Java Refactoring Case: a VIATRA Solution*

Dániel Stein Gábor Szárnyas István Ráth
Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
daniel.stein@inf.mit.bme.hu, {szarnyas, rath}@mit.bme.hu

This paper presents a solution for the Java Refactoring Case of the 2015 Transformation Tool Contest. The solution utilises Eclipse JDT for parsing the source code, and uses a visitor to build the program graph. EMF-INCQUERY, VIATRA and the Xtend programming language are used for defining and performing the model transformations.

1 Introduction

This paper describes a solution for the extended version of the TTC 2015 Java Refactoring Case. The source code of the solution is available as an open-source project.[1] There is also a SHARE image available.[2]

The use of automated model transformations is a key factor in modern model-driven system engineering. Model transformations allow the users to query, derive and manipulate large industrial models, including models based on existing systems, e.g. source code models created with reverse engineering techniques. Since such transformations are frequently integrated to modeling environments, they need to feature both high performance and a concise programming interface to support software engineers. EMF-INCQUERY and VIATRA aim to provide an expressive query language and a carefully designed API for defining model queries and transformations.

2 Case Description

Refactoring operations are often used in software engineering to improve the readability and maintainability of existing source code without altering the behaviour of the software. The goal of the Java Refactoring Case [11] is to use model transformation tools to refactor Java source code. We decided to solve the extended version of the case. To achieve this, the solution has to tackle the following challenges:

1. Transforming the *Java source code* to a *program graph* (PG).
2. Performing the refactoring transformation on the program graph.
3. Synchronising the source code and the program graph.

The source code is defined in a restricted sub-language of Java 1.4. The EMF metamodel of the PG is provided in the case description. The case considers two basic refactoring operations: Pull Up Method and Create Superclass.

*This work was partially supported by the MONDO (EU ICT-611125) project.

[1] https://github.com/FTSRG/java-refactoring-ttc-viatra

[2] http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=ArchLinux64_java-refactoring-viatra.vdi

Submitted to:
TTC 2015

The solution is tested in an automated test framework, ARTE (Automated Refactoring Test Environment). ARTE runs a set of test cases, each consisting of a source code project and refactoring operations to check the correctness and the performance of the solution.

3 Technologies

Solving the case requires the integration of a model transformation tool and a Java source code parser. In this section, we introduce the technologies used in our solution.

3.1 EMF-INCQUERY

The objective of the EMF-INCQUERY [4, 6] framework is to provide a declarative way to define queries over EMF models. EMF-INCQUERY extended the pattern language of VIATRA2 with new features (including transitive closure, role navigation, match count) and tailored it to EMF models, resulting in the INCQUERY Pattern Language [5]. While EMF-INCQUERY is developed with a focus on *incremental query evaluation*, the latest version also provides a *local search-based query evaluation* algorithm.

3.2 VIATRA

The VIATRA framework supports the development of model transformations with a particular emphasis on event-driven, reactive transformations [9]. Building upon the incremental query support provided by EMF-INCQUERY, VIATRA offers a language to define transformations and a reactive transformation engine to execute certain transformations upon changes in the underlying model. The VIATRA project provides:

- An internal DSL over the Xtend [10] language to specify both batch and event-driven, reactive transformations.
- A complex event-processing engine over EMF models to specify reactions upon detecting complex sequences of events.
- A rule-based design space exploration framework to explore design candidates as models satisfying multiple criteria.
- A model obfuscator to remove sensitive information from a confidential model, e.g. for creating bug reports.

The current VIATRA project is a full rewrite of the previous VIATRA2 framework, now with full compatibility and support for EMF models. The history of the VIATRA family is described in [8].

3.3 Java Development Tools

The solution requires a technology to parse the Java code into a program graph model and serialize the modified graph model back to source code. While the case description mentions the JaMoPP [1] and MoDisco [2] technologies, our solution builds on top of the Eclipse Java Development Tools (JDT) [7] used in the Eclipse Java IDE as we were already using JDT in other projects.

Compared to the MoDisco framework (which uses JDT internally), we found JDT to be simpler to deploy outside the Eclipse environment, i.e. without defining an Eclipse workspace. Meanwhile, the JaMoPP project has almost completely been abandoned and therefore it is only capable of parsing Java 1.5 source files. While this would not pose a problem for this case, we think it is best to use an actively developed technology such as JDT which supports the latest (1.8) version of the Java language. As JDT is frequently used to parse large source code repositories, it is carefully optimised and supports lazy loading. Unlike JaMoPP and MoDisco, JDT does not produce an EMF model.

The JDT parser generates Abstract Syntax Trees (ASTs) from the provided source code files with an optional binding resolution mechanism. This way, the separate syntax trees can be connected to an Abstract Syntax Graph (ASG).

4 Implementation

The solution was developed partly in IntelliJ IDEA and partly in the Eclipse IDE. The projects are not tied to any development environment and can be compiled with the Apache Maven [3] build automation tool. This offers a number of benefits, including easy portability and the possibility of continuous integration.

The code is written in Java 8 and Xtend [10]. The queries and transformations were defined in EMF-INCQUERY and VIATRA, respectively. For developing the Xtend code and editing the graph patterns, it is required to use the Eclipse IDE. For setting up the development environment, please refer to the readme file.

