TÉCNICO LISBOA

An Interactive Tool for Computer-Aided Refactorization of Java Source Code

João Afonso do Carmo Rodrigues

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Luís Jorge Brás Monteiro Guerra e Silva
Eng. Artur David Félix Ventura

Examination Committee

Chairperson: Prof. José Carlos Alves Pereira Monteiro
Supervisor: Prof. Luís Jorge Brás Monteiro Guerra e Silva
Member of the Committee: Prof. António Paulo Teles de Menezes Correia Leitão

May 2017

Acknowledgments

I would like to thank to those without whom I would not had been able to finish this work: my advisors, Professor Luís Guerra e Silva and Artur Ventura, for their guidance; my parents for their hard work to provided me clothes, food, accommodations and the family comfort; my friends for the their companionship and fun times; finally I would like to thank Patrícia for making me believe that I could finish this overwhelming task.

Abstract

Code refactorings are source code changes that do not change the behavior of a program, only its structure. They are applied to make the code easier to modify and understand. Many IDEs already have prebuilt refactorings. Some of them enable the user to program his own refactorings like the Eclipse LTK framework. However this framework has a steep learning curve and does not allow for interactive development. To ease the task of creating refactorings there are scripting tools that either are standalone (Recoder, Inject/J) or use the facilities of an IDE but hide their complex interface (JTransformer, JunGL).

The current tools use DSLs (Domain Specific Languages), the language of the target program (in this case Java) or other general purpose languages like Prolog. Usually the tools that use imperative languages lack a more expressive querying, when the other more declarative languages such as Prolog or DSL's are less known languages with paradigms that are more expressive. We improve on that work by providing a transformation tool that combines both the expressiveness of logic for searching with imperative paradigm for transforming the AST. Instead of using a DSL we use JavaScript as scripting language. We evaluated our work by comparing other tools, by creating some academic transformations and by creating two transformations that were applied to the FenixEdu project's source code.

Keywords: Refactoring, Interactive, JTransformer, Java, JavaScript

Resumo

Refactorizações de código são aterações à estrutura do código sem alterar o comportamento do programa. Elas são aplicadas com o objectivo de tornar o código mais fácil de compreender e modificar. Muitos IDEs (*Integrated Development Environments*) já fazem refactorizações. Alguns permitem até que o utilizador programe as suas próprias refactorizações, como é o caso do Eclipse. Contudo a framework the o Eclipse disponibiliza para implementar as refactorizações, é díficil de aprender e não permite um desenvolvimento interactivo. Para facilitar a tarefa de criar uma refactorização existem ferramentas de *scripting* que, ou são autónomas (Recoder e Inject/J), ou estão integradas num IDE (R2, R3, JTransformer e JunGL).

As ferramentas actuais usam DSLs (*Domain Specific Languages*), a linguagem do código alvo da transformação (neste caso Java), ou liguaguens de uso geral como Prolog. As ferramentas que usam liguagens com o paradigma imperativo não permitem definir as pesquisas de formas tão expressivas como as linguagens que usam o paradigma declarativo. O nosso trabalho é uma ferramenta de *scripting* de transformações, que combina o paradigma declarativo para as pesquisas e o paradigma imperativo para as transformações. Em vez de uma DSL usamos JavaScript como linguagem de *scripting*. Avaliou-se a ferramenta, criando alguns *scripts* para transformações académicas e criando duas transformações ao código do projecto FenixEdu.

Palavras-Chave: Refactorização, Interactiva, JTransformer, Java, JavaScript

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A programmer can be seen as an executive chef that writes recipes for his cooks to prepare. For example, the queen has asked her royal chef to prepare the dinner banquet for a diplomatic visit. The meal must be composed of national dishes, although with special care for the nut allergy of the Hungarian diplomat. With only one week to prepare the banquet, the chef can not create the recipes for the entire meal from scratch. Instead he looks up its the royal archive of recipes. This archive contains the recipes created by all previous royal chefs. Each new recipe that is added to the archive may reference other recipes in order to avoid repetition. For example the chicken soup uses the chicken braising recipe, created by the first royal chef, and the vegetable stew added by the ninth chef. The easy part was choosing national dishes. The problem is that none of the national dishes in the royal archive was written to have in mind people with nut allergy. The chef knows that in most dishes, nuts can be replaced by roasted beans while keeping the flavor. However the on deserts nuts should be replaced with pumpkin seeds, or the the desert will acquire a bad taste.

This problem is similar to the problem of a programmer that has to change a program. The description of a program is usually spread over several files, that in turn are spread over several folders. These files usually use 'recipes' that are defined in other files, stored in other folders. Same as the chef, the programmer also has to adapt programs to requirements they were not designed for. This task may be performed by hand but similarly in chefs case, where one has to be careful to change all the instances of the nuts by the right replacement, the programmer has to be careful no apply right changes on the right places, or else he will add errors to the recipes.

In order to avoid the death of the Hungarian diplomat or a bad desert, there are tools that allow programmers to search and change code. The goal of this work is to implement one such tool, with the added benefit of allowing the programmer to write his own changes, and reapply them at will. In the hands of the royal cook, a tool such as this would allow him to define a general change to the recipes, for example replacing an allergenic ingredient. This way the chef can use the same change to help him prepare for the dinner with the Chinese diplomat who is allergic to garlic.

In order to keep the original recipes, the royal cook first copies the original recipe and then replaces the allergenic ingredients on the copy. However there is a lot of the text of the recipes that is repeated. If any change to the original recipe is made, for example, the new royal cook might have found that reducing the amount of salt would make it taste better, both the original and the allergenic altered recipes should be changed. In programming this problem is known as code duplication, which happens when there are two blocks of code that have similar behavior and are replicated in distinct places.

The solution to code duplication is to find the common behavior, and move it to a more general recipe that can be used to replace the duplicated code. For the cook this means that he has to write a more general recipe in function of general types of the ingredients. The concrete recipes then reference

the general recipe, and set the concrete ingredients. For example a recipe that uses nuts, would be generalized by instead of directly using a concrete ingredient such as nuts, it would reference an abstract ingredient that has the same flavor as nuts. Then the concrete recipes would refer to this general recipe and chose whether to use nuts, which results into the original recipe, or roasted beans, which results on the recipe served to the Hungarian diplomat.

Not only this restructuring of the recipe reduces the size of the recipe book, it also confines changing the amount of salt to only one file, instead of two separate files. Now the chef can improve the general recipe by changing there the amount of salt used, or add new versions of the recipe with different replacements for nuts.

This process of changing a programs structure without changing its behavior (for example generalizing a block of code to remove code duplication) is called Refactoring. It is an error prone task, that should be carefully done. Martin Fowler, on his book 'Refactoring: Improving the Design of Existing Code' [3], suggests a methodical way of applying refactorings in order to avoid the introduction of bugs, that involves the application of small changes, step by step, interleaved by compilation and testing. Because of its procedural definition, most of the Fowler's refactorings have been already automated. The most common Integrated Development Environments (IDEs), tools used by programmers to help them write code, provide this kind of automated code transformations.

However pre-written refactorings are not flexible enough to accommodate all the programmer's needs. Eclipse has even a plugin system [4] that allows programmers to write their own refactorings, to be used alongside the refactorings already provided. Being plugins, this user developed transformations, can be shared with other developers.

The royal chef now wants to expand the repertoire of French recipes. Again, to avoid creating all French recipes from scratch, he will reference recipes of a famous French chef, and then add his own touch to them. The French chef, being referenced by many other chefs, has written his recipe book in a way that facilitates other chefs to extend on his recipes. Like the royal cook, the French chef has general recipes defined in terms of general ingredients and cooking procedures, that are left blank for the other chefs to add their own touch.

Programmers often share their programs in a similar way, so that others do not repeat the work already done. Instead of recipe books they are called libraries and have an interface through which other programs interact with them. In the process of improving those libraries, some of their recipes, may change the way programs interact with them. Like the royal cook that has to change the recipes that use the french's chef foie gras, so do the programmers have to change the code that use the changed library.

Scripting transformations could solve the problem of library interface change. The library developer could write a transformation that would update the code that used the old version of the library to the the new version. The programmers that use the changed library, could then update their code using the transformation written by the library developer.

1.1 Problem Description

Through the royal chef analogy, we see that there are several reasons to change code already written. May it be ingredients that need to be replaced across files, code duplication or updating the uses of a library to a new version. These changes most often consist of error prone and repetitive steps. In those cases the change can be automated in order to make sure that all occurrences of the use of the ingredient get changed and the uses of the library get updated.

The refactorings described by Fowler [3] are already provided by most Integrated Development Envi-

ronments (IDE) such as Eclipse, Visual Studio and InteliJ. These are monolithic refactorings that are not flexible enough for some use cases. The Eclipse Language Toolkit provides a way to write new source code transformations as Eclipse plugins. These new transformations can be shared. For example if the developers of a library remove an argument of a method, they can write a transformation that removes that argument in every call of that method. They can then give that transformation to the programmers that use that library in order for them to update their code. However, writing a plugin is a task that requires some knowledge of the complex Eclipse Plugin API. Moreover to test the refactoring one would have to compile the plugin, and launch an instance of Eclipse, which further delays the process of testing a refactoring.

1.2 Goal

The goal of this work is the development of powerful environment for scripting code refactorings based on of Prolog. While JavaScript is a popular language among programmers, Prolog is not. Moreover, most programmers are much more comfortable with imperative programing (JavaScript) than declarative programming (Prolog). The environment to be developed should provide an interpreter window, that allows users to iteratively test expressions to further develop their scripts. To accomplish this it should have the following features:

- interactively build source code queries;
- interactively build refactorings on the queried source code;
- do and undo refactorings;

The environment will consist of a JavaScript runtime with search and transformation APIs. The Search API implements a mix of logic and object oriented approach, and the transformation API emulates accesses to the program AST as database accesses.

