their behavior is identical. However, proving that two implementations are identical is undecidable in more complex cases. However, as refactorings are always requested by the software developers in our application scenario, we assume the decision whether the methods have equivalent functionality is done by the developer before initiating the actual refactoring. Nevertheless, the PG still contains an abstraction of the instruction level represented by the access references.

In the following, we give a schematic overall picture of the intended transformation chain (Figure 1) and its constituting artifacts. In Section 2.1, some details regarding the input Java code and the PG meta-model (called the *type graph*) are given, while Section 2.2 provides information on the individual transformation steps and the arising difficulties.

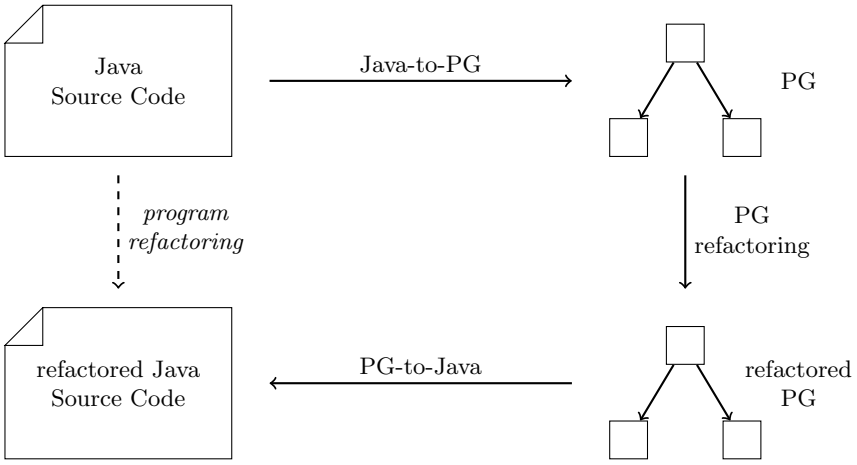


Fig. 1: Sketch of the Transformation Chain

2.1 Setting of the Case Study

Java Source Code All input programs considered for refactoring for TTC are fully functioning (although, abstract) Java programs built for the very purpose of checking the correct and thorough implementation of the given refactoring operations. Some test input programs are openly available and will be also described later on, while some others serve as blind tests and are not accessible to the solution developers.

paper-example01/src/example01/ParentClass.java

```
1  package example01;
2
3  public class ParentClass {
4
5          public ParentClass(){}
6  }
```

paper-example01/src/example01/ChildClass1.java

```
7  package example01;
8
9  public class ChildClass1 extends ParentClass {
10
11         public ChildClass1(){}
12
13         public void method(String message, int repeat) {
14                 for(int i=0; i<repeat; i++){
15                         System.out.println(message);
16                 }
17         }
18
19         public static void main(String[] args){
20                 ChildClass1 c1 = new ChildClass1();
21                 c1.method("c1:Hello World", 3);
22
23                 ChildClass2 c2 = new ChildClass2();
24                 c2.method("c2:Hello World", 3);
25         }
26 }
```

paper-example01/src/example01/ChildClass2.java

```
27 package example01;
28
29 public class ChildClass2 extends ParentClass {
30
31         public ChildClass2(){}
32
33         public void method(String string, int k) {
34                 int i = 0;
35                 while(i++ < k) System.out.println(string);
36         }
37 }
```

Source Code 1: Source Code of the Java Program shown in Refactoring Scenario 1a

The Java programs conform to the Java 1.4 major version. Moreover, the following features and language elements are explicitly *out of scope* for this case study:

- access modifiers (all elements have to be `public`)
- interfaces
- the keywords `abstract`, `static` and `final` except for `public static void main(String[] args)`
- exception handling
- inner, local and anonymous classes
- multi-threading

On the other hand, we would like to point out that the following Java language elements and constructs should be considered:

- inheritance
- method calls, method overloading and method overriding
- field accesses and field overriding
- constructors (handled as methods)

Type Graph for Representing Java Programs Figure 2 shows the type graph meta-model that is also part of the case study assets as an EMF meta-model – nevertheless, other meta-modeling technologies are allowed in solutions as well. The PG meta-model used in the solution has to contain all information of the recommended type graph, where some additional, custom abstraction layers, ancestor classes etc. are also allowed if required by the actual implementation. However, such a customization should not affect the semantics of the meta-model and, especially, should not enrich the type graph with additional information.

In conformance with the restrictions on the considered Java programs and with the nature of classical refactoring, the type graph does not include any modeling possibilities for access modifiers, interfaces, etc. and any code constituents lying deeper than the method level. In the following, we describe the meaning of some of the most important nodes and edges of the type graph.

The type graph represents the basic structure of a Java program. The node `TypeGraph` serves as a common container for each program element as the root of the containment tree. The Java package structure is modeled by the node `TPackage` and the corresponding self-edge for building a package tree. The node `TClass` stands for Java classes and contains members (the abstract class `TMember`), which can be method and field definitions (`TMethodDefinition` or `TFieldDefinition`, respectively). In addition, a `TClass` refers to the abstract class `TSignature`, which is the common ancestor of method and field signatures.

Methods and fields are represented by a structure consisting of three elements:

- The name of the method (field) contained in the attribute `tName` of `TMethod` (`TField`), which is globally visible in the PG.