4.1 Workflow of the Transformation

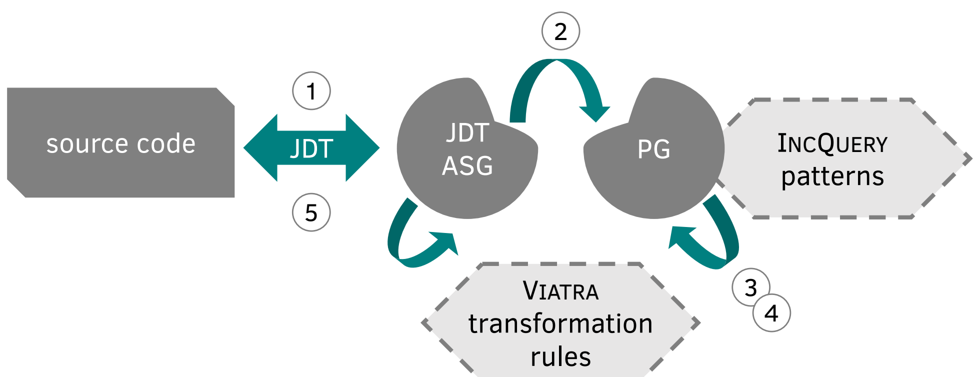

Figure 1: Workflow of the transformation.

Figure 1 shows the high-level workflow of the transformation. This consists of five steps: the source code is parsed into an ASG ①; a PG is produced ②; based on the PG, the possible transformations are calculated ③; if possible, these transformations are executed ④; finally, the results of the transformations are serialized ⑤. In the following, we discuss these steps in detail.

4.2 Parsing the Source Code ①

The solution receives the path to the directory containing the source files. JDT parses these files and returns each file parsed as an AST. These ASTs are interconnected, which means that using JDT's binding resolution mechanism, the developer can navigate from one AST to another one.

4.3 Producing the Program Graph ②

Since JDT does not produce EMF models, the generated ASTs do not support complex queries and traversal operations as the ones provided by EMF and EMF-based query languages (e.g. Eclipse OCL or EMF-INCQUERY). To extract information and to build the PG, our solution applies a visitor resulting a two-pass traversal on the ASG.

1. For each object, the visitor method creates the corresponding object(s) in the PG. Since the order of these visits is non-deterministic, the visitor maintains maps to store the mapping from the objects in the ASG to the objects in the PG.
   These maps provide trace information between the JDT model and the partially built PG. The visitor also collects the relations between JDT nodes and caches the unique identifiers of each connected node for every relation type.
2. After every compilation unit has been parsed, the previously populated caches are used to create the cross-references between the objects in the PG (e.g. TMember.access).

4.4 Extending the PG with the Trace Model ②③④

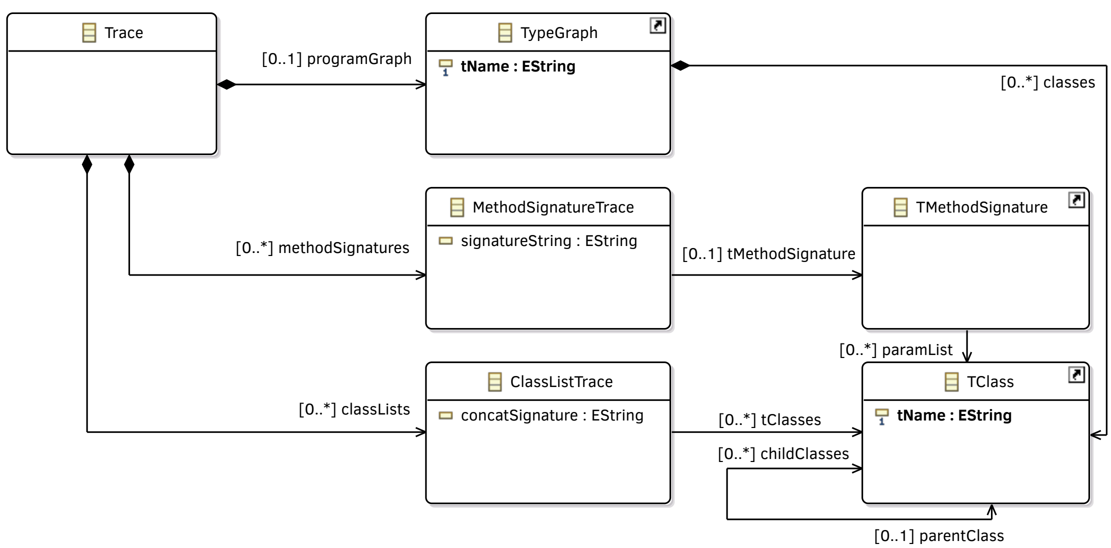

Figure 2: Metamodel of the trace model.

The main patterns for both refactoring operations contain a condition that EMF-INCQUERY does not support out of the box, e.g. checking "every child" (a collection of classes) of a certain class. Passing collections as pattern parameter is only possible with a workaround. Also, the INCQUERY Pattern Language does not support universal quantifiers. To overcome these limitations, we extended the program graph metamodel with a *trace model* shown in Figure 2. The trace model defines traces for method signatures and class lists:

- MethodSignatureTrace. Java methods are uniquely identifiable by their signature. The basic PG metamodel contains a TMethodSignature class, which only identifies itself with the name of the method (using a relation to the TMethod object) and the list of its parameter types.

To support querying TMethodSignature objects with EMF-IncQuery, we created a trace reference for each of them identified by their partial[3] method signature. For example, a method method() expecting a String and an Integer will have the `.method(Ljava/lang/String;I)` trace signature.

- ClassListTrace. To express the collection of classes, a ClassListTrace object will identify them with their signatures joined by the # character. For example, a list of the ChildClass1 and ChildClass2 classes in the example04 package has the `Lexample04/ChildClass1;#Lexample04/ChildClass2;` trace signature.