1.3 Outline

The remainder of this document is structured as follows. Chapter 2 makes a more profound exposition of the concepts behind automating refactorings. Chapter 3 presents an overview of the area of program query languages and refactoring tools, followed by a comparison of those tools and their relevance for the project. Chapter 4 describes the JavaScript interface for writing program searches and transformations.Chapter 5 gives information on the development process and the difficulties found. Chapter 6 further details the methodology to assess the success of this project. Chapter 7 presents some concluding remarks and highlights critical aspects of this work.

Chapter 2

Background

In this chapter we are going to introduce the main concepts related to program transformation. We begin by talking about the concept of programming language. Then we talk about several representations of a program, and their uses. Finally we present the concept of program transformation.

2.1 Programming Language

Processors receive instructions through binary strings, however programming in binary is a tedious task that requires a great level of knowledge about the processor's inner workings. To make it easier to program, languages were invented that were closer to the spoken English than binary. Programs called compilers read programs written in those languages that are easier for a human to understand and translate them to the binary commands that processors understand.

Early programming languages had less features. Their structure resembled the structure of commands given to the processor. As languages evolved, they started to gain more vocabulary and to check for more programming errors. The more useful those concepts were the more productive programmers became.

There were two approaches for developing a new language. The first one was to mimic the processor way of executing commands. We say that these languages are imperative, because the programs written with them are streams of instructions. As they evolved, new structure elements were introduced such as splitting code into reusable parts (like the chef that generalized the recipe with nuts). The compilers for these languages were simpler, because the concepts they had were easily translated into processor instructions. By being closer to the processor instructions, these languages allowed for greater control over the final binary code that the processor was going to execute. Which in turn enables the programmers to better optimize their program.

At the same time in those early stages of programming languages, other language developers took a different approach. Instead of directly providing computer hardware concepts to the programmer, this other family of languages, focused on mathematical concepts. These languages called declarative had the objective of being closer to the brains of a programmer, instead of the brains of a computer. They were harder to translate to the binary format processors understand and their abstractions did not allow programmers to optimize the generated code, at least as much as imperative languages did. Therefore, in most cases programs written in these more abstract languages are slower than programs written on languages closer to the hardware.

As processors became more powerful and compilers started producing ever faster programs, the need for the programmer to manually optimize his programs gradually faded. The new imperative lan-

guages that emerged abstracted further the hardware from the programmer. Moreover, the compilers of declarative languages became better at optimizing the abstract programs into faster executables. Eventually declarative languages that started as academic experiments, became mature enough to be used in production.

We make now a special note about Prolog because it is a less known language.

2.1.1 Prolog

Prolog is an example of a declarative programming language. A Prolog program is a list of facts and rules. Rules have a head and a body. They express the implication between the facts in the body and the fact on the head. If the body is true then the head is true. Both facts and rules make the knowledge base that is used by the Prolog engine to find the truth about queries made to it.

```
1 parent(luke, darth).
2 parent(leia, darth).
3 sibling(A, B) :- parent(A, P), parent(B, P).
```

Listing 2.1: Prolog program that defines the sibling relation, for the Skywalker family.

Listing 2.1 presents an example of a Prolog program. The lines 1 and 2, represent facts that we know to be true. On Line 3 we declare that for two persons to be siblings they should have the same parent.

```
1 ?- sibling(luke, S).
2 S = leia.
3 true
```

Listing 2.2: Query to find the siblings of luke.

Listing 2.2 shows a query made to the Prolog engine. A query is a fact that the Prolog engine will try to prove true or false. The program in Listing 2.1 finds that for the fact 'sibling(luke, S)' to be true, 'S' must have value 'leia'. The program does not specify explicitly how to compute a sibling for a given person. It only declares which facts must be true, to make the sibling rule true. The Prolog runtime engine has the responsibility of interpreting the code of the program and find which instructions to give the processor in order to obtain the answer.

2.2 Program Representation

Now that we presented what a programming language is we will talk about how program transformation tools represent programs. Much like spoken languages, programming languages have syntactic rules. Syntactic rules define how sentences in the language are structured. On spoken languages those rules define how to form correct sentences. In programming languages they tell how to form correct programs.

Programs are written in text files. This representation of the program as a string of characters is to unstructured. Tools like compilers extract the syntatic structure of the programs before processing them.

2.2.1 Abstract Syntax Tree

The syntactical structure of a program is represented by an Abstract Syntax Tree (AST), which is a tree where nodes are syntactical constructs. The Fig. 2.1 illustrates an example of such structure for a

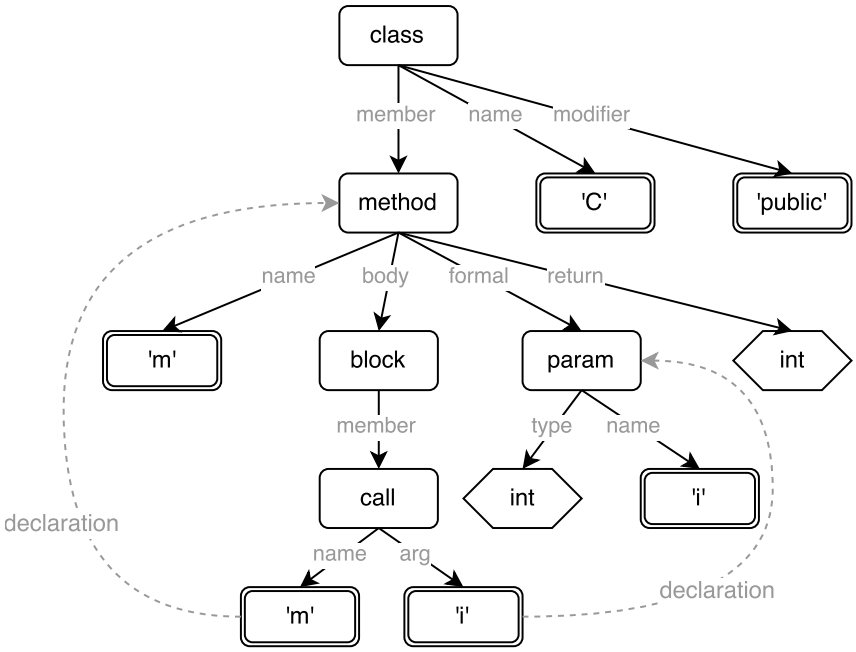


Figure 2.1: An object oriented program and its simplified AST. [1]

Java program (Listing 2.3), where the nodes represent elements of the Java language such as classes, methods and method calls. The children of a node on an AST are the attributes of the syntactical construct a node represents, for example a class node has as child a method node that, in turn, has children of its own. ASTs are used by programs such as compilers to perform static type analysis and code generation [1].

```
1 public class C {
2   public int m(int i) {
3     m(i);
4   }
5 }
```

Listing 2.3: Java program whose AST is represented in 2.1

2.2.2 Program Flow

On imperative languages, the programmer lists the instructions in the order he wants them to be executed. Sometime the order of execution of those instructions can be altered without changing what the program does (for example one can switch the line 1 with line 2 in Listing 2.4). On the Program Dependency Graph (PDG) [5] edges represent that a node must be executed before the other in order keep the meaning of a program. For example, Fig. 2.2 2.2 is the PDG for the program in Listing 2.4, where the *Entry* node is where the program starts. The *while(i<=n)* node is a conditional operator node, which means that nodes dependent on this node only execute if the condition is true. The same *while* node has a data dependency from the statement on line 1 (through variable *n*) and the statement on line 2 (through variable *i*). One of the uses of the PDGs is automatic parallelization of programs, by finding which paths in the graph can be executed in parallel. Griswold [6] uses PDGs to check for changes that can be made to the source code without changing program functionality. PDGs are also used on

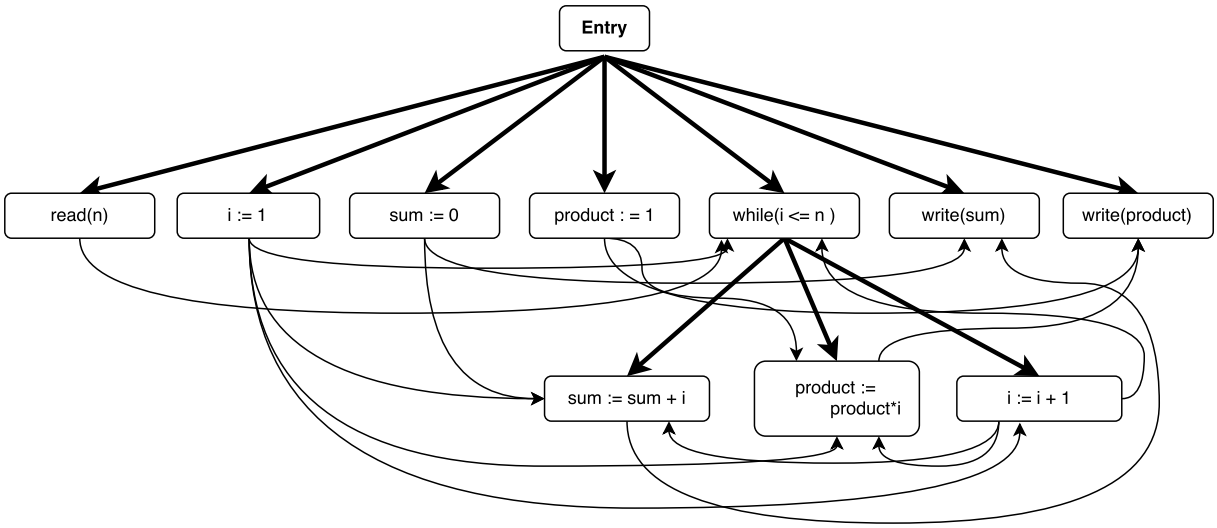


Figure 2.2: PDG for the code on the Listing. 2.4. [2]

program quering

```
1  read(n);
2  i := 1;
3  sum := 0;
4  product := 1;
5  while i <= n do
6  begin
7    sum := sum + i;
8    product := product * i;
9    i := i + 1;
10 end;
11 write(sum);
12 write(product);
```

Listing 2.4: Example program for the PDG. [2]

2.3 Program Transformation

Program transformation, as the name suggests, is the act of modifying a program into a new one. It can be used for several tasks, such as program optimization and refactoring, in which case one would want to maintain the behavior of the program (meaning-preserving transformations). Or they can be used on transformations that do not preserve behavior, like one of the transformations we present on Chapter 6 that fixes some transactions in the FenixEdu project [7].