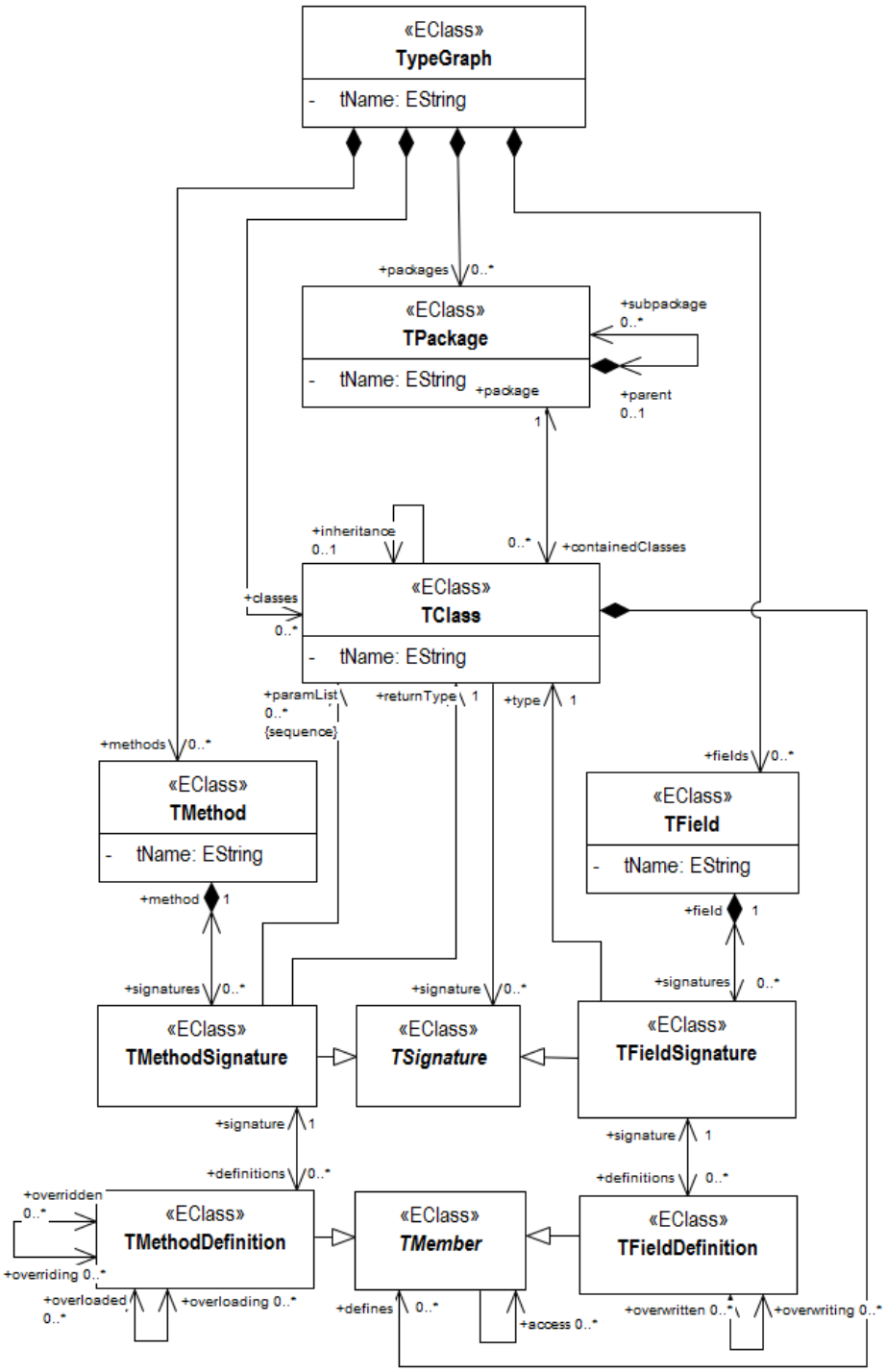


Fig. 2: Meta-model of the Proposed Type Graph

- The signatures of the methods (fields) of this name, represented by the class `TMethodSignature` (`TFieldSignature`). The signature of a method consists of its name and its list of parameter types `paramList`, while the signature of a field consists of its name and its `type`. Different signatures having the same name (i.e., a common container `TMethod` or `TField`) allow overloading. Signatures have a central role in the Java language, as all method calls and field accesses are based on signatures.
- `TMethodDefinition` (`TFieldDefinition`) is an abstraction layer representing the instruction level of Java. Relevant information is expressed by reference edges in the type graph. Overloading and overriding is declared by the corresponding edges between definition instances, although the overloading/overriding structure is also implicitly given through signatures, definitions and inheritance. The `access` edges between member instances represent dependencies between one member and the other members. This single edge type stands for all kinds of semantic dependencies among class members, namely *read*, *write* and *call*.

2.2 Transformations

The transformation chain consists of three consecutive steps which are detailed here.

First Step: Java Code to Program Graph Given a Java program as described in Sec. 2.1, it has to be transformed into an abstract PG representation conforming to the type graph meta-model (*ibid.*). Important note: the fact that some information necessarily disappears during this transformation calls for a solution where some preservation technique is employed, i.e., it is possible to rebuild those parts in the third step (see below) which are not present in the PG.

We remark that any intermediate program representations like JaMoPP[1], MoDisco[2], AST models etc., are allowed to facilitate the Java-to-PG and PG-to-Java transformations.

Second Step: Refactoring of the Program Graph This step essentially consists in an endogenous (PG-to-PG) restructuring of the program graph, according to the specifications of the refactoring operations `Pull Up Method` resp. `Create Superclass`. For those specifications and actual refactoring examples, see Sec. 3.

Third Step: Program Graph to Java Code As already mentioned at the first step (Java-to-PG), one of the most difficult tasks is to create a solution which provides a means to recover the program parts not included in the PG when

[1] `http://www.jamopp.org`
[2] `http://eclipse.org/MoDisco/`

transforming its refactored state back into Java source code. In other words, it is impossible to implement the Java-to-PG and the PG-to-Java transformations (the first and the third step) independently of each other. Furthermore, over the challenges posed by the abstraction level of the PG, one has to pay extra attention if a newly created PG element has to appear in the refactored code.

The resulting Java code has to fulfill the requirements of (i) having those code parts unchanged which are not affected by the refactoring and (ii) retaining the observable behavior of the input program. These properties are checked using before-after testing (as usual in the case of behavior-based test criteria) provided by the automated test framework that is part of the case study and is further described in Section 4.

After this brief overview of both the static and the dynamic ingredients of the transformation scenario to be dealt with, we proceed as follows: In Section 3, we put the second step in Sec. 2.2 under the microscope and present the two aforementioned refactoring operations with associated examples to also provide an intuition how and why they are performed. Thereupon, in Section 4, we describe our automated before-after testing framework for checking the correctness of the implementations, which also serves as a basis for the solution ranking system described in the same section including further evaluation criteria.

3 Refactorings

The refactoring descriptions that follow are based on the informal specification of Fowler [1], although using a somewhat more formal specification style inspired by [2]. Our aim was to find a balance between not over-specifying the operations in order to leave space for creativity and still avoiding ambiguity as far as possible.

3.1 Pull-up Method

First, we provide an intuition of `Pull Up Method` textually à la Fowler [1]. Additionally, we give some further information and examples to clarify the requirements.

Situation and action. There are methods with identical signatures (name and parameters) and equivalent behaviours in direct subclasses of a single superclass. These methods are then moved to the superclass, i.e., after the refactoring, the method is a member of the superclass and it is deleted from the subclasses.