After the PG is produced, it is extended with the trace model. The traces are based on EMF-IncQuery patterns (Listing 1) and generated with a Viatra transformation (Listing 4).

The *universal quantifier* is implemented as a double negation of the existential quantifier using the well-known identity $(\forall x)P(x) \Leftrightarrow \neg(\exists x)\neg P(x)$.

4.5 Refactoring ③④

The refactoring operations are implemented as model transformations on the JDT ASG and the PG. Each model transformation is defined in Viatra: the LHS is defined with an EMF-IncQuery pattern and the RHS is defined with imperative Xtend code. As Viatra does not support bidirectional transformations, for each transformation on the PG, we also execute the corresponding actions on the ASG to keep the two graphs in sync.

4.5.1 Pull Up Method

After creating the method signature traces, the following preconditions must be satisfied before pulling up a method:

- every child class has a method with the given signature,
- the parent class does not have a method with this signature,
- the transformation will not create an unsatisfiable method or field access.

To decide whether the refactoring can be executed, every $\langle$parent class, method signature$\rangle$ pair satisfying the preconditions is collected by the main pattern (possiblePUM). The LHS is defined with six patterns in total. The execution is controlled by parameterising the main pattern listed in Listing 2. The RHS is defined in Listing 5, using one utility pattern.

4.5.2 Create Superclass

To create a new superclass, the parent class and the list of selected classes (connected to a class list trace) have to be passed to the pattern. The transformation can be executed if the following preconditions are satisfied:

- the target parent class does not exist,
- every selected child class has the same parent.

The LHS is defined in Listing 3 with the possibleCSC pattern using five other patterns. The RHS is defined in Listing 6, also using a utility pattern.

[3]The complete signature would also contain the defining type (class or interface) signature and the return type signature.

4.6 Transforming the ASG to Source Code ⑤

The changes in the ASG made by the transformations are propagated to the source code. JDT is capable of incrementally maintaining each source code file (compilation unit) based on the changes in its AST.[4]

5 Evaluation

The benchmarks were conducted on a 64-bit Arch Linux virtual machine running in SHARE. The machine utilized a single core of a 2.00 GHz Xeon E5-2650 CPU and 1 GB of RAM. We used OpenJDK 8 to run the ARTE framework and the solution.

5.1 Benchmark Results

We executed the tests and used the log files to determine the execution times. The execution times of the test cases are listed in Table 1.

test case	**time [s]**
pub_pum1_1_paper1	0.463
pub_pum1_2	0.013
pub_pum2_1	0.333
pub_pum3_1	0.094
pub_csc1_1	0.189
pub_csc1_2	0.093
hidden_pum1_1	0.063
hidden_pum1_2	0.013
hidden_pum2_1	0.114
hidden_pum2_2	0.082
hidden_csc1_1	0.081
hidden_csc1_2	0.007
hidden_csc2_1	0.058
hidden_csc3_1a	0.189
hidden_csc3_1	0.179

Table 1: Execution times for the test cases.

5.2 Analysis of the Solution

The results show that all public and hidden test cases have been executed successfully. Hence, we consider the solution *complete* and *correct*. As the test cases only contained small examples, we cannot draw conclusions on the performance of the solution. Still, it is worth noting that all test cases executed in less than half a second.

The implementation of the solution required quite a lot of code. The patterns were formulated in about 150 lines of IncQuery Pattern Language code. The transformations required 400 lines of Xtend code, while implementing the interface required by ARTE and the visitor for the transformation required more than 800 lines of Java code. However, the source code is well-structured and is easy to comprehend.

[4] The CompilationUnit.rewrite() method returns a set of text manipulation operations in the form of a TextEdit object.

6 Summary

This paper presented a solution for the Java Refactoring case of the 2015 Transformation Tool Contest. The solution addresses both challenges (bidirectional synchronisation and program transformation) and both refactoring operations (Pull Up Method, Create Superclass) defined in the case. The framework is flexible enough to allow the user to define new refactoring operations, e.g. Extract Class or Pull Up Field.

Acknowledgements. The authors would like to thank Ábel Hegedüs, Oszkár Semeráth and Zoltán Ujhelyi for providing valuable insights into EMF-INCQUERY and VIATRA.

References

[1] *JaMoPP*. `http://www.jamopp.org/index.php/JaMoPP`.

[2] *MoDisco*. `https://eclipse.org/MoDisco/`.

[3] Apache.org: *Maven*. `http://maven.apache.org`.

[4] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh & András Ökrös (2010): *Incremental Evaluation of Model Queries over EMF Models*. In: *Model Driven Engineering Languages and Systems, 13th International Conference, MODELS2010*, Springer, Springer, doi:http://dx.doi.org/10.1007/978-3-642-16145-2_6. Acceptance rate: 21%.

[5] Gábor Bergmann, Zoltán Ujhelyi, István Ráth & Dániel Varró (2011): *A Graph Query Language for EMF models*. In: *Theory and Practice of Model Transformations, Fourth Intl. Conf.*, *LNCS* 6707, Springer.

[6] Eclipse.org: *EMF-IncQuery*. `http://eclipse.org/incquery/`.

[7] Eclipse.org: *Java Development Tools (JDT)*. `https://eclipse.org/jdt/`.

[8] Eclipse.org: *VIATRA History*. `https://wiki.eclipse.org/VIATRA/History`.

[9] Eclipse.org: *VIATRA Project*. `https://www.eclipse.org/viatra/`.

[10] Eclipse.org: *Xtend – Modernized Java*. `https://www.eclipse.org/xtend/`.