Program transformations may have as input and output the source code or the binary code of a program. An example of source-to-binary transformation is the compilation of a program. It transforms the source code in a given language into a format such as machine code. This process is mean-preserving because the result of a compilation must have the same meaning of the source code, otherwise it would not correctly implement the language specification. Transformations such as optimizations can also be applied to the binary or intermediate representation of the program [8] (binary-to-binary). The focus of

this work is source-to-source transformations such as refactorings.

2.3.1 Program Querying

In order to change a part of a program we first must find the places where to change it. Program querying is the process of searching source code for relevant information. For example programmers may use this tool when adding a new feature to a system. To do so the programmer has to understand and find the source code related to that feature. This may not be a problem if the programmer developed the code himself or if the program is well documented. However this is not always the case. Program querying tools help programmers understand a program by providing features such as finding uses of a function or even finding architectural patterns [9].

There are many approaches to program querying. The most popular approaches use queries constructed with keywords or natural language sentences. These are easy to use for beginners. There is another type of searching tools that allow for more complex queries. Queries on these tools are specified using programing languages. Often those domain specific languages are specially developed to the end of querying source code and are called program query languages.

Program query languages allow for more expressive queries. There are languages based on several paradigms such as relational algebra, relational calculus and logic. The most successful ones are based on logic. Their declarative style makes it easier to express graph patterns which represent the abstract syntactic tree (AST) of the program.

In this work we intend to make use of a logic-based language similar to DATALOG [10] to describe queries on the AST and on the Program Dependency Graph(PDG). A detailed description of this representation of programs is described in the following section.

2.3.2 Program Slicing

Program Slicing [11] is a technique to either find the lines of code that affect a value of a variable (backward slicing), or to find the lines of code that are affected by the value of a variable (forward slicing). A slice can be computed by walking on the edges of the PDG.

The value of a variable may depend on previous instructions but not all. Some times it is hard to figure out the concrete lines of code a variable depends on. The instructions that affect the value of a variable can be on different places.

2.3.3 Refactoring

The development of a program often starts by the definition of a structure that implements its requirements. However after the structure is defined and implemented, it is common that new requirements need to be added, that do not conform to the initial structure. Most often this requires the program to be restructured, even if only slightly. However changing the structure of a program is a time consuming task, and often introduces new bugs and adds no new functionality. Because project managers are often bound by tight deadlines, the restructuring process is routinely avoided. This forces development teams to ‘hack’ the new functionality into a structure that was not designed to support it. As the development continues those ‘hacks’ accumulate and degrade the program structure. Bugs become harder to fix and new features more difficult to add. Refactoring, which is the restructuring of programs, has been shown to be the solution to avoid the erosion of the programs structure.

However as described in [12], when refactoring one must take in account the costs and risks of refactoring and consider if its benefits are worthwhile. The main cost of refactoring is time. Time is spent

on three activities: doing the refactoring, testing the changes and updating tests and documentation. These activities prevent the formation of the 'big balls of mud' [13]. There is also the risk of the refactoring process introducing new bugs that are not detected in the tests.

Much effort has been devoted to the development of refactoring tools that would appeal to programmers, always bound by tight deadlines. Such tools should help the programmer to find the areas of the code that must be changed (program query tools), find areas of code affected by the changes (program slicing) and perform the more known restructurings so that the programmer does not to manually make them (automatic restructuring). Automatic restructuring has the advantage of ensuring that the program maintains the same behavior, while not introducing new bugs. Thus two of the disadvantages pointed in [12] are mitigated: the time spent refactoring and the risk of introducing new bugs is significantly reduced.

2.3.4 Tool Aided Transformations

On [3] Fowler presents a methodology for programmers to apply a set of 72 refactoring patterns. However this manual modification of the source code does not ensure that the source keeps its observable behavior. It is also prone to the introduction of new bugs.

The advantage of having a algorithmic description of a refactoring would be to have a tool that could apply it automatically. This tool would have to check if a given change preserves the functionality of the program, avoiding the introduction of new bugs. William Griswold [6] defined a set of meaning-preserving transformations that could be made to a program dependency graph without changing its functionality. From changes on a PDG the corresponding source code transformations could be extrapolated. With this set of meaning-preserving transformations Griswold was able to implement a tool that perform automatic restructurings such as renaming and function extraction.

IDEs such as Eclipse [14] implement a large set of the Fowler's refactorings. However, as the study [15] shows, the users of Eclipse are not using the more complex refactorings. The authors of the study suggest that this is due to the undocumented and overly complex interface to use this refactorings.

2.4 Chapter Summary

In this chapter we introduced the main concepts of Program Transformation and some of its uses. There are several kinds of program transformation, but this work focuses on the source-to-source transformations, more specifically the transformation of Java programs. Therefore we started by introducing the concept of programming languages and presented the language Prolog, because it is a language that uses the logic paradigm, a less well known paradigm that we want to use on our searches.

We introduce the concept of program representation. Program transformation tools use the Abstract Syntax Tree as an intermediate representation of the program. Transformation access the AST's nodes and change them. The altered AST is then used to generate the transformed source code. Control flow and data flow gives information about the execution of the program. They can be computed from the AST and extend it with additional information.

Program querying and program slicing are also concepts related to Program Transformation. Program querying is used on program transformations to search the AST for the nodes that need to be changed. Program slicing is not used on transformation tools, but is useful for developers to find the lines of code a value depends on. We implemented a simplified version of this functionality by computing the Data Flow of the program.

Finally we explain what is Refactoring, a use case of program transformations. Refactorings can be made by hand, but IDEs such as Eclipse [14] already automate some.

Chapter 3

Related Work

As discussed previously, the focus of this work is source-to-source transformations. More specifically, a tool that allows users to program their own transformations, giving access to the AST of the program. In this chapter we are going to present the current tools used for scripting transformations.

One feature all transformation tools have is parsing the source code (on some analysis tools the user writes the analysis with regular expressions that match directly to the source code, however complex transformations require at least an AST of the code). After parsing the source code, the user must be provided a way of extracting information about the program (querying API) and an interface for applying changes to it (transforming API). The transformation tools should also generate the source code from the transformed AST.

3.1 Stratego/XT

Stratego/XT [16] is a tool set that provides a basis for writing program transformations with many applications (refactoring, program optimization, compiling, etc.). It takes as input a specification of a grammar, a rewriting strategy and a specification for pretty printing. A rewriting strategy is a list of rules (also called rewrite rules) that map a certain pattern on the AST to the a node that replaces it. The tool set includes already definitions for common languages such as Java and C++. The parser outputs a tree of terms as defined by the ATerm library. The rewritings strategies receive the tree from the parser as input and return a tree in the same specification. The rewritten tree is then passed to the pretty printer that generates the desired output. For example in the case of a refactoring, the program would receive source code in a target language and would output source code in the same language.

The Stratego language is the DSL used to write the rewriting strategies. Includes libraries written using Stratego itself which proofs that new functionality can be implemented in the language itself (similar to JunGL predicate and edge definition). The implemented functionalities include: sequential control, generic traversal, built-in data type manipulation (numbers and strings), standard data type manipulation (lists, tuples, optionals), generic language processing, and system interfacing (I/O, process control, association tables.

One of the advantages of Stratego is the ability to use concrete syntax of the parsed language in place of AST patterns. This is implemented by using the grammar specification provided to parse the target language.

Contrary to tools like JTransformer Stratego/XT does not store the AST from transformation to transformation. The tool generates a program that parses and the target code and generates the transformed code.

3.1.1 GraphStratego

GraphStratego [17] is an extension of the Stratego language to include references. These references can then be used to reduce the space used by the AST of Stratego, by replacing equal subtrees with a reference to one single instance of the subtree. This extension can also reduce the complexity of comparing equal subtrees by comparing just the pointers instead of performing a full comparison. The addition of references makes it easy to define rules that add flow info to the AST. For example, references enable one to write an AST where each node is a reference to its parent, or method calls can have pointers to their definition. This extension also introduces constructs to deal with no-termination issue when transversing graphs with cycles. References may turn the tree into a cyclic graph. For example edges of the flow of a loop would introduce a cycle to the graph. GraphStratego deals with this issue my providing several primitives that avoid visiting the same node twice.

3.2 Rascal

Rascal [18] is a domain specific language for source code analysis and manipulation (SCAM). It is a combination of many concepts into a single language. On one side, Rascal combines the concepts of previous source code transformation efforts, such as Stratego/XT, with programming language concepts like first order functions, anonymous functions, immutable data types and comprehensions for lists, sets and maps. This means that besides the AST rewriting, already provided by Stratego, it also includes grammar descriptions and syntax for pretty printing, which in Stratego/XT toolkit are provided by other DSL's. Note that Rascal has deprecated rewrite rules, however it still has visit expressions, expressions similar to rewrite rules that allow the user to write a visitor to a given node. The main difference is that on a visit expression, the rules do not necessarily produce an AST node. Instead they can perform a void action like printing text to the output. Listing 3.1 presents an example of a visit expression that replaces '==' with a method call to equals.

```
1 let transformedAST = visit(ast) {
2   case i:infix(expr1,"==", expr2) =>
3     \methodCall(false, receiver, "equals", [arg])
4 };
```

Listing 3.1: Visit expression example

Another feature of Rascal is pattern matching which makes it easier to inspect the internal structur of a value. Pattern matching is used in Listing 3.1 line 2 to do two things in one go. The first is to check if a node represents a comparison expression. The second is to extract the values for the left and right expressions.