Motivation. This refactoring aims at eliminating code duplicates, which can have negative effect on the maintainability of the software.

Graphical representation. Figure 3 shows a schematic representation of how `Pull Up Method` is performed. We use the elements of the type graph introduced in Sec. 2.1 and a notation with left- and right-hand sides as usual for graph transformation. Here, the left-hand side shows which elements have to be present or absent in the PG when applying the refactoring to it; an occurrence of the left-hand side is replaced by the right-hand side by preserving or deleting the elements of it and optionally creating some new elements and gluing them to the PG. It is implicitly given through object names which parts are preserved. In addition, we explicitly show the parts to be deleted on the left-hand side in red and marked with `--` and the parts to be created on the right-hand side in green and marked with `++`. The left-hand side also includes a forbidden pattern or *NAC*, which in this case consists of a single edge and is shown crossed through on the very left of the figure and is additionally highlighted in blue. This edge has to be absent in the input graph for the refactoring to be possible.

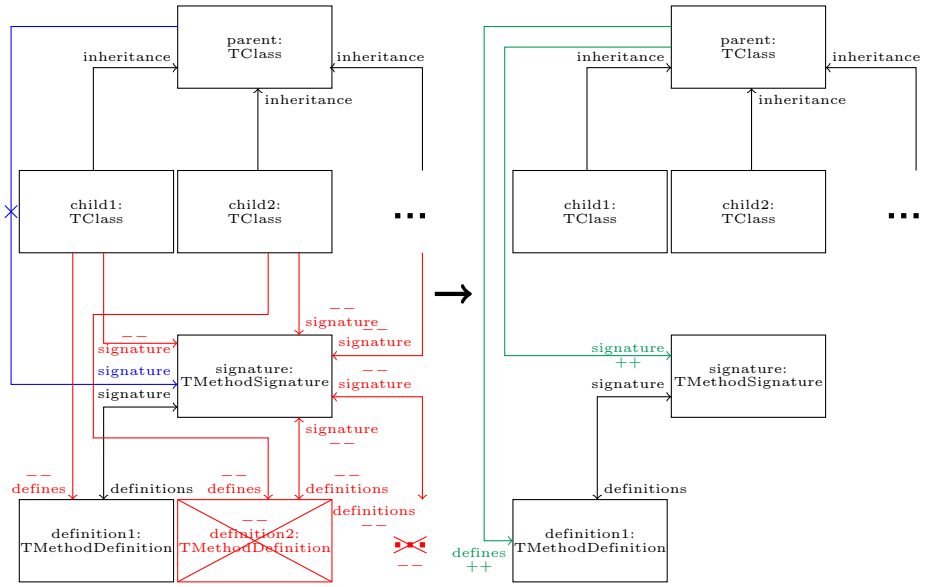


Fig. 3: Schematic Representation of a `Pull Up Method` Refactoring - Left-Hand and Right-Hand Side

Definition. In this case study, a `Pull Up Method` refactoring is specified as `pum(parent, signature)` with the following components:

- a superclass `parent`, whose direct child classes are supposed to contain at least one equivalent method implementation, and
- the method signature `signature` of such an equivalent method implementation, which represents the method to be pull-upped to `parent`.

Note that the equivalence of the implementations of `signature` in the child classes is considered here as a user decision and checking the method bodies for this equivalence is out scope for this case study.

In case the application conditions (see below) are fulfilled, the method signature `signature` as well as a corresponding method definition will be part of the `parent`. The copies of the other definitions of `signature` will be deleted from all child classes. Note that a `Pull Up Method` instance does not necessarily represent a valid refactoring - it marks merely a part of the input program where it is looked for a possible pull-up action.

Application conditions. The following preconditions have to be fulfilled for a `Pull Up Method` refactoring instance `pum(parent, signature)` so that a refactoring can take place:

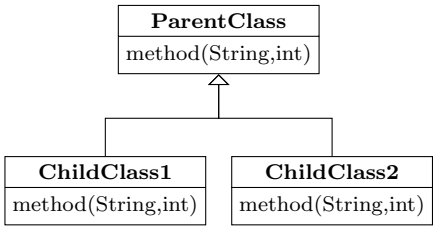
1. Each child class of the class `parent` has at least one common method signature `signature` with the corresponding method definitions ($definition_i$ for the i-th child class) having equivalent functionality.
2. Each $definition_i$ of `signature` in the child classes is only accessing methods and fields accessible from `parent`. Methods and fields defined in the child classes are not accessible.

Important remarks. Although it is not explicitly shown in Figure 3, all access edges in the PG pointing to a method definition deleted by the refactoring have to be redirected to point to the one which is preserved, so that subsequent refactorings are able to consider a coherent state of the PG. The actual choice of the preserved definition is irrelevant and the definitions can be arbitrarily matched, as the actual method implementations are out of scope for this case study.

Examples

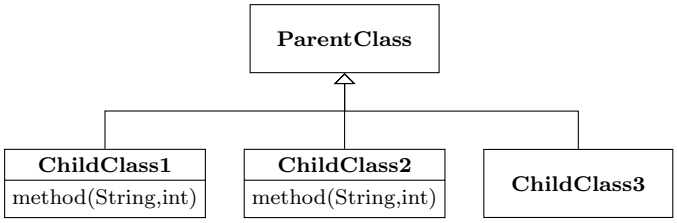
Example 1. Our first and most basic example for `Pull Up Method` is the one we have already shown as a general motivation for refactoring in the introduction part of Section 2.

Example 2. Given the program in Refactoring Scenario 2, the `Pull Up Method` refactoring `pum(ParentClass, method(String, int))` seen in the previous example is not possible. In `ParentClass`, a method with the given signature is already present which is overridden by methods in `ChildClass1` and `ChildClass2`. Accordingly, the NAC shown on the left-hand side of Figure 3 is violated.



Refactoring Scenario 2: Refactoring `pum(ParentClass, method(String, int))` not possible – `method(String, int)` already exists in `ParentClass`

Example 3. Given the program in Refactoring Scenario 3, the `Pull Up Method` refactoring `pum(parent, method(String, int))` is not possible. In this case, Precondition 1 is not fulfilled as `ChildClass3` does not contain the common method with the signature `method(String, int)`.