[11] Géza Kulcsár, Sven Peldszus & Malte Lochau (2015): *The Java Refactoring Case*. In: *8th Transformation Tool Contest (TTC 2015)*.

A Appendix

A.1 Patterns

```
package hu.bme.mit.ttc.refactoring.patterns

import "platform:/plugin/TypeGraphBasic/model/TypeGraphBasic.ecore"

pattern methodSignature(methodSignature) {
  TMethodSignature(methodSignature);
}

pattern tClassName(tClass, className) {
  TClass(tClass);
  TClass.tName(tClass, className);
}
```

Listing 1: Patterns for generating the trace model.

```
package hu.bme.mit.ttc.refactoring.patterns

import "platform:/plugin/TypeGraphBasic/model/TypeGraphBasic.ecore"
import "platform:/plugin/TypeGraphBasic/model/TypeGraphTrace.ecore"

/*
 * Main decision pattern. If the preconditions are statisfied (parentClass
 * and methodSignatureTrace can be bound as parameters), the pattern returns
 * its parameters, if:
 *  - every child class has a method with the given signature (N = M)
 *  - the parent class does not have it already
 *  - the transformation will not create unavailable access
 */
pattern possiblePUM(parentClass : TClass, methodSignatureTrace : MethodSignatureTrace) {
  MethodSignatureTrace.tMethodSignature(methodSignatureTrace, methodSignature);

  // every child class has the method signature
  N == count find childClassesWithSignature(parentClass, _, methodSignature);
  M == count find childClasses(parentClass, _);
  check(N == M && N != 0);

  // parent does not already have this method
  neg find classWithSignature(parentClass, methodSignature);

  // the fields and methods will still be accessible after PUM
  neg find childrenClassMethodDefinitionsAccessingSiblingMembers(childClass, methodSignature);
}

pattern childClasses(parentClass : TClass, childClass : TClass) {
  TClass.childClasses(parentClass, childClass);
}

pattern childClassesWithSignature(parentClass : TClass, clazz : TClass, methodSignature : TMethodSignature)
    {
  TClass(parentClass);
  TClass.childClasses(parentClass, clazz);

  find classWithSignature(clazz, methodSignature);
}

pattern classWithSignature(clazz : TClass, methodSignature : TMethodSignature) {
  TClass(clazz);
  TMethodSignature(methodSignature);
  TMethodSignature.definitions(methodSignature, methodDefinition);
  TClass.defines(clazz, methodDefinition);
```

```
}

pattern methodsAccessingSiblingMembers(methodDefinition : TMethodDefinition) {
  TMember.access(methodDefinition, accessedMember);
  TClass.defines(tClass, methodDefinition);
  TClass.defines(tClass, accessedMember);
} or {
  TClass.defines(tClass, methodDefinition);
  TMember.access(methodDefinition, accessedMember);
  TClass.defines(otherClass, accessedMember);
  TClass.parentClass.childClasses(tClass, otherClass);
}

pattern childrenClassMethodDefinitionsAccessingSiblingMembers(parentClass : TClass, methodSignature :
    TMethodSignature) {
  TClass.childClasses(parentClass, childClass);
  TClass.defines(childClass, methodDefinition);
  TMethodSignature.definitions(methodSignature, methodDefinition);
  find methodsAccessingSiblingMembers(methodDefinition);
}

// fire precondition pattern
pattern classWithName(tClass : TClass, className) {
  TClass.tName(tClass, className);
}

// fire precondition pattern
pattern methodWithSignature(trace : MethodSignatureTrace, signature) {
  MethodSignatureTrace.signatureString(trace, signature);
}

// pattern for PG refactor
pattern methodDefinitionInClassList(parentClass : TClass, methodSignature : TMethodSignature, clazz :
    TClass, methodDefinition : TMethodDefinition) {
  TClass.childClasses(parentClass, clazz);
  TMethodSignature.definitions(methodSignature, methodDefinition);
  TClass.defines(clazz, methodDefinition);
}
```

Listing 2: Patterns for the Pull Up Method refactoring.

```
package hu.bme.mit.ttc.refactoring.patterns

import "platform:/plugin/TypeGraphBasic/model/TypeGraphBasic.ecore"
import "platform:/plugin/TypeGraphBasic/model/TypeGraphTrace.ecore"

/*
 * Main decision pattern. If the preconditions are statisfied (the
 * targetClass should not exist), the pattern returns its parameters, if:
 *  - every child class has the same parent
 */
pattern possibleCSC(concatSignature, methodSignature : TMethodSignature) {
  ClassListTrace.concatSignature(classListTrace, concatSignature);
  ClassListTrace.tClasses.signature(classListTrace, methodSignature);

  neg find childClassesWithDifferentParents(classListTrace, _, _);
}

pattern childClassesWithDifferentParents(classListTrace : ClassListTrace, classOne : TClass, classTwo :
    TClass){//, parentClassOne : TClass, parentClassTwo : TClass) {
  ClassListTrace.tClasses(classListTrace, classOne);
  ClassListTrace.tClasses(classListTrace, classTwo);
  find differentParents(classOne, classTwo);
}