Queries can be created using comprehensions, a way of filtering and mapping the elements of a collection to a new collection. On line 1 of Listing 3.2 is an example of comprehensions in Rascal, that was extracted from [18], which results in a list of squared numbers where the numbers divisible by 3 are squared. The operator (<-) denotes an iteration of the collection at the left, and each element will be bound to the variable at the right. When all boolean expression left of the (|) are true, the expression at the right of the operator (|) is evaluated and added to the new collection. A comprehension can contain any number of collection iterations as well as boolean expressions as can be seen on line 2.

```
1 { x * x | int x <- [1..10], x % 3 == 0}
```

```
2 { (x,y) | int x <- [1..10], int y <- [1..10], x > 1,  x == y}
```

Listing 3.2: List comprehensions in Rascal.

Comprehensions over collections of nodes can be used to create arbitrary complex queries, because they allow the use of any expression valid on the language such as pattern matching and calling functions. Transformations on an AST are made through the visit construct: similar to the rules of Stratego, a visit defines a set of patterns and actions to be run when the pattern is matched. The main difference is that the user as the ability to chose the sub-trees transversed.

The broad range of applications Rascal is targeted for is also its weakness. While it implements a wide variety of concepts to perform analysis, it lacks the generation of source code from the AST. Rascal was first presented in 2009 [18] and in 2012 some of the authors presented a paper [19] that implemented a Visitor-to-Interpreter refactoring, where the resulting source code is generated by the script instead of relying on the tool to generate the source code from the changed AST.

3.3 JTransformer

JTransformer [20] is an Eclipse plugin that uses Prolog as a scripting language. It uses the JDT compiler of Eclipse to extract the AST of the source code and stores it in a Prolog factbase where each node of the AST is a term. This plugin updates the factbase every time the project is built. JTransformer provides a graphical interface via Eclipse, a debugging view, to better visualize the factbase (useful to debug transformations that changed the AST in the wrong way by, for example, writing wrong values into node facts). Through the graphical interface of Eclipse it provides a console to interact with the factbase. That console is connected to the Prolog engine SWI-Prolog [21] responsible to interpret the Prolog part of the plugin. This way one can query the AST by inserting Prolog queries into a SWI-Prolog console.

The transformations are made through an implementation of conditional transformations (CTs) [22, 23]. A conditional transformation is a ternary tuple as in Listing 3.3.

```
ct(<transformName>(<parameters>), <precondition>, <transformation>)
```

Listing 3.3: Conditional transformation definition.

The first element of the tuple sets the name and defines the parameters of the transformations. The precondition element is the query that checks that the transformation can be applied. The simplest form of precondition is to check the existence of the nodes to be modified. When running the CT the Prolog engine will unify the precondition with the facts on the database effectively fetching the values that turn the precondition true. Therefore checking the existence of a node will also fetch it, making it available to be used on the transformation part of the CT. If the transformation requires the creation of new nodes, then the precondition query must also create the id's for them. Queries are not allowed on the transformation element of a CT. It must be a list of operations to be applied on the AST such as deleting nodes or changing attributes of existent nodes. This kind of operations cannot be used outside the transformation part of a CT. JTransformer also provides a commit command to save the transformations, and a rollback command to undo all transformations made since the last commit.

3.4 GenTL

GenTL [1] is a high level approach to logic meta-programing. Like JTransformer its semantic base is first order logic but instead of using Prolog it uses a DSL that has primitives to describe program transformations. It allows the use of concrete syntax both to search the AST and to create new subtrees. It also provides operators to combine transformations.

3.5 JunGL

JunGL [24] is a domain specific language to write refactorings for different languages of the Microsoft .NET framwork [25]. Searching is done recurring to logical concepts. Besides the predicates provided by the language, users can also define their own predicates. In order to extend the AST with information such as control flow, one can define new edges using path queries. A path query is a way of writing a predicate using AST edges. For example `[a]parent[b]` is true if there is an edge `parent` from the node `a` to node `b`.

New edges can be defined using other preexisting edges and predicates. Listing 3.4 is an example extracted from [24], which presents a definition of an new edge on the AST called `treePred`, which connects a node either to its predecessor on a list, or to its AST parent. This edge can then be use to define the `lookup` edge that links variable nodes to their definitions.

```
let edge treePred n -> ?pred =
  first ( [n]listPredecessor[?pred] | [n]parent[?pred] )

let edge lookup r:Var -> ?dec =
  first ( [r]treePred+[?dec:VarDecl] & r.name == ?dec.name )
```

Listing 3.4: Example of a definition of an edge in JunGL.

User defined edges are lazy which means that are only computed when needed, avoiding computing them for the entire AST. The edges are automatically updated if the needed, when a transformation makes them outdated. For the implementation of the query system JunGL uses a special implementation of Datalog that guarantees that the results of a path query are returned from the shortest path match until the the longest path matched, hence its name Ordered Datalog.

While searching in JunGL is done with a logic paradigm the transformation is done on an imperative paradigm. JunGL uses list comprehension to retrieve results for the predicates (see Listing 3.5). In JunGL when used on predicates, comprehensions are called streams. Streams are lazy so that new results of the query are calculated as needed. Node objects returned from the streams can be changed imperatively just like the attributes of c structs.

```
{ (?x,?y) | sibling(?x,?y) }
```

Listing 3.5: Definition of a predicate in JunGL. [24]

The user can also define new predicates that are used to define edges and to retrieve information from the AST. The code in Listing 3.6 defines the predicate *sibling* that is true if *?x* and *?y* declare a variable with the same name but are different declarations.

```
let predicate sibling(?x,?y) =
  [?x:VarDecl] & [?y:VarDecl] & ?x != ?y & ?x.name == ?y.name
```

Listing 3.6: Definition of a predicate in JunGL. [24]

3.6 Recoder

Recoder [26, 27] is a Java library that provides functions to parse, analyze/transform, and regenerate Java source code. The main difference from other tools is the absence of special syntax to support queries. To search all nodes with a given condition, one has to iterate over the AST node by node and test whether it fits the desired conditions. Despite the complexity of querying the AST, the nodes have helper methods that provide precomputed information about the AST. For example a method that returns the next statement to be executed (control flow information), and methods that provide the declarations for a given variable or method.

Transformations in Recoder are a matter of accessing the right object and call its getters and setters. How ever one must inform a global object that stores the history of changes to be applied. This object also provides a way of rolling back transformations.

Recoder is not an interactive tool, but a library. Each transformation or analysis is compiled to a different program, and parsing information is not stored between runs. In order to be more efficient it has a lazy approach to parsing, i.e. it only parses the files as needed.

3.7 Inject/J

Inject/J [27, 28] is a scripting language implemented on top of Recoder. It still lacks an interactive approach: each analysis or transformation is a single program. However it abstracts several details from the programmer while providing better support for concrete syntax (allows users to describe AST patterns with code in the target language). It provides the concept of patterns, which is a way of defining parameterized searches. The definition of a search using a pattern is still very similar to the imperative paradigm used in Recoder, but has the advantage of encapsulating the computation of the condition into a function call. Transformations also use the imperative paradigm used in searches but the responsibility of updating the global transformation history is left to the language interpreter.

An Inject/J program has two kinds of files. The files that define scripts and the main file that has configuration information such as the input and output directories and specifies which scripts to run and in which order to run them. Scripts can have analysis and or transformations that are modular and sequentially combined on the main file. This modular approach makes Inject/J more flexible than Recoder, by allowing the composition of different analysis and transformations without having to recompile them.

3.8 Eclipse Language Toolkit

The Language Toolkit (LTK) [4] is an API provided by Eclipse to program refactorings and adding them to the IDE's refactoring menu i.e. the new refactorings can be accessed from the mouse right click menu. In order to isolate the code in the refactoring from the rest of the environment, each refactoring is a plugin, which has the added advantage of sharing a refactoring being as easy as sharing a new plugin. However to develop a refactoring in Eclipse one must learn the complex API necessary to create new plugins, besides having to run a new instance of Eclipse in order to test the refactoring.

LTK provides integration with the graphical interface of the IDE, enabling the creation of wizards for the parameterization of the refactorings, as well as creation of progress monitors that inform of the current progress of the refactoring. There are hooks for pre- and pos-conditions that are used by the toolkit know the opportune moment to inform the user any of the conditions is not met. For example by checking the pre-conditions at the end of collecting the user parameters, the toolkit can prompt the user to correct the parameters. The pos-condition checks the result of the transformation before regenerating the source code.

3.9 Comparative Analysis

	Search	Transform	Program Flow	Concrete Syntax	REPL	Pretty Printing
StrategoXT	Graph patterns	Rewrite rules	Lib	Yes	-	-
Rascal	Comprehensions and Graph patterns	Rewrite rules	Refs	Yes	Yes	Lib
JTransformer	Logic	Logic	Refs	Yes	No	Yes
GenTL	Logic	Logic	Refs	Yes	-	-
JunGL	Logic	Imperative	Lib	No	-	-
Recoder	Imperative	Imperative	Full	Yes	No	Yes
Inject/J	Imperative	Imperative	Full	Yes	No	Yes
Eclipse LTK	Imperative	Imperative	Refs	No	No	Yes

Table 3.1: Table comparing the source code transformation tools presented.