Refactoring Scenario 3: Refactoring `pum(ParentClass, method(String, int))` not possible – one of the child classes does not have `method(String, int)`

All examples shown here also have a corresponding test case in our test framework which is described in Sec. 4, with the example programs being accessible to the solution developers. In addition, there are some built-in test cases that are hidden in the framework and check trickier situations. For each of these hidden test cases, a textual hint for its purpose is provided by the test framework.

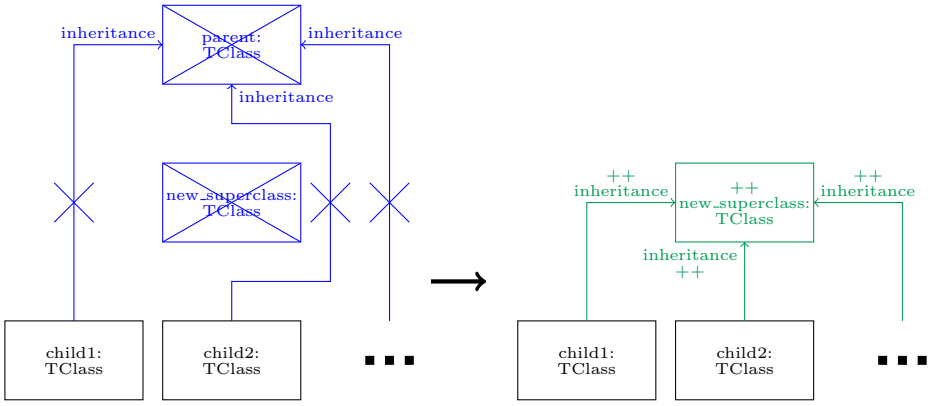
3.2 Create Superclass

The refactoring operation `Create Superclass` is described in a similar fashion as the `Pull Up Method` refactoring above.

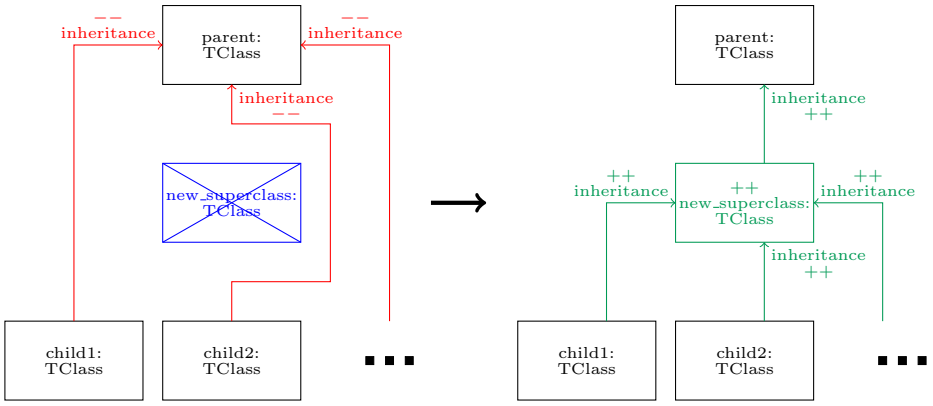
Situation and action. There is a set of classes with similar features. As a first step towards an improved program structure, a new common superclass of these classes is created.

Motivation. The `Create Superclass` refactoring aims at proactively avoiding code duplicates by refining the program structure for future implementation increments.

Graphical representation. Figure 4 shows a schematic representation of how the `Create Superclass` refactoring is performed with the same notation as by the `Pull Up Method` refactoring above. The classes either has to have the same superclass in the PG or none of them has a superclass modeled in the PG. (Note that from a technical point of view, each Java class has a superclass. Also, the distinction above refers to the representation in the PG.) Here, both cases are shown.



(a) The classes have no superclass in the PG



(b) All classes have the same superclass in the PG

Fig. 4: Schematic Representation of a `Create Superclass` refactoring – Left-Hand Side and Right-Hand Side

Definition. In this case study, a `Create Superclass` instance is defined as `csc(classes, new_superclass)` consisting of the following components:

- a list of classes `classes`, where all classes have identical inheritance relations (i.e., each of them inherits from the same class or they do not inherit from any class in the PG), and
- a superclass `new_superclass`, which does not exist before the refactoring and has to be generated.

In case the application pre- and postconditions (see below) are fulfilled, a new class `new_superclass` will be created which becomes the superclass of the classes in `classes`. Note that a `Create Superclass` refactoring does not necessarily represent a valid refactoring - it marks merely a part of the input program where it is looked for a possible refactoring operation.

Application conditions. The following precondition has to be fulfilled for a `Create Superclass` instance `csc(classes, new_superclass)` so that a refactoring can take place:

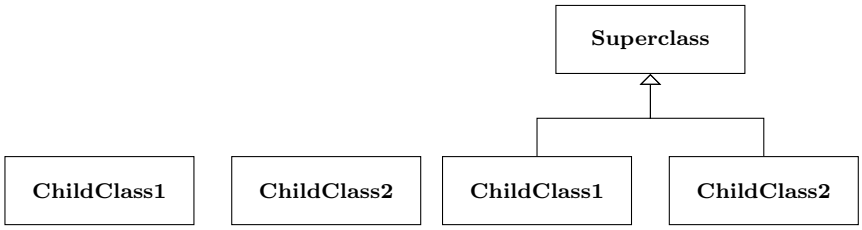
1. The classes contained in `classes` are implementing the same superclass. Note that classes with no explicit inheritance reference in Java are implementing `java.lang.Object` – modeling this class explicitly in the PG is a developer decision which does not influence the conditions for `Create Superclass`.

Additionally, the result of $csc(classes, new\_superclass)$ has to fulfill the following postconditions:

1. Each class in `classes` has an inheritance reference to `new_superclass`.
2. In case the classes in `classes` had an explicit inheritance reference to a superclass `parent` before the refactoring, their new superclass `new_superclass` has an inheritance reference to `parent`.

Examples

Example 1: Refactoring Scenario 4 shows the most basic example on which `Create Superclass` is applicable. The refactoring operation `csc({ChildClass1, ChildClass2}, NewSuperclass)` is possible as the desired new class does not exist yet.



(a) Class Diagram before the Refactoring

(b) Class Diagram after the Refactoring

Refactoring Scenario 4: Structure of the Java Program before and after the Application of the Refactoring `csc({ChildClass1,ChildClass2}, NewSuperclass)`

As demonstrated by the previous example, the `Create Superclass` refactoring itself is relatively uncomplicated, however, there are additional hidden test cases in the framework for `Create Superclass` as well. Note that, as already stated before, the main challenge by this refactoring is not to restructure the PG but to propagate the new element into the Java source code.