```

```
pattern differentParents(classOne : TClass, classTwo : TClass) {
  TClass.parentClass(classOne, parentClassOne);
  TClass.parentClass(classTwo, parentClassTwo);
  parentClassOne != parentClassTwo;
} or {
  TClass(classTwo);
  find hasParent(classOne);
  neg find hasParent(classTwo);
} or {
  TClass(classOne);
  find hasParent(classTwo);
  neg find hasParent(classOne);
}

pattern hasParent(tClass : TClass) {
  TClass.parentClass(tClass, _);
}

pattern classesOfClassListTrace(concatSignature, tClass : TClass) {
  ClassListTrace.concatSignature(classListTrace, concatSignature);
  ClassListTrace.tClasses(classListTrace, tClass);
}

pattern methodSignatureAndTrace(trace : MethodSignatureTrace, methodSignature : TMethodSignature) {
  MethodSignatureTrace.tMethodSignature(trace, methodSignature);
}

// pattern for PG refactor
pattern packageWithName(tPackage : TPackage, packageName) {
  TPackage.tName(tPackage, packageName);
}

// pattern for PG refactor
pattern typeGraphs(typeGraph : TypeGraph) {
  TypeGraph(typeGraph);
}

// fire precondition pattern
pattern classWithName(tClass : TClass, className) {
  TClass.tName(tClass, className);
}
```

Listing 3: Patterns for the Create Superclass refactoring.

A.2 Transformations

```
package hu.bme.mit.ttc.refactoring.transformations

import TypeGraphBasic.TClass
import TypeGraphTrace.Trace
import TypeGraphTrace.TypeGraphTracePackage
import hu.bme.mit.ttc.refactoring.patterns.TraceQueries
import java.util.ArrayList
import java.util.List
import org.apache.log4j.Level
import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.incquery.runtime.api.AdvancedIncQueryEngine
import org.eclipse.incquery.runtime.evm.api.RuleEngine
import org.eclipse.incquery.runtime.evm.specific.RuleEngines
import org.eclipse.incquery.runtime.evm.specific.event.IncQueryEventRealm
import org.eclipse.viatra.emf.runtime.modelmanipulation.IModelManipulations
import org.eclipse.viatra.emf.runtime.modelmanipulation.SimpleModelManipulations
import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationRuleFactory
```

```
import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationStatements
import org.eclipse.viatra.emf.runtime.transformation.batch.BatchTransformation

class TraceTransformation {

  extension BatchTransformationRuleFactory factory = new BatchTransformationRuleFactory
  extension BatchTransformation transformation
  extension BatchTransformationStatements statements
  extension IModelManipulations manipulation

  extension TypeGraphTracePackage tgtPackage = TypeGraphTracePackage::eINSTANCE
  extension TraceQueries queries = TraceQueries::instance
  val AdvancedIncQueryEngine engine
  Resource resource
  val Trace trace

  new(AdvancedIncQueryEngine engine, Resource resource) {
    this(RuleEngines.createIncQueryRuleEngine(engine), resource)
  }

  new(RuleEngine ruleEngine, Resource resource) {
    engine = (ruleEngine.eventRealm as IncQueryEventRealm).engine as AdvancedIncQueryEngine
    transformation = BatchTransformation.forEngine(engine)
    statements = new BatchTransformationStatements(transformation)
    manipulation = new SimpleModelManipulations(iqEngine)
    transformation.ruleEngine.logger.level = Level::OFF
    this.resource = resource
    this.trace = resource.contents.get(0) as Trace
  }

  val methodSignatureTraceRule = createRule.precondition(methodSignature).action [
    val methodSignatureTrace = typeGraphTraceFactory.createMethodSignatureTrace
    trace.methodSignatures += methodSignatureTrace

    val sb = new StringBuilder(".")
    sb.append(methodSignature.method.TName)
    sb.append("(")
    methodSignature.paramList.forEach[sb.append(it.TName)]
    sb.append(")")

    methodSignatureTrace.signatureString = sb.toString
    methodSignatureTrace.TMethodSignature = methodSignature
  ].build


  def run() {
    fireAllCurrent(methodSignatureTraceRule)
  }

  def addNewClassListTrace(List<String> classSignatures) {
    val List<TClass> tClasses = new ArrayList
    for (signature : classSignatures) {
      tClasses += engine.getMatcher(TClassName).getAllValuesOftClass(signature)
    }

    val classListTrace = typeGraphTraceFactory.createClassListTrace
    classListTrace.concatSignature = classSignatures.join("#")
    classListTrace.TClasses += tClasses

    trace.classLists += classListTrace
    return classListTrace
  }
}
```