Several tools targeting program transformations have been presented. Table 3.1 summarizes the comparison between them. The first two columns of the table show which paradigms the tools use for searching and transforming. The 'Program Flow' column lists the kind of support the tool has for control flow and data flow. The lines with 'Lib' indicate that the tool has support for adding the control flow to the searching model, however it must be implemented on a library or by the user; the tools with 'Refs' are the tools that add references to declarations (for example there is a reference on a node that represents the use of a variable, to the declaration of that variable); tools with 'Full' support of program flow are those which besides adding references to the declarations also adds information about the the order of execution of the statements (for example which statement is executed after a given one). The concrete syntax column indicates whether the tool allows the use of snippets of the target language to describe AST subtrees, or if the user has to write the subtrees using the node model. The 'REPL' column indicates whether the tool provides an interpreter window with an read-eval-print-loop. 'Pretty Printing' column indicates whether it is provided by the tool ('Yes'), or if the tool facilitates pretty printing, but it still requires work from the user to implement it ('Lib').

Both Rascal [18] and Stratego/XT [16] use the concept of graph patterns to describe AST templates for matching. On Stratego this is the only way to get a node of the AST. On Rascal the AST nodes can be further grouped in sets and filtered or changed through comprehensions. They are also the only tools presented that use rewriting rules as the way to alter the AST. Rascal provides more control to the application of rewriting rules with their visit expressions. Rascal only tries to match rules with the nodes on a subtree. Both allow the use code in the target language to describe AST patterns. While Rascal provides some informations about the flow provided by Eclipse's JDT, Stratego, through its language

extension GraphStratego, allows the addition of program flow edges to the AST. We did not have access to the tool so we do not know if the program flow was implemented on a library or if the user had to define it. Not having access to Stratego also turns it impossible to know if it provided an interactive interpreter, and if the pretty printing for Java was provide or if the user had also to implement it. On the case of Rascal we know that it provides an interpreter console and that it does not provide Java pretty printing out of the box.

JTransformer [20], GenTL [1] and JunGL [24] were designed for scripting refactorings, use the logic paradigm for searching. JTransformer because it stores the AST as collection of facts, allows the user to search the AST like querying a Prolog factbase. GenTL implemented over JTransformer adds the ability to use concrete syntax instead of forcing the user to know the API of the AST. JunGL's path queries are more expressive than JTransformer's Prolog queries, however it does not provide support for concrete syntax. For changing the AST both JTransformer and GenTL use the logic paradigm, but JunGL uses an imperative approach. In JunGL nodes are created like data structures on other imperative languages such as C: a node can be seen as a struct and the edges as the strutc's fields. The only difference is that JunGL provides simpler syntax to create AST subtrees. Still GenTL by providing concrete syntax to create subtrees, simplifies this process. JTransformer and GenTL, that was implemented on top of it, provides the program flow information from JDT's parser. JunGL on the other end allows the user to add edges to the AST that provide flow information, which can be accessed through a library. Of the three, JTransfomer is the only that is available for public use.

Recoder [26, 27] and InjectJ [27, 28] differ the most from other tools, with their object oriented paradigm coupled with concrete syntax support. Because the most popular languages are object oriented, they may be more accessible to a larger audience of programmers. They show how verbose a pure imperative approach can be. To find the right places on the AST one needs to explicitly iterate over the tree and imperatively check the structure of the nodes until the right places to apply the transformation are found. These are still better than Eclipse's LTK because they unburden the programmer from the need to learn the complex API's, compile faster and do not require the launch of a new Eclipse instance to test the transformations. However the absence of an interactive console makes it overly complex to incrementally refine searches.

Chapter 4

Proposed Approach

The current tools for program transformation either require the user to learn a new language, do not provide a good search API, do not provide a interactive console, or even require you to learn complex plugin APIs. Our goal is to implement a tool that: uses a widely known language, has an expressive API for searching, has an interactive console. As scripting language we propose JavaScript because it is widely known and can be integrated with Java through the Nashorn [29] script engine. We wanted to integrate JavaScript with Java in order to use the JTransformer plugin for Eclipse. We developed an Eclipse plugin that provides a console interface for the user to insert JavaScript commands. We use the JTransformer's searching and transformation capabilities such as AST creation and source code pretty printing and focus the work on the APIs for transformation. Another advantage of JTransformer is that it uses the logic paradigm which makes it easier to provide logic for searches on our tool.

In this chapter we are going to present our solution. First we present the architecture of our Interpreter plugin. Then we present the Search and Transformation APIs we developed in JavaScript that act as a wrapper to the JTransformer plugin.

4.1 Architecture

There are four core functionalities that all tools that allow the scripting of refactorings provide:

1. source code parsing
2. an API to query the source code
3. an API to transform the source code
4. generation of the source code from the modified AST

We are going to use the JTransformer plugin that implements these functionalities and provide a JavaScript interface for the user to access them. The main focus of this work is the development of just the imperative layer on top of the query and transform APIs. Both search and transformation interfaces of JTransformer are used just to interact with JTransformer AST model. Another benefit of using JTransformer is that it already has a transaction model for transformations, which allows users to rollback a change, a useful feature that enables one to test a transformation incrementally.

To interpret the JavaScript input we used the Nashorn engine. It is the JavaScript engine include in Java JDK 8 which allows calling Java methods from JavaScript code.

The APIs are implemented in JavaScript to make use of its flexibility. The ability to add methods to objects at runtime is used to generate both the Search and Transformation APIs from the AST meta-model

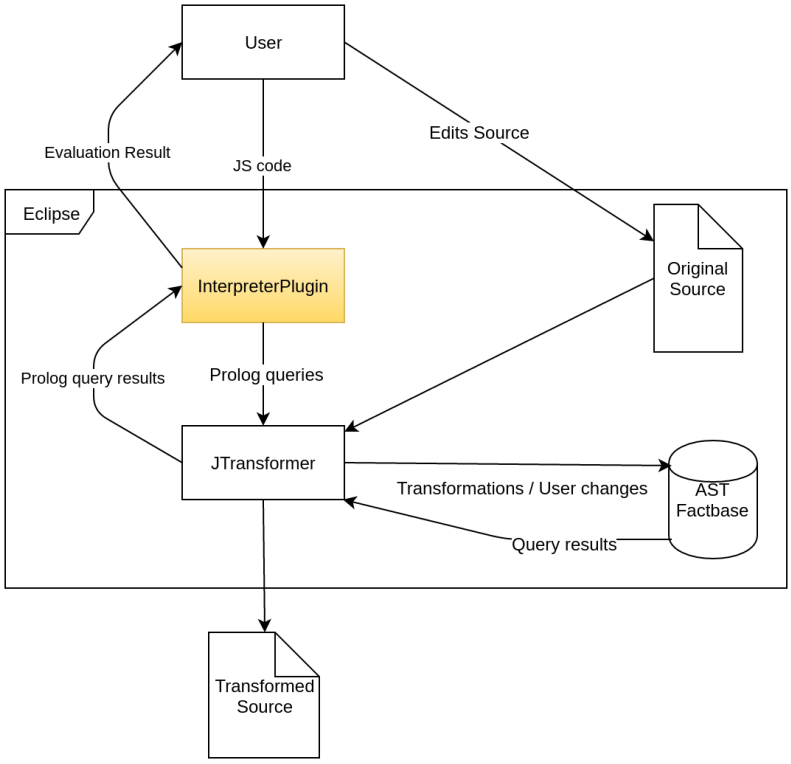


Figure 4.1: Diagram of how the our interpreter plugin interacts with the user and JTransformer.

provided by JTransformer. This process makes it easier to adapt to newer versions of JTrasnformer, and removes the boilerplate necessary to define a different class for every node type. Moreover the dynamic types of JavaScript make it faster to prototype new approaches to the API implementation.

On Figure 4.1 we can see a diagram of the interaction between our interpreter plugin, the user and JTransformer. Our plugin reads the JavaScript commands inserted by the user, evaluates them and prints the results. The evaluation of the JavaScript commands may result in calls to the JTransformer plugin methods to run Prolog queries. In response to those queries, the results are sent to our plugin.

In this diagram is also represented the interaction between the user, the source code being transformed (Original Source) and JTransformer. The user can edit directly the source code. JTransformer creates a factbase from the source code and keeps it updated as the user changes the source code.

Firgure 4.2 is a diagram of the internal structure of the interpreter plugin. The Interpreter includes the Eclipse Console, the Nashorn engine, the JavaScript source, the Prolog source and the RefactoringApi.

We developed the highlighted modules in the diagram: JavaScript source, Prolog source and RefactoringApi. The JavaScript source contains the Search and Transformation APIs. The Prolog source contains the control and data flows implementation, and some other predicates that make it easier to interact with JTransformer. The RefactoringApi, implemented in Java, sends the queries to the JTransformer Plugin. The results of these queries can be received synchronously or asynchronously. The RefactoringApi hides the complexity of the JTransfomer API for asynchronous queries and provides an interface to access the current time, which is used to chronometer the searches.

At the initialization phase, the Interpreter plugin creates an instance of the class RefactoringApi which is then added to the context of the Nashorn engine. Moreover, the JavaScript source of the plugin is loaded into the the Nashorn engine, making the APIs for searching and transformation available at the global context and the Prolog sources are loaded into the JTransformer plugin.