4 Evaluation

In this section, we introduce our test framework ARTE (Automated Refactoring Test Environment) for checking the correctness of implementations (Sec. 4.1) and the criteria and the scoring system which will be used to evaluate and rank the submitted solutions (Sec. 4.2).

4.1 Test Framework

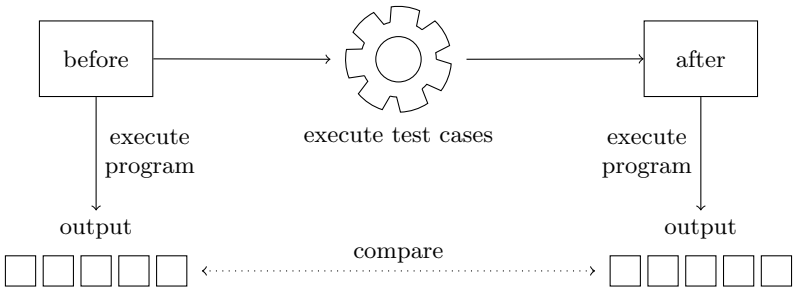


Fig. 5: Schematic Process of Before-after Testing

To enable the evaluation and ranking of the solutions for our case study, we have created an automated refactoring testing environment called ARTE,

whose mechanism is sketched in Figure 5. This test framework relies on the well-known principle of before-after testing, which is often used in behavior-critical scenarios: the behavior of the input is determined by stimulating it through the test environment and it is then checked if the output of the transformation reacts identically to the same stimulation.

In our framework, before-testing consists in compiling and executing the program and recording its console output. On the other hand, after-testing consists in compiling and executing the refactored program created by the actual solution under test, and comparing its console output to the one recorded in the before-testing phase.

The testing procedure is described in *test cases*. A test case consists of the following:

- a Java program assigned to it, on which the transformation takes place (one program can be assigned to multiple test cases) and
- a sequence of commands which can be (i) actual transformation operations or (ii) assertions to check if the transformations provided the expected result (e.g., nothing has changed if there is no correct refactoring possible). Note that a transformation operation cannot be executed without a corresponding assertion check for success.

The *execution* of a test case comprises the following steps:

- the before-testing phase as described above,
- the execution of the commands in the test case and
- the after-testing phase as described above.

For further details on how to use our testing framework ARTE, please refer to Appendix A.

4.2 Ranking Criteria and Scoring System

In this section, we propose a systematic way of evaluating and ranking the solutions for the case study. Although some evaluation criteria leave space for the opinion of the reviewers, we would like to ask the solution developers to provide some preliminary numbers which will serve as a basis for the reviewers. We will indicate by the individual evaluation aspects what is actually expected from the solution developer.

There is a total of 100 points that can be achieved by a solution. These 100 points are composed as follows (with a detailed description of the various aspects thereafter):

- max. 60 points: successful accomplishment of the test cases (public, hidden and to-be-announced ones)
- max. 10 points: comparison of the execution times of the solutions
- max. 30 points: various soft aspects regarding quality, verified by the reviewers (15 points per reviewer assuming 2 reviewers per solution)
- ...and a maximum of 10 points *beyond* the total of 100 by comparing how much of the case extensions described in Section 5 has been implemented

Accomplishing the test cases - 60 points. By the final ranking of the solutions, there are three kinds of test cases considered: (i) the public ones, which are part of the test framework ARTE and have been also discussed in Sec. 3, (ii) the hidden ones, also being part of ARTE but being not further specified except for some hints within ARTE and (iii) some additional test cases which will not be announced until the final evaluation occurs. There is a fixed amount of points assigned to each test case; these numbers are not public, however, the developers may assume that the point distribution reflects the levels of difficulty. *The solution developer should provide*: a simple summary of the test cases accomplished by the solution.

Execution times - 10 points. The test framework ARTE provides an execution time measurement (per test case), whose result is then displayed on the console in the test summary. The fastest solution gets 10 points and the slowest 1 point, while the remaining ones will be distributed homogeneously on this scale. *The solution developer should provide:* the overall execution time of all the test cases in the framework. (Note that this measurement possibly incorporates test cases where the actual solution fails, on the other hand, this rating procedure makes it possible to appreciate those solutions which quickly recognize an input not in their scope.)

Reviewer opinion - 2 x 15 points. There are 6 aspects (listed below) along which the quality of the solution should be ranked on a scale of 1 (lowest) to 5 (highest), accepting that such a ranking cannot be free of subjective opinions. Each reviewer has 15 points to award to the solution, which is calculated as the sum of the scores given for the individual aspects divided by 2.

- **Comprehensibility**: we think that the question if a solution works with an *understandable* mechanism which is not exclusively accessible for the high priests of a cult is of high importance, especially in the scope of the Transformation Tool Contest where such a comprehensible solution facilitates discussion and contributes to a profitable event.
- **Readability**: in contrast to comprehensibility, this aspect refers to the outer appearance of the tool - whether it has a nice and/or user-friendly interface, can be easily operated, maybe even with custom-tailored commands or a DSL, ...
- **Communication with the user**: although related to readability, this aspect refers to the quality, informativeness and level of detailedness of the actual messages given to the user. In other words: Am I as user informed that everything went smoothly? In case of some failure or malfunction, am I thoroughly informed what actually went wrong?
- **Robustness**: this classical software quality aspect characterizes how a software behaves if put into an erroneous environment, getting malformed input, ... E.g., what happens if some out-of-scope keywords appear in a Java program to be refactored?

- **Extendability**: this one also examines the inner structure of the solution concerning its possibilities to expand in the future. E.g., would it be easily feasible to build in the support of additional Java constructs or new refactorings?
- **Debugging**: in contrast to readability, this aspect refers solely to the debugging capabilities of the tool used to create the solution. In case a problem is uncovered through erroneous behavior, what means are provided to locate the cause of a design failure? Does the tool provide suggestions for fixing errors? How precise are the debug messages?

The solution developer should provide: an evaluation score (from 1 to 5) with a short textual explanation to each aspect of his own solution, in order to facilitate a more balanced evaluation and to provide the reviewers with a starting point for the evaluation.

We have created a simple online form[3] for the reviewers to send in their opinion regarding the aspects above. Ideally, the reviewer should be able to test the tool and, thus, to check the provided execution time measurement. It is also possible using this form to remark a deviating measurement result (as well as to provide an extension score if applicable – see Section 5).