Listing 4: Transformation for generating the trace model.

```
package hu.bme.mit.ttc.refactoring.transformations

import TypeGraphBasic.TClass
import TypeGraphBasic.TypeGraphBasicPackage
import TypeGraphTrace.MethodSignatureTrace
import com.google.common.collect.BiMap
import hu.bme.mit.ttc.refactoring.patterns.PUMQueries
import java.io.File
import java.util.ArrayList
import java.util.List
import java.util.Scanner
import java.util.Set
import org.apache.log4j.Level
import org.eclipse.emf.ecore.util.EcoreUtil
import org.eclipse.incquery.runtime.api.AdvancedIncQueryEngine
import org.eclipse.incquery.runtime.evm.api.RuleEngine
import org.eclipse.incquery.runtime.evm.specific.RuleEngines
import org.eclipse.incquery.runtime.evm.specific.event.IncQueryEventRealm
import org.eclipse.jdt.core.dom.ASTNode
import org.eclipse.jdt.core.dom.CompilationUnit
import org.eclipse.jdt.core.dom.MethodDeclaration
import org.eclipse.jdt.core.dom.TypeDeclaration
import org.eclipse.viatra.emf.runtime.modelmanipulation.IModelManipulations
import org.eclipse.viatra.emf.runtime.modelmanipulation.SimpleModelManipulations
import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationRuleFactory
import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationStatements
import org.eclipse.viatra.emf.runtime.transformation.batch.BatchTransformation

class PUMTransformation {
  extension BatchTransformationRuleFactory factory = new BatchTransformationRuleFactory
  extension BatchTransformation transformation
  extension BatchTransformationStatements statements
  extension IModelManipulations manipulation

  extension TypeGraphBasicPackage tgPackage = TypeGraphBasicPackage::eINSTANCE
  extension PUMQueries queries = PUMQueries::instance

  val AdvancedIncQueryEngine engine
  val String parentSignature
  val String methodSignature
  val BiMap<String, CompilationUnit> compilationUnits

  new(AdvancedIncQueryEngine engine, String parentSignature, String methodSignature, BiMap<String,
     CompilationUnit> compilationUnis) {
    this(RuleEngines.createIncQueryRuleEngine(engine), parentSignature, methodSignature, compilationUnis)
  }

  new(RuleEngine ruleEngine, String parentSignature, String methodSignature, BiMap<String, CompilationUnit>
      compilationUnits) {
    engine = (ruleEngine.eventRealm as IncQueryEventRealm).engine as AdvancedIncQueryEngine
    transformation = BatchTransformation.forEngine(engine)
    statements = new BatchTransformationStatements(transformation)
    manipulation = new SimpleModelManipulations(iqEngine)
    transformation.ruleEngine.logger.level = Level::OFF

    this.parentSignature = parentSignature
    this.methodSignature = methodSignature
    this.compilationUnits = compilationUnits

    compilationUnits.values.forEach[ try { it.recordModifications } catch (Exception e) {}]
  }

  val PUMRule = createRule.precondition(possiblePUM).action [
    val parentClassKey = parentClass.TName
```

```
    val childClasses = engine.getMatcher(childClasses).getAllValuesOfchildClass(parentClass)

    var TypeDeclaration astParentClass
    var List<TypeDeclaration> astChildClasses = new ArrayList
    var List<MethodDeclaration> astMethodDeclarations

    astParentClass = findCompilationUnits(parentClassKey, childClasses, astChildClasses)
    astMethodDeclarations = findMethodDeclarations(astChildClasses, methodSignatureTrace)

    updateASTAndSerialize(astParentClass, astChildClasses, astMethodDeclarations)


    // --------------- /\ JDT transformation ------------- PG transformation \/ ---------------


    val methodDefinitionsToDelete = engine.getMatcher(methodDefinitionInClassList).getAllMatches(
      parentClass, methodSignatureTrace.TMethodSignature, null, null
    )

    val firstMethodDefinition = methodDefinitionsToDelete.get(0)
    val savedSignature = firstMethodDefinition.methodSignature
    val savedReturnType = firstMethodDefinition.methodDefinition.returnType
    val savedAccess = firstMethodDefinition.methodDefinition.access

    methodDefinitionsToDelete.forEach[
      it.clazz.signature.remove(it.methodDefinition.signature); // remove signature from class
      EcoreUtil.delete(it.methodDefinition, true)  // remove the method definition
    ]

    val tMethodDefinition = tgPackage.typeGraphBasicFactory.createTMethodDefinition
    tMethodDefinition.returnType = savedReturnType
    tMethodDefinition.signature = savedSignature
    tMethodDefinition.access += savedAccess

    parentClass.defines += tMethodDefinition

    println(tMethodDefinition)
  ].build

  protected def readFileToString(String path) {
    new Scanner(new File(path)).useDelimiter("\\A").next
  }

  protected def TypeDeclaration findCompilationUnits(String parentClassKey, Set<TClass> childClasses, List<
     TypeDeclaration> astChildClasses) {
    var TypeDeclaration result
    for (cu : compilationUnits.values) {
      // the just created CU can not resolve
      val firstTypeKey = "L"
            + cu.package.name.fullyQualifiedName.replace('.', '/')
            + "/"
            + ((cu.types.get(0) as TypeDeclaration).name.fullyQualifiedName)
            + ";"

      if (parentClassKey.equals(firstTypeKey)) {
        result = cu.types.get(0) as TypeDeclaration
      }

      for (child : childClasses) {
        if (firstTypeKey.equals(child.TName)) {
          astChildClasses += cu.types.get(0) as TypeDeclaration
        }
      }
    }

```

```
    return result
  }

  protected def List<MethodDeclaration> findMethodDeclarations(List<TypeDeclaration> astChildClasses, MethodSignatureTrace methodSignatureTrace) {
    val List<MethodDeclaration> astMethodDeclarations = new ArrayList

    for (childCU : astChildClasses) {
      val methodSignature = childCU.resolveBinding.key + methodSignatureTrace.signatureString;
      val types = (childCU.root as CompilationUnit).getStructuralProperty(CompilationUnit.TYPES_PROPERTY) as List<TypeDeclaration>
      for (type : types) {
        for (method : (type as TypeDeclaration).methods) {
          if (method.resolveBinding.key.startsWith(methodSignature)) {
            astMethodDeclarations += method
          }
        }
      }
    }

    return astMethodDeclarations
  }

  protected def updateASTAndSerialize(TypeDeclaration astParentClass, List<TypeDeclaration> astChildClasses, List<MethodDeclaration> astMethodDeclarations) {
    if (astMethodDeclarations.size > 0) {
      astParentClass.bodyDeclarations.add(ASTNode.copySubtree(astParentClass.AST, astMethodDeclarations.get(0)) as MethodDeclaration)

      for (methodDeclaration : astMethodDeclarations) {
        methodDeclaration.delete
      }
    }
  }

    def fire() {
    fireAllCurrent(
      PUMRule,
      "parentClass.tName" -> parentSignature,
      "MethodSignatureTrace.signatureString" -> methodSignature
      )
  }

  def canExecutePUM() {
    // get the method signature by string, then get one arbitrary match with it bound
    val parentTClass = engine.getMatcher(classWithName).getOneArbitraryMatch(null, parentSignature)
    val trace = engine.getMatcher(methodWithSignature).getOneArbitraryMatch(null, methodSignature)

    return
    parentTClass != null &&
    trace != null &&
    engine.getMatcher(possiblePUM).getOneArbitraryMatch(parentTClass.TClass, trace.trace) != null
  }
}
```