The user inserts JavaScript commands in the Eclipse console. The console is simply a way of reading

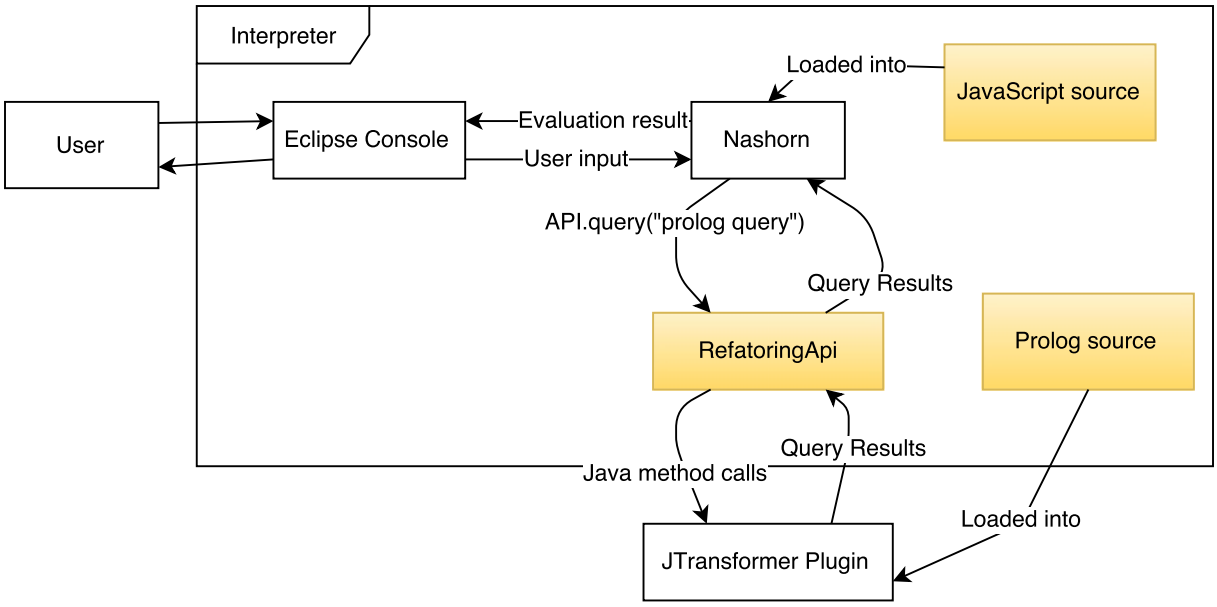


Figure 4.2: Interpreter diagram

input as strings from the user. The inputs are then evaluated by the Nashorn engine, which sends the results to be printed on the Eclipse console.

When the evaluation of the JavaScript command requires a query to the JTransformer plugin, the RefactoringApi is called, which in turn calls the JTransformer plugin methods. The result of the query is passed to the Nashorn engine through the RefactoringApi.

4.2 Search and Transformation APIs

In this section we present the JavaScript APIs for performing searches and transformations to the AST, which is the main contribution of this theses. These APIs will be gradually introduced while creating a script that replaces string comparisons that use == instead of calling the `equals` method. This example is proposedly simple, because the main focus of this chapter is to present the APIs. The capabilities of the APIs were tested with more complex scripts, which are presented in Chapter 6.

There are common mistakes that an untrained programmer make that are a result of some misunderstandings about a language. For example in Java, comparing a String with the equality operator ==, instead of calling the equals method, may result in unexpected behavior. This happens because the compiler replaces all the string literals with instances of the String class. If two string literals represent the same sequence of characters the compiler will use the same String instance to represent them. Comparing those values with == will return true because they point to the same instance (see Listing 6.7.b). However if a String is only known at runtime (Listing 6.7.c) a new instance will be created, which will result in the == to return the unwanted value. To avoid such errors, instances of the String class should be compared using the `equals` method, which is the conventional way of computing the equality of two objects in Java.

```
a. String command = "run"; // initialization from literal

b. command == "run"  // true

c.
   command = getLine(); // returns the string "run" inserted by the
      user
   command == "run" // false
```

Listing 4.1: Java Strings expressions

For our first example we write a script that turns the string comparisons that use the ==, on the source code in Listing 4.3, onto calls to the `equals` method turning it into the code in Listing 4.4. First we will introduce the API for searching while wringing the query to find the parts of the AST that we want to modify. Then we will introduce the API for changing the AST.

In order to simplify the transformation we do not take into consideration the possibility of one of the arguments of == having null value. Calling method equals on such value would result in a RuntimeException being thrown.

4.2.1 Searching Compares

The most basic search is one that searches for all nodes of a certain type, for example class nodes. The ones that we are interested in are operation nodes, which represent the use of operators such as +, - or ==. Searches of this kind have the form Listing 4.2.a. For example running the search of Listing 4.5.a on source code that we are transforming (Listing 4.3), would produce the results presented in Listing 4.6. When presenting the results of searches we are going to present interactions with the console of our tool. The characters :> represent the prompt. Code on the same line as :> are the JavaScript commands inserted. The results of a JavaScript command are presented on following lines. The information about the source location of a node is comprised by the full qualified name of the class followed by the method it is in. In this case the class `StringCompare` is defined on package `codesmell`, and the method the results were found was the `smellyCompare`.

```
a. _<node type>()  // basic search functions
b. <node search>.<attribute>(<value>) // filter by attribute value

// search for the attribute instead of the parent
c. <node search>.<attribute>()

d. <node search>.<attribute>(
      <attribute search> ->
               <attribute filter>
   )
```

Listing 4.2: API for search creation

The underscore is there to avoid possible conflicts between the JavaScript keywords when one wants to search for nodes like `if` (Listing 4.5.b) or `for` (Listing 4.5.c). Calls to those functions return objects

that represent a search, not the nodes themselves. The Prolog query for a search is only generated and sent to JTransformer during transformations or on the console, as it will be described further ahead, on Section 4.2.2.

```
1  package codesmell;
2
3  public class StringCompare {
4
5   public String oo() {
6    return "oo";
7   }
8
9   public void smellyCompare(){
10   String a = "a";
11
12   if(a == new String("a")) {
13    System.out.println(true);
14   } else {
15    System.out.println(false);
16   }
17
18
19   if("f" + oo() == "foo") {
20    System.out.println(true);
21   } else {
22    System.out.println(false);
23   }
24
25   // control code
26   int b = 1;
27   if(b == 1){
28    // do nothing
29   }
30  }
31
32 }
```

Listing 4.3: Source for the == to `equals` example

```
1  package codesmell;
2
3  public class StringCompare  {
4
5   public String oo() {
6    return "oo";
7   }
8
9   public void smellyCompare() {
10   String a = "a";
11
12   if((a).equals(new String("a"))
13   ) {
14    System.out.println(true);
15   } else  {
16    System.out.println(false);
17   }
18
19   if(("f" + oo()).equals("foo"))
20   {
21    System.out.println(true);
22   } else  {
23    System.out.println(false);
24   }
25
26   // control code
27   int b = 1;
28   if(b == 1)  {
29    // do nothing
30   }
31  }
32 }
```

Listing 4.4: Output for the == to `equals` example

```
// finds all nodes of type operation such as '+', '-' or '=='
a. _operation()

b. _if()
c. _for()

// finds operation nodes that represent '==' calls
d. _operation().op('==')

e. _operation().op('==').arg().ofType( _class().name('String') )
```

Listing 4.5: Examples of the use of the Search API on 4.2

```
:> _operation()
codesmell.StringCompare.smellyCompare(): line 10
  a == new String("a")
codesmell.StringCompare.smellyCompare(): line 16
  "f" + oo() == "foo"
codesmell.StringCompare.smellyCompare(): line 16
  "f" + oo()
codesmell.StringCompare.smellyCompare(): line 22
  b == 1
printed 4 of 4 results.
:>
```

Listing 4.6: Application of `_operation()` to Listing 4.3

Now that we know how to find operations we need to specify that we only want those operations whose operator is `==`. In order to achieve this we use the methods for specifying node attributes (Listing 4.2.b). Like in Listing 4.2.d, calling the method `op` with `==` as argument on the search for `operation` nodes, selects only the equality nodes.

The next step is to search only the comparisons that have arguments of type String. The arguments are attributes of the operation nodes, which can be accessed with getter like methods (Listing 4.2.c). To select only nodes whose expressions are of type String we use the helper method `ofType`. in Listing 4.5.e we present the query that finds all arguments for == nodes of type String. The result of this search is in Listing 4.7. For simplicity we assume that 'java.lang.String' is the only class named 'String' in the project. Note that operation nodes have a list of arguments, but the getter is in the singular (`arg`) instead of being in the plural (`args`). In cases where the attribute is a list the search returns a result for each element of that list.

```
:> _operation().op('==').arg().ofType( _class().name('String') )
codesmell.StringCompare.smellyCompare(): line 16
  "f" + oo()
codesmell.StringCompare.smellyCompare(): line 16
  "foo"
codesmell.StringCompare.smellyCompare(): line 10
  a
codesmell.StringCompare.smellyCompare(): line 10
  new String("a")
printed 4 of 4 results.
:>
```

Listing 4.7: Application of `_operation().op('==').arg().ofType( _class().name('String') )` to Listing 4.3

However, what we really want are the operation nodes, not their arguments, so that we may replace each of them with a call to `equals`. The problem here is that we want to apply filters to the attributes of the node we are searching for. It is for this case that we use the form in Listing 4.2.d. By passing a function to the selector method, one can call selector methods on it and return the resulting search. The Listing 4.8 shows the final search and the results.

```
:> _operation().op('==').arg(function(arg) {
     return arg.ofType( _class().name('String') );
   })

codesmell.StringCompare.smellyCompare(): line 18
  "f" + oo() == "foo"
codesmell.StringCompare.smellyCompare(): line 12
  a == new String("a")
printed 2 of 2 results.
:>
```

Listing 4.8: Search that finds all the comparisons between arguments of the type String in Listing 4.3

4.2.2 Replacing with equals

Transformations are similar to transactions. The AST would be the database, queries are made to access the AST and return objects that represent AST nodes. Those objects act as proxy objects, such as POJOs of Hibernate or Records of ActiveRecord, which can be used to read and change node attributes. Until the transformation commits,the attribute changes are only seen by the nodes accessed during that transformation, and the AST stays unaltered. We will refer to those proxy objects also as records.