How the extension bonus of 10 points can be achieved is a topic of Section 5.

5 Case Extensions

While the core case described in Sections 1-4 is already a full-fledged refactoring use-case on its own right, it can still be extended in various ways inspired by the theory and practice of refactorings. Here, we mention some interesting possibilities for extending a solution beyond the requirements of the core case. It is not our aim to test any extension implementation for correctness and, thus, our testing framework does not have any means to do so - the mentioned extensions rather serve as an inspiration for creative thinking concerning the vast and exciting universe of refactorings. Nevertheless, there is a limited amount of bonus points (maximally 10 points altogether) which can be achieved with providing some (maybe fragmentary) answers to one or more extensions or at least outlining a concept with relation to the core case solution. These points are awarded according to the reviewers' opinion and we only give some recommendations which may serve as scoring guidelines. Although it would be theoretically possible, note that the sum of the acquired bonus points can never exceed 10 in order to keep the focus on carefully implemented core solutions. The final bonus score is calculated as the average of the reviewers' scores.

[3] `http://goo.gl/forms/8VJuiD82Sg`

5.1 Extension 1: Extract Superclass

The two refactoring operations considered in the core case, namely `Pull Up Method` and `Create Superclass`, are simple actions compared to some complex operations which are still described as a single refactoring step in the literature. A classical example for such a more complex refactoring is `Extract Superclass`, which can be specified as a combination of `Create Superclass` and `Pull Up Method` (and its pendant for fields, `Pull Up Field`). After executing a `Create Superclass` for some classes, one can use `Pull Up Method` resp. `Pull Up Field` on the newly created parent class to move the common members there.

Recommendation. 10 points for a full-fledged implementation which can be executed on an appropriate example program; 6 points for a working implementation which misses `Pull Up Field`; 1-2 points for a concept sketch using the actual rule implementations.

5.2 Extension 2: Propose Refactoring

In our core refactoring use-case, the hypothetical user already realized the need of refactoring and also identified the spot where it would be possible and the kind of action to be executed. Nevertheless, one can imagine a somewhat orthogonal approach where an automatic refactoring environment permanently monitors an evolving software and proactively proposes refactorings being feasible on the code base. To be more concrete, as a first step towards such a system, one might implement a method which takes as input the whole program and returns one (or more) feasible refactoring(s).

Recommendation. 10 points for a full-fledged implementation which can be executed on an appropriate example program; 5-10 points for an alternative way of implementation according to its scope and usability; 1-5 points for a plausible concept sketch according to its ambition and clarity.

5.3 Extension 3: Detecting Refactoring Conflicts

From a practical point of view, it is not unlikely that two developers of the same software might want to execute refactorings independently of each other. In this case, it can happen that the refactored code states are not compatible to each other anymore and a merge is not possible. Concerning only two alternative refactorings, it is equivalent with stating that the result of the refactorings is not independent of their execution sequence. As a concrete step towards a conflict detection for refactorings, one can e.g. think of extending the framework so that it checks consequent refactoring operations and notifies the user if their execution sequence is considered as critical.

Recommendation. 10 points for a full-fledged implementation which can be executed on an appropriate pair of refactorings; 6-10 points for an alternative way of implementation according to its scope and usability; 1-8 points for a plausible concept sketch according to its ambition and clarity.

Beyond the ones mentioned above, the number of imaginable extensions regarding the supported refactorings or the framework is unlimited. The reviewers can also reward some other creative extension approaches using the extension score.

References

1. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
2. Mens, T., Eetvelde, N.V., Demeyer, S., Janssens, D.: Formalizing Refactorings with Graph Transformations. Journal of Software Maintenance 17(4), 247–276 (2005)
3. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. Tech. rep. (1992)

A Appendix: Handbook of the Automated Refactoring Test Environment ARTE

ARTE is a Java terminal program which executes test cases specified in a Domain Specific Language (DSL) on 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 test cases are collected in a test suite in ARTE. The tests aim at checking the correct analysis of pre- and postconditions for refactorings and the execution of these refactorings.

For executing the provided test framework, Java JDK 1.7 is needed and the path variable has to be set to point to the JDK and not to a JRE. With a JRE and no JDK, the test framework will still start but the compilation of Java programs during testing will fail.

ARTE has been tested on Windows command line and in Bash. However, ARTE should be executable in every Java-capable terminal.

A.1 Case Study Solutions and ARTE

A solution for the case study has to implement an interface that specifies method signatures which ARTE relies on. This interface is called `TestInterface` and is provided in the file `TTCTestInterface.jar`. Additionally, the solutions have to be exported as a simple (not executable) JAR file. This file has to contain a folder `META-INF/services` with a file called `arte.interfaces.TestInterface`. This latter file has to contain the fully qualified name of that class which implements the `TestInterface`. In the `TTCSolutionDummy` project, a dummy implementation of this interface is demonstrated.

An implementation fulfilling these conditions can be dynamically loaded into ARTE using the Java `ServiceLoader`. Further information can be found on the Oracle website [4][5].

The single methods which have to be implemented are:

getPluginName():
Returns the name of the actually loaded solution.

setPermanentStoragePath(File path):
Is called by ARTE to hand over a location at which data can be stored permanently by the solution.

setTmpPath(File path):
Is called by ARTE to hand over a location at which data can be stored temporally. All contents written to this location will be automatically deleted by closing ARTE.

[4] `http://www.oracle.com/technetwork/articles/javase/extensible-137159.html`

[5] `http://docs.oracle.com/javase/7/docs/api/java/util/ServiceLoader.html`

setLogPath(File path):
Is called by ARTE to hand over a location at which logs can be stored permanently. ARTE will store reports at this location as well.

createProgramGraph(String path):
Is called by ARTE to instruct the solution to build the program graph for the Java program located at `path`.

applyPullUpMethod(Pull_Up_Refactoring refactoring):
Is called by ARTE for a `Pull Up Method` refactoring to be performed. The structure of the type `Pull_Up_Refactoring` is explained in the following DSL part – its fields have similar names as the corresponding keywords of the DSL. Note that `name` fields contain names of variables inside the DSL and not method or class names.

applyCreateSuperclass(Create_Superclass_Refactoring refactoring):
Is called by ARTE for a `Create Superclass` refactoring to be performed. For the type `Create_Superclass_Refactoring` holds the same as for the type `Pull_Up_Refactoring` above.

synchronizeChanges():
Is called by ARTE to instruct the loaded solution to synchronize the Java source code with the PG. This means that the changes made on the PG have to be propagated into the Java program.