Listing 5: Pull Up Method transformation.

```
package hu.bme.mit.ttc.refactoring.transformations

import TypeGraphBasic.TClass
import TypeGraphBasic.TMethodSignature
import TypeGraphBasic.TPackage
import TypeGraphBasic.TypeGraph
import TypeGraphBasic.TypeGraphBasicPackage
import com.google.common.collect.BiMap
```

```
import hu.bme.mit.ttc.refactoring.patterns.CSCQueries
import java.io.File
import java.util.ArrayList
import java.util.List
import java.util.Scanner
import java.util.Set
import org.apache.commons.lang3.StringUtils
import org.apache.log4j.Level
import org.eclipse.incquery.runtime.api.AdvancedIncQueryEngine
import org.eclipse.incquery.runtime.evm.api.RuleEngine
import org.eclipse.incquery.runtime.evm.specific.RuleEngines
import org.eclipse.incquery.runtime.evm.specific.event.IncQueryEventRealm
import org.eclipse.jdt.core.dom.ASTNode
import org.eclipse.jdt.core.dom.CompilationUnit
import org.eclipse.jdt.core.dom.MethodDeclaration
import org.eclipse.jdt.core.dom.Modifier.ModifierKeyword
import org.eclipse.jdt.core.dom.Name
import org.eclipse.jdt.core.dom.Type
import org.eclipse.jdt.core.dom.TypeDeclaration
import org.eclipse.viatra.emf.runtime.modelmanipulation.IModelManipulations
import org.eclipse.viatra.emf.runtime.modelmanipulation.SimpleModelManipulations
import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationRuleFactory
import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationStatements
import org.eclipse.viatra.emf.runtime.transformation.batch.BatchTransformation

class CSCTransformation {
  extension BatchTransformationRuleFactory factory = new BatchTransformationRuleFactory
  extension BatchTransformation transformation
  extension BatchTransformationStatements statements
  extension IModelManipulations manipulation

  extension TypeGraphBasicPackage tgPackage = TypeGraphBasicPackage::eINSTANCE
  extension CSCQueries queries = CSCQueries::instance

  val AdvancedIncQueryEngine engine
  val String concatSignature
  val String targetPackage
  val String targetName
  val BiMap<String, CompilationUnit> compilationUnits

  var CompilationUnit targetCU

  new(AdvancedIncQueryEngine engine, List<String> childClassSignatures, String targetPackage, String
     targetName, BiMap<String, CompilationUnit> compilationUnis) {
    this(RuleEngines.createIncQueryRuleEngine(engine), childClassSignatures, targetPackage, targetName,
     compilationUnis)
  }

  new(RuleEngine ruleEngine, List<String> childClassSignatures, String targetPackage, String targetName,
     BiMap<String, CompilationUnit> compilationUnits) {
    engine = (ruleEngine.eventRealm as IncQueryEventRealm).engine as AdvancedIncQueryEngine
    transformation = BatchTransformation.forEngine(engine)
    statements = new BatchTransformationStatements(transformation)
    manipulation = new SimpleModelManipulations(iqEngine)
    transformation.ruleEngine.logger.level = Level::OFF

    this.concatSignature = childClassSignatures.join("#")
    this.targetPackage = targetPackage
    this.targetName = targetName
    this.compilationUnits = compilationUnits

    compilationUnits.values.forEach[ try { it.recordModifications } catch (Exception e) {}]
  }

  val CSCRule = createRule.precondition(possibleCSC).action [
```

```
    val tClasses = engine.getMatcher(classesOfClassListTrace).getAllValuesOftClass(concatSignature)

    val List<TypeDeclaration> astChildClasses = findCompilationUnits(tClasses)

    val firstChild = astChildClasses.get(0)

    if (targetCU == null) {
      targetCU = createTargetClass(firstChild, firstChild.superclassType)
    }

    setParentClass(astChildClasses)

    serializeCUs


    // --------------- /\ JDT transformation ------------- PG transformation \/ ---------------

    val oldParentTClass = tClasses.get(0).parentClass
    if (oldParentTClass != null) {
      oldParentTClass.childClasses -= tClasses
    }

    val targetSignature = "L" + targetPackage.replace('.', '/') + "/" + targetName + ";";
    val typeGraph = engine.getMatcher(typeGraphs).oneArbitraryMatch.typeGraph

    val targetTClassMatch = engine.getMatcher(classWithName).getOneArbitraryMatch(null, targetSignature)
    var TClass targetTClass
    if (targetTClassMatch == null) {
      targetTClass = tgPackage.typeGraphBasicFactory.createTClass
      targetTClass.TName = targetSignature

      targetTClass.package = createPackagesFor(typeGraph, targetPackage)
      targetTClass.parentClass = oldParentTClass
    } else {
      targetTClass = targetTClassMatch.TClass
    }

    (tClasses.get(0).eContainer as TypeGraph).classes += targetTClass
    targetTClass.childClasses += tClasses
  ].build

  protected def createPackagesFor(TypeGraph typeGraph, String pkg) {
    val String[] split = pkg.split("\\.");

    var previous = "";
    var TPackage previousTPackage
    for (var i = 0; i < split.length; i++) {
      var String current = previous
      if (i != 0)  {
        current += "."
      }
      current += split.get(i);

      var currentTPackageMatch = engine.getMatcher(packageWithName).getOneArbitraryMatch(null, current)
      if (currentTPackageMatch != null) {
        previousTPackage = currentTPackageMatch.TPackage
      } else {
        val TPackage currentTPackage = tgPackage.typeGraphBasicFactory.createTPackage
        currentTPackage.TName = current
        if (previousTPackage != null) {
          currentTPackage.parent = previousTPackage
        } else {
          typeGraph.packages += currentTPackage
        }