The Search API described in the previous section is used to write the queries. It is on the context of a transformation that the searches are executed, to produce the objects that reify AST nodes. The queries are also executed by the interpreter, whenever the evaluation of the user input results on a search.

Besides records that represent nodes in the AST, there are also those that represent new nodes. Like the searched records, creation records are also produced in the context of a transformation. The trans-

formation object keeps track of the searched and created records, such that when the transformation is committed, it is able to create a query that updates the AST.

After this introduction to the transformations, it is time to resume writing the transformation that replaces the == with a call to `equals`. First we will use the search from the last section, to obtain the records that represent the ==. Then for each of them we will create a new node representing a call to `equals`.

```
:> var compareSearch = _operation().op('==').arg(function(arg) {
     return arg.ofType( _class().name('String') );
   })
:> var transform = new Transform()
:> var compares = transform.search( compareSearch )
```

Listing 4.9: Get the records for the desired == operator

```
:> compares[0].getOp()
'=='
:> compares[0].setOp('!=')
:> compares[0].getOp()
'!='
:> transform.apply()
:> rollback()
```

Listing 4.10: Example of transformation. It sets the operator of an operation node to `!=`

In Listing 4.9, we see the interaction that fetches the records for the comparisons between Strings. The `search` method returns an array with the results. If the search yields no result the array is empty. Records have getter and setter methods for their attributes. An example an interaction with the console of our tool using them, can be seen in Listing 4.10. It shows an example of a simple transformation that changes the operator from == to `!=`. Note that the change is only applied after the call of the commit method named `apply` and the call of `rollback` that reverses the previous transformation.

Same as records for already existing nodes, the records for nodes that will be created are requested through the transformation object. The transformation object has a different method for each kind of node.

```
:> var callToEquals = transform.createCall()
        .setReceiver( compares[0].getArg(1) )
        .addToArgs( compares[0].getArg(2) )
:> compares[0].replaceOnParentBy( callToEquals )
:> compares[0].remove()
```

Listing 4.11: Replacing operator == with `equals`. Note that the receiver of the call will be the left argument of ==. The right argument of == will be passed as argument to `equals`.

Listing 4.11 presents an interaction that would replace one comparison by a call to `equals`. First thing to note is the chained call of the setters. In order to help initialize the created nodes, the setter methods return the receiver of the call, i.e. the record. This example also demonstrates the use of manipulation of list attributes. In the case of the operator == node, it demonstrates the use of indexed getters to read the left and right arguments. In the case of the call node, the `addToArgs` is an example

of a method to change list arguments.

Creating a node is not enough to produce a change visible in the source code (although it would change the factbase). The new nodes must be added to the AST either by adding them to existing nodes, or by replacing the existing node by new ones. The last interactions shown in Listing 4.11, are an example of a node replacement. The old node is now detached from the AST and can be signaled to be removed from the factbase by calling `remove`. This last step concludes the example. Replacing the other string comparisons is just a matter of iterating over the `compares` array and creating an `equals` call for each one. At the end, a call `transform.apply()` method to will commit the transformations.

Table 4.1 lists and describes the methods of the search objects, Table 4.2 does the same for the records, Table 4.3 describes the methods specific to the records that represent a node on the AST and finally Table 4.4 describes the methods of the transformations. The string 'attribute name' is a place holder for the attributes of the node that is represented by the search or record. Those methods are loaded from the JTransformer meta model, which is in the JTransformer's Java PEF page [30].

4.3 Chapter Summary

In this chapter we presented the architecture of our solution and the JavaScript APIs for transforming and searching. We presented how our Interpreter plugin interacts with JTransformer and the user and how its internal modules interact with each other. The JavaScript APIs which are loaded into Nashorn generate Prolog queries and send them to the JTransformer plugin through the RefactoringApi.

Search method	Description
'attribute name'()	•Results in a search object that represents the value of the attribute.
'attribute name'(String — Search)	•Results in a search where the attributes will have the value passed as parameter. It accepts either a String or a search object.
'attribute name'(function(Search) : Search)	•Calls the function with a search for this attributes value. The resulting search will have the same node type as the original.
in(Search)	•Selects from the original search the nodes that are in the tree of the argument search.
enclosing (Search)	•Filters from the original search the nodes that are ancestors of the argument search.
ofType (Search)	•Filters from the original search the nodes whose expression in Java is of type passed as argument.
modifier()	•Returns a search that will find the modifiers of the original search. Only useful for methods, classes and fields.
modifier (string)	•Selects the nodes that have the modifier passed as argument. Only useful for methods, classes and fields.

Table 4.1: Methods of the search objects.

Record method	Description
set‘attribute name’(Record — String)	•Sets the attribute of the record to the given value.
get‘attribute name’()	•Returns the value of the attribute of the record. The first time it is called the AST is consulted to retrieve the value.
get‘attribute name’(int)	•For attributes whose values are lists. Returns the value at the given position of the list.
addTo‘attribute name’(Record)	•For attributes whose values are lists. Adds the given value to the list.
removeFrom‘attribute name’(Record)	•For attributes whose values are lists. Remove the given record form the list.
setIs‘relation name‘(boolean)	•To add certain modifier like ”isInterface”.
getIs‘relation name‘(boolean)	•Returns true if the record is in the given JTransformer relation. For example `classRecord.getIsInterface()` returns true if the class is an interface.

Table 4.2: Methods of the transformation records.

SearchRecord methods	Description
remove()	•Removes the node represented by the record from the AST.
deepRemove()	•Same as `remove()` but it also removes all the nodes that reverence it.
replaceOnParentBy(Record)	•Replaces the search record’s place in AST with the one passed as parameter.

Table 4.3: Methods that only records that represent existing AST nodes have.

Transform methods	Description
search(Search)	•Returns the AST values the search matches with.
searchOne(Search)	•Finds only the first result without waiting for the rest.
apply()	•Applies the changes to the AST.

Table 4.4: Methods of the transformations.

Chapter 5

Implementation

This chapter describes the implementation steps of our tool. We start describing the implementation steps of the Search and Transform APIs. Then we highlight how the order of the Prolog terms in the query can affect the size of the Prolog engine search space. We then proceed to describe our console interface. Finally we detail the implementation of the control flow and the data flow.

In this section we present the implementation of the Search and Transformation APIs. We start by introducing the core functions of the implementation, which are used to change a node attribute. Then we describe the implementation of the Search API in two steps. First we describe how it manages Prolog variables, and then how it was improved to combine searches. Finally, we present the implementation of the Transformation API: first with the generation of a single Prolog query for each transformation and then an improved version that uses an ORM approach.

5.1 Changing Node Attributes

In order to change a node attribute we must first fetch it from the AST. The core function of our implementation is the function `query` that sends Prolog strings to the JTransformer plugin. For example the function call in Listing 5.1 would result in an array with all the class ids and their respective names. This function is the basis of the functionality of the API.

```
query("classT(Class,_,Name,_,_)")
```

Listing 5.1: Example of the `query` function.

The step after providing a means of searching the AST, was to allow the AST to be changed. On JTransformer the transformations are made through Conditional Transformations (CTs). As mentioned before, CTs have three parts: the name, the conditions and the transformation. Each time we want to change the parameter of a node define a transformation specific for that change, called `setparam`. The definition for this transformation is generated depending on the type of node and the parameter we want to change. The function `applyCtToSetParam(nodeId, paramName, value)` implements that behavior. It generates and applies `setparam` according to the type of the node with the given id (`nodeId`) and the parameter name (`paramName`) and the new value given for it (`value`). An example usage of this function is presented in Listing 5.2. There we show how to rename a class named `Main` into `NewName`.

The CT generated for that example is presented in Listing 5.3. Variables `A`, `B` and `C` are used to store the values of the old node that do not change. The predicate `verifyParameterAttrib` ensures

that the new value is valid for that parameter of the node. For example, if the attribute holds a list of ids like the definitions of a class, then the transformation only occurs if the new value is a list of ids. The transformation part of the CT uses the predicate `replace`, which replaces a given fact for another one. Transforming predicates such as `replace` can only be used on the transformation part of a CT.

```
var results = search("classT(Class,_,'Main',_,_)");
applyCtToSetParam(results[0]["Class"], "name", "NewName");
API.export(); // generate the files
```

Listing 5.2: How to rename a class named `Main` into `NewName` using applyCtToSetParam.

```
user:ct(setparam,
  % condition
  verifyParameterAttrib(1, name, 'NewName'),
  % transformation
  replace(
    classT(1, A, _, B, C),
    classT(1, A, 'NewName', B, C))).
```

Listing 5.3: CT resulting from the code in Listing 5.2

After these steps, we already have a way of searching for nodes and a helper to apply transformations. The user can use the `query` function to obtain the ids of the nodes he wants to change and then apply the changes to those nodes using the function `applyCtToSetParam`.

The next step is to turn nodes into JavaScript objects. For this purpose we extend the functionality of the `query` function in `searchNodes(variable, query)`. The parameter `variable` specifies the name of the Prolog variable in which the user is interested. With the call `searchNodes("Class","classT(Class,_,'Main',_,_)")` the user obtains a list of objects that represent class nodes with the name `'Main'`.

With the objects that represent the nodes in the AST the user can now change the parameters of the node in an object oriented way. in Listing 5.4 we show an example of how to change the name of a class. The call to `searchNodes` returns all classes with name `Main`. The `setProperty` method is then used to change the property `"name"` of the class node to `"NewName"`. Finally `exportTransformations()` generates the source code from the altered AST.

```
var results = searchNodes("Class","classT(Class,_,'Main',_,_)");
var classe = results.get(0);
classe.setProperty("name", "NewName");
exportTransformations();
```

Listing 5.4: First first way of transforming nodes.