A.2 Defining Test Cases

We provide a custom DSL to make the creation of new test cases more convenient. For developing test cases, we provide an Eclipse plug-in which supports syntax highlighting and basic validation of the test files. However, test files can be written using any text editor.

In the following, we show on a `Pull Up Method` and on an `Create Superclass` example how our DSL can be used to create test cases and how to perform tests using ARTE. We explain the commands within the examples in a practical, step-by-step fashion. For further information about commands not covered by these simple examples, refer to the in-line explanations and to Appendix B where a full command list is provided.

DSL Example - Pull Up Method

As `Pull Up Method` test case example, we recapitulate our Example 1 that has been used in Section 2 to motivate refactorings. The structure of the Java program used in this example is shown in Refactoring Scenario 1 and the corresponding source code in Source Code 1. On this program, we are going to execute the refactoring `pum(ParentClass, method(String, int))`.

Test cases are wrapped in a `TestFile` environment that also defines the name of the test case. This name should to be identical with the name of the file containing this `TestFile` environment. If this is not the case, it is automatically renamed during import into the test framework, which can lead to failing imports with no obvious cause. The `TestFile` name has to be unique.

TestFile 1.1: PUM Example 1

```
1  TestFile public_pum_1 {
```

A `TestFile` contains everything needed for a test execution, namely classes, methods, refactorings and test cases. All elements used in the test cases have to be defined in the corresponding test file. Therefore, we define all classes we want to use in the test case.

In our example, the first class to be defined is `ParentClass`. To unambiguously identify the class in the program during test execution, the package containing the class has to be given as well. If the class is contained in the default package, the `package` parameter can be omitted.

```
2    class existing_parent {
3      package "example01"
4      name    "ParentClass"
5    }
```

As we have to check after the refactoring whether the pull-upped method is no longer contained in the child classes, those have to be defined as well.

```
6    class child1 {
7      package "example01"
8      name    "ChildClass1"
9    }
10
11   class child2 {
12     package "example01"
13     name    "ChildClass2"
14 }
```

Required primitive types and classes from libraries have to be also explicitly defined. In this example, we need these in the signature of the method to be pulled-up.

```
15   class String {
16     package "java.lang"
17     name    "String"
18   }
19
20   class int {
21     name "int"
22   }
```

For specifying a method signature, the name of the method and its parameters are necessary. The `params` command is optional as a method may have an empty parameter list. The order of the parameter list is important.

```
23    method child_method {
24      name    "method"
25      params String, int
26    }
```

According to Example 1, we are going to specify the `Pull Up Method` refactoring `pum(ParentClass, method(String,int)`. For this purpose, we have defined the necessary elements above and now, we combine them using the keyword `pullup_method`. (A refactoring can be used in multiple test cases within the test file.)

```
27    pullup_method executable_pum {
28      parent existing_parent
29      method child_method
30    }
```

Each test case has a mandatory description, which will be displayed during execution of the test case. As second argument, the name of a Java program is given. By having a look on the file structure shown in Source Code 1, it can be seen that this program name refers to the folder containing the input program.

The single steps of a test case are defined in a list starting with the keyword `testflow`. A `testflow` environment automatically induces both before- and after-testing. As the previously defined refactoring is supposed to succeed on the given Java program, we assert a successful refactoring by using the `assertTrue` command. Refactorings can only be executed with an accompanying assertion.

After executing the refactoring, we check if the resulting Java program has the structure shown in Refactoring Scenario 1. Therefore we are checking if the method has been moved to the parent and if the child classes do not contain the method anymore.

```
31    case pub_pum1_1_paper1 {
32      description "PUM-POS: (paper-ex1) Pull-up of two ..."
33      program "paper-example01"
34
35      testflow {
36        assertTrue(executable_pum)
37
38        existing_parent contains child_method
39        child1 ~contains child_method
40            child2 ~contains child_method
41      }
42    }
43  }
```

DSL Example - Create Superclass In the following, we describe a test case for a `Create Superclass` refactoring. For this purpose, we recycle our example shown in Refactoring Scenario 4. The refactoring `csc({ChildClass1, ChildClass2}, Superclass)` is expected to succeed.

Again, we first define the necessary elements for the refactoring. The classes for which a new superclass will be created are enumerated in a list called `child` by using the `classes` keyword. One class can be added to multiple lists and lists can be used by multiple refactorings.

```
1  TestFile public_exs_1 {
2
3    class child1 {
4      package "example04"
5      name        "ChildClass1"
6    }
7
8    class child2 {
9      package "example04"
10     name        "ChildClass2"
11   }
12
13   classes child {child1, child2}
```

Elements defined in a `TestFile` do not have to exist in the input or output program. However, accessing these elements will result in a failure if they have not been created before by, e.g., a refactoring. Here, we define the variable `new_superclass` as a "placeholder" for the class `Superclass` which will be created by the `Create Superclass` refactoring.

```
14   class new_superclass {
15     package "example04"
16     name "Superclass"
17   }
```

For the definition of the `Create Superclass` refactoring, we are referencing the elements defined before.

```
18   create_superclass refactoring {
19     child childs
20     target new_superclass
21   }
22
23   case pub_exs1_1 {
24     description "EXS–POS: Create a superclass for two..."
25     program "example04"
26     testflow {
```

As we are expecting the refactoring to succeed, we use the `assertTrue` keyword. If we expect a refactoring to fail, we can use the keyword `assertFalse`. The additional keywords `expectTrue` and `expectFalse` can be used in ambiguous cases; these result in success if the expectation is fulfilled and in a warning instead of a failure otherwise. Additionally, these two keywords include an `else`-block where static tests on the unexpected outcome can be executed. For more details, refer to Appendix B.

```
27        assertTrue(refactoring)
```

The `step` keyword allows for grouping the different stages in a `testflow` but has no influence on the execution.

At this point, we have to check whether the child class extend the new superclass or not.

```
28        step{
29          child1 extends new_superclass
30          child2 extends new_superclass
31        }
32      }
33    }
34  }
```

It is possible to execute multiple refactorings in a single test case.

A.3 Using ARTE

On Windows, ARTE can be started by double-clicking `run_windows.bat`. On Linux, the file `run_linux.sh` has to be executed.

```
[foo@bar ARTE]$ sh run_linux.sh
```

If ARTE has been launched for the first time, a solution has to be loaded.

```
load --solution /home/foo/dummy.jar
```

The entered path has to be absolute. The referenced solution will be copied to the permanent storage path of ARTE. The same command has to be used again to load a different version of the solution. The previous loaded solution will be deleted.