```

```
      previousTPackage = currentTPackage
    }
  }

  previousTPackage
}

protected def List<TypeDeclaration> findCompilationUnits(Set<TClass> childClasses) {
  val List<TypeDeclaration> astChildClasses = new ArrayList

  for (cu : compilationUnits.values) {
    for (child : childClasses) {
      if (cu.findDeclaringNode(child.TName) != null) {
        astChildClasses += cu.findDeclaringNode(child.TName) as TypeDeclaration
      }
    }
  }

  return astChildClasses
}

protected def List<MethodDeclaration> findMethodDeclarations(List<TypeDeclaration> astChildClasses,
   TMethodSignature tMethodSignature) {
  val List<MethodDeclaration> astMethodDeclarations = new ArrayList
  val methodSignatureTrace = engine.getMatcher(methodSignatureAndTrace).getAllValuesOftrace(
   tMethodSignature).get(0)

  for (childCU : astChildClasses) {
    val methodSignature = childCU.resolveBinding.key + methodSignatureTrace.signatureString;
    val types = (childCU.root as CompilationUnit).getStructuralProperty(CompilationUnit.TYPES_PROPERTY)
   as List<TypeDeclaration>
    for (type : types) {
      for (method : (type as TypeDeclaration).methods) {
        // match
        if (method.resolveBinding.key.startsWith(methodSignature)) {
          astMethodDeclarations += method
        }
      }
    }
  }

  return astMethodDeclarations
}

protected def CompilationUnit createTargetClass(TypeDeclaration childClass, Type superClassType) {
  val ast = childClass.AST
  val compilationUnit = ast.newCompilationUnit

  if (targetPackage != null) {
    val packageDeclaration = ast.newPackageDeclaration
    var Name packageName
    for (part : targetPackage.split("\\.")) {
      if (packageName == null) {
        packageName = ast.newSimpleName(part)
      } else {
        packageName = ast.newQualifiedName(packageName, ast.newSimpleName(part))
      }
    }
    packageDeclaration.name = packageName
    compilationUnit.package = packageDeclaration
  }

  compilationUnit.imports += ASTNode.copySubtrees(ast, (childClass.root as CompilationUnit).imports)

  val typeDeclaration = ast.newTypeDeclaration
```

```
    typeDeclaration.modifiers().add(ast.newModifier(ModifierKeyword.PUBLIC_KEYWORD))
    typeDeclaration.name = ast.newSimpleName(targetName)

    if (superClassType != null) {
      typeDeclaration.superclassType = ASTNode.copySubtree(ast, superClassType) as Type
    }

    compilationUnit.types += typeDeclaration

    compilationUnit
  }

  protected def insertMethodDeclaration(MethodDeclaration declaration) {
    val typeDeclaration = targetCU.types.get(0) as TypeDeclaration
    typeDeclaration.bodyDeclarations.add(ASTNode.copySubtree(targetCU.AST, declaration) as
     MethodDeclaration)
  }

  protected def setParentClass(List<TypeDeclaration> typeDeclarations) {
    val ast = targetCU.AST

    var Type fqn
    if (targetPackage != null) {
      for (part : targetPackage.split("\\.")) {
        if (fqn == null) {
          fqn = ast.newSimpleType(ast.newSimpleName(part))
        } else {
          fqn = ast.newQualifiedType(fqn, ast.newSimpleName(part))
        }
      }

      fqn = ast.newQualifiedType(fqn, ast.newSimpleName(targetName))
    } else {
      fqn = ast.newSimpleType(ast.newSimpleName(targetName))
    }

    for (declaration : typeDeclarations) {
      declaration.superclassType = ASTNode.copySubtree(declaration.AST, fqn) as Type
    }
  }

  protected def removeChildMethodDeclarations(List<MethodDeclaration> methodDeclarations) {
    for (declaration : methodDeclarations) {
      declaration.delete
    }
  }

  def serializeCUs() {
    val targetDir = StringUtils.substringBefore(
                compilationUnits.keySet.get(0),
                "/src/"
              ) + "/src/" + targetPackage.replace('.', '/')
    val targetPath = targetDir + "/" + targetName + ".java"

    val targetFile = new File(targetPath)
    targetFile.parentFile.mkdirs

    compilationUnits.put(targetPath, targetCU)
  }

  protected def readFileToString(String path) {
    new Scanner(new File(path)).useDelimiter("\\A").next
  }
```

```
  def fire() {
    fireAllCurrent(
      CSCRule,
      "concatSignature" -> concatSignature
      )
  }

  def canExecuteCSC() {
    val targetSignature = "L" + targetPackage.replace('.', '/') + "/" +  targetName + ";"
    val targetTClass = engine.getMatcher(classWithName).getOneArbitraryMatch(null, targetSignature)

    if (targetTClass != null) {
      return false
    }

    engine.getMatcher(possibleCSC).countMatches > 0
  }

}
```

Listing 6: Create Superclass transformation.