Test cases can be loaded similarly. In contrast to the load solution command, multiple test cases can be imported.

```
load --test /home/public_pum_1.ttc /home/public_exs_1.ttc
```

The import of the Java programs is a bit more complex. In addition to the path where the program is located, the main class of the program has to be given as well. The Java programs have to be structured like the example shown in Source Code 1. The referenced program folder has to contain a `src` folder.

The package structure is represented by further subfolders containing classes. The referenced program folder is equivalent to an Eclipse project folder.

A Java program loaded into ARTE has to contain a class defining a `main` method that is executed during testing. At least the constructors used during the execution of the `main` method have to be explicitly defined. It is a good idea to define the constructors as in Source Code 1 where all constructors are explicitly given.

```
load --src /home/foo/paper-example01 --main ChildClass1
```

It is possible to print out each loaded test case and Java program.

```
testcases --list
```

```
programs --list
```

There are three ways to execute test cases:

1. Execute test cases by name. This is only possible for our public test cases and for self-written test cases. In this variant, multiple test cases can be chosen. If the name of a test file is entered, each test case in this file will be executed.

In the example below, all cases contained in the file `public_pum_1.ttc` and the test case `pub_exs1_1` will be executed.

```
execute --test public_pum_1.ttc pub_exs1_1
```

2. Execute all hidden test cases.

```
execute --hidden
```

3. Execute all public, hidden and self-written test cases.

```
execute --all
```

It is indispensable to use the `exit` command after using ARTE.

```
exit
```

Most of the presented commands can be executed in various ways. Feel free to find your favourite. The `help` command will help you with this. If you, e.g., want to know more about test cases, try the following:

```
help testcases
```

B Appendix: Command Table of the DSL

Command	Subcommand	Description
TestFile *file_id* **{**		A test file always starts with this command. The file has to be called "file_id.ttc".
	file_content	The content of a test file can be a combination of elements **class**, **classes**, **method**, **pullup_method**, **create_superclass** and **case**.
}		

Command	Subcommand	Description
class *class_id* **{**		The class command is used to describe a Java class.
	package [String]	An optional String value like "subsubpackage.subpackage.package".
	name [String]	The name of the class.
}		

Command	Subcommand	Description
classes *classes_id* **{**		The classes command is used to define sets of classes for further use.
	$class\_id_0, ..., class\_id_n$	A comma separated list of classes which should be grouped.
}		

Command	Subcommand	Description
method *method_id* **{**		The method command is used to describe a Java method signature.
	name [String]	The name of the method.
	param $class\_d_0, ..., class\_id_n$	Parameters are optional and are an ordered list of comma separated references to classes.
}		

Command	Subcommand	Description
pullup_method *refactoring_id* **{**		The definition of a `Pull Up Method` refactoring.
	parent *class_id*	A reference to the parent class whose childs a method should be pulled up from.
	method *method_id*	A reference to the method which should be pulled up.
{		

Command	Subcommand	Description
create_superclass *refactoring_id* **{**		The definition of a `Create Superclass` refactoring.
	classes *classes_id*	A reference to the set of classes for which a superclass should be created.
	target *class_id*	A reference to a class variable describing the superclass which will be created.
}		

Command	Subcommand	Description
case *test_case_id* **{**		A test case can be identified by the test suite through *test_case_id*.
	description [String]	A textual description of the test case. This description is also shown in the test tool.
	program [String]	The name of the program on which the test case should operate.
	testflow {	A container for the test commands.
	$test\_step_0, ..., test\_step_n$	An ordered list of test commands that can contain **step**, **assertTrue()**, **assertFalse()**, **expectTrue()**, **expectFalse()**, **contains**, **∼contains**, **extends**, **∼extends**, **synchronize** and **compile**.
	}	
}		

Command	Subcommand	Description
step {		Allows for grouping, has no effect on the execution.
	$test\_step_0, ..., test\_step_n$	An ordered list of test steps which can contain **step**, **assertTrue()**, **assertFalse()**, **expectTrue()**, **expectFalse()**, **contains**, ∼**contains**, **extends**, ∼**extends**, **synchronize** and **compile**.
}		

Command	Subcommand	Description
assertTrue(**assertFalse(**		Checks whether a refactoring has been executed successful. The result is compared with the assertion.
	refactoring_id	The refactoring which will be handed to the solution for execution.
)		

Command	Subcommand	Description
expectTrue(**expectFalse(**		Checks whether a refactoring has been executed successful. The result is compared with the expected result. If the expected result is not matched, the execution can still be successful.
	refactoring_id	The refactoring which will be handed to the solution for execution.
){		
	$test\_step_0, ..., test\_step_n$	An ordered list of test steps executed if the expectation has been matched. The test steps can contain **step**, **assertTrue()**, **assertFalse()**, **expectTrue()**, **expectFalse()**, **contains**, ∼**contains**, **extends**, ∼**extends**, **synchronize** and **compile**.
} else {		
	warning [String]	A message displayed if the else block has been entered.
	$test\_step_0, ..., test\_step_n$	An ordered list of test steps executed if the expectation has not been matched. The test steps can contain **step**, **assertTrue()**, **assertFalse()**, **expectTrue()**, **expectFalse()**, **contains**, ∼**contains**, **extends**, ∼**extends**, **synchronize** and **compile**.
}		

LHS Variable	Command	RHS Variable	Description
class_id	**contains**	*method_id* *field_id*	Checks if the method or field (RHS) is contained in the class (LHS). The test case fails if the method or field is not contained in the class.

LHS Variable	Command	RHS Variable	Description
class_id	∼**contains**	*method_id* *field_id*	Checks if the method or field (RHS) is not contained in the class (LHS). The test case fails if the method or field is contained in the class.

LHS Variable	Command	RHS Variable	Description
class_id	**extends**	*class_id*	Checks whether the LHS class extends the RHS class. The test case fails if LHS does not extend RHS.

LHS Variable	Command	RHS Variable	Description
class_id	∼**extends**	*class_id*	Checks whether the LHS class does not extend the RHS class. The test case fails if LHS extends RHS.

Command	Description
synchronize	Triggers the propagation of changes made on the program graph to the Java source code.
compile	Triggers the compilation of the Java source code.