5.1.1 Base Version

In our approach, a query starts by a search that finds all the nodes of a certain type. For example `_class()` generates a query that finds all the nodes of type class. The attributes of those nodes can

be queried by calling getters (for example `_class().name()` to get the names of the classes). Nodes can be filtered by specifying the desired values of the attributes (ex: `_class().name('Main')` finds all classes named `Main`). It is also possible to set the desired value of an attribute to be equal to the result of a query. We present in Listing 5.5 a query that finds all methods that are defined in class `Main`. The equivalent Prolog query is presented on the same listing.

```
_method().parent(_class().name('Main'))
// same as
// searchNodes("Method","classT(Class,_,'Main',_,_),
//                       methodT(Method,Class,_,_,_,_,_,_)")
```

Listing 5.5: A search using the API and its equivalent using the function `seachNodes`

Our goal was to hide the logic paradigm behind the searches, therefore we use object oriented concepts such as getters and setters to manage the logic variables for the user. When calling the `serachNode` function the user needs to specify which variable holds the values he is interested in. For example in the expression `searchNodes("Class","classT(Class,_,_,_,_)")` the user explicitly indicates the Prolog variable that holds the desired ids. Using our API the search becomes `_class()`. Functions like `_class()` and `_method()` generate a search object each time they are called. The generated object has the information about the parameters of that node and which variable is the one the user wants.

By default the variable we will be interested in is the node's id. Getter methods change the variable to other parameters. In Listing 5.6 calling the getter changed the variable we are interested in from `Id` to `Parent`, while not changing the generated Prolog.

```
_class()
// results in the function call:
// searchNodes("Id","classT(Id, Parent, Name, ParamRef, Defs)"),

_class().parent()
// results in the function call:
// searchNode("Parent","classT(Id, Parent, Name, ParamRef, Defs)").
```

Listing 5.6: Search object before and after calling a getter.

The variable names for the attributes of a node are formed with the names of those attributes. In order to avoid conflicts, each instance of a search object has its own variable names according to the attributes of the node it represents. This is implemented by assigning a different id to each search object, which is added to the names of its variables.

```
var childClass = _class();
// resulting query:
// searchNodes("Id1",
//             "classT(Id1, Parent1, Name1, ParamRef1, Defs1)")

var parentClass = _class();
// resulting query:
// searchNodes("Id2",
```

```
//             "classT(Id2, Parent2, Name2, ParamRef2, Defs2)")

childClass.parent(parentClass);
// resulting query:
// searchNodes("Id1","classT(Id2, Parent2, Name2, ParamRef2, Defs2),
//                    classT(Id1, Id2, Name1, ParamRef1, Defs1)")
```

Listing 5.7: Setter call and its resulting query.

The Listing 5.7 has an example of how the API manages logic variables. By passing a search as a parameter to a setter we actually are setting the variable representing the attribute. Calling the setter `parent(parentClass)` we change the parent attribute from the `Parent1` variable to the `Id2`, which is the main variable of `parentClass`.

Search objects store the values of their parameters and the context of the search. By context of the search we are referring to the searches passed as arguments to setters. For example `parentClass` is part of the context of `childClass.parent(parentClass)`. It is the `parentClass` search that gives meaning to the variable `Id2` that was set to the parent attribute.

Listing 5.8 has several examples of how a setter changes the state of a search object. Below each expression is the state of the resulting search object. The attributes of this object are: the attribute `mainVar` stores the variable of interest; the attribute `type` holds the type of node that the object represents; the attribute `context` holds the searches on which the search object depends on; the attribute `parameters` maps each parameter to its current value.

The Prolog generated for a search object is created from its `context`, `type` and `parameters`. Instead of search objects, the `context` stores their generated Prolog strings. When a search object is passed to the setter the Prolog string is generated and added to the context. On the second example of Listing 5.8 the setter `parent` changes the corresponding parameter from `"Parent3"` to `"Id1"` which is the `mainVar` value of the argument. Because by itself the variable `"Id1"` has no meaning, the context of the argument search is added to the context of the search object. In cases where the the argument is not a search but a name of a class or of a method, the we add single quotes to the value in order to convert it into a string atom.

```
var classA = _class().parent( _class() ).name('A')
// { mainVar: "id",
//   type: "classT",
//   context: "classT(Id2,Parent2,Name2,ParamRef2,Defs2)",
//   parameters: {
//     id: "Id1",
//     parent: "Id2",
//     name: "'A'"
//     ...
//  }
// }

var methodFoo = _method().name('foo').parent( classA )
// { mainVar: "id",
//   type: "methodT",
//   context: "classT(Id2,Parent2,Name2,ParamRef2,Defs2),
```

```
//              classT(Id1,Id2,'A',ParamRef2,Def2)",
//   parameters: {
//     id: "Id3",
//     parent: "Id1",
//     name: "'foo'"
//     ...
//  }
// }
```

Listing 5.8: Examples of how the internal state of a search objects changes with setter calls.

A getter only changes the `mainVar` to the parameter that the user wants. One could implement this by changing the value of `mainVar` directly, however this approach poses a problem. A getter call could change the type of search to the type of the parameter. For example `_method().body()` should change to a query of type block, which means that the resulting object should have the methods for block queries, like `.stmts()` and `.encl()`. However by simply changing the `mainVar` the search would still have methods of a `method` search.

Getters wrap the search object with an object that only provides sufficient information to generate the Prolog query. If we only changed the value of `mainVar` we would allow searches like the one in Listing 5.9 to be created. In that case the call `.parent()` would change the `mainVar` to be `parent`, the call `.name('Main')` would actually overwrite the first call `.name('main')`.

```
_method().name('main').parent().name('Main')
```

Listing 5.9: A query possible to write if the getters only changed the value of `mainVar`.

We include an implementation of the behaviour just described in Appendix A).

5.1.2 Improving the searches

We improve on the previous implementation of the Search API by adding the ability to compose getter calls. Before, a getter call would just change the main variable of a query. The search object continued to be the same. This does not allow for more specific conditions on the result of a getter call to be specified. For example, if we wanted to find all the methods that contain a try catch in a specific class we would want to write a query like the one presented in Listing 5.12.

```
_class().name('Main').def().enclosing(_try())
```

Listing 5.10: Query to find all methods that contain a try catch in a the class Main

The expression `_class().def()` only changes the main variable of the search, not the actual type of the object. To be able to call methods on the result of the getter, we need to return a search object that has the type of the attribute, not the type of `_class()`.

Sometimes the value of an attribute can be a node of one of several types. For example a class can define either fields or methods, so we do not know which type of node the expression `_class().def()` has. In these cases we return a node of type `UnkownTerm` which behaves like the other search objects, without having methods for a specific type. It implements the method `.as(<generation function>)` that works as a cast, by transforming the `UnknownTerm`, to a term of the given type. For

example we can correct the code in Listing 5.12 by using the method `.as(<generation function>)`.

```
_class().name('Main').def().as(_method).enclosing(_try())
```

Listing 5.11: Correct the code in Listing 5.12 by using the method `.as(<generation function>)`.

The method `.as()` also acts as a filter, therefore in Listing 5.12 only the methods of class `Main` are considered, not fields. In this concrete case it is not a problem because that is what we want. However our approach does not allow to search for common attributes of the possible node types. If we need to find in a class the definitions of a given name, regardless if it is a field or a method, we must write two queries (Listing 5.12).

```
var definitionsInMain =
  _class().name('Main').def()
var methodsQuery =
  definitionsInMain.as(_method).name(givenName)
var fieldsQuery =
  definitionsInMain.as(_field).name(givenName)
```

Listing 5.12: Queries to find a definition inside the class main with a `givenName`.

5.2 Transform API

5.2.1 Single Prolog Query Approach

The Transformation API was first developed extending the Search API. Besides the searching methods, search objects had special methods to apply transformations. For instance, in order to change the name of a class one would write an expression such as: `_class().name('Main').setName('NewClass')`. As we can see from the example, the transformation method names were derived from the search methods, with `set` prefixed to them.

Nodes were created by the parent of the new node. This approach is similar to the approach taken by Recoder. For example `class().name('Main').createMethod( method().name('main') )` would create the method named `'main'` in class `'Main'`. Note that when creating a node one would use a search to define the values for the attributes of the new node.

There was no separation between a search object and a transformation object. Search objects held a context for the searches, and an array of transformations to be applied. The transformation would then be constructed along with the searches. At the end a single query would be generated and sent to JTransformer.

When a transformation method is called, such as the change of an attribute value, a Prolog string for that transformation would be stored in the search object. This Prolog string would contain a call to one of those predicates that are only accessible on CTs.

At the moment of generating the final Prolog query to be sent to JTransformer, a CT is generated if the search object holds any transformation to be applied. A search is generated otherwise.

This approach has an advantage: there is only one query computed for each transformation. Besides decreasing the interaction between the JTransformer plugin and the Interpreter plugin, having all the

conditions for the variables allows the Prolog engine to reduce the search for results. Both these aspects result in faster queries.

However this way of describing transformations becomes confusing when one wants to make complex transformations. For example the control flow concepts like 'if then else' could not be handled. To make the transformations easier to describe we opted to use an imperative paradigm.

The major sign that we were on the wrong path, was that JTransformer CTs were cleaner by separating the search part from the transformation part. The target users of our API are programmers that had no or little knowledge of the logic paradigm. It would be difficult to convince them to use an API that was harder to learn than a Prolog API.

5.2.2 ORM Approach

In order to simplify our previous approach we separated the Transformation API from the Search API in order to give it an imperative paradigm. Our new approach is inspired by the Object Relational Model (ORM) were the AST is treated as if it was a database and the Search API is treated like a query language.

Active Record is a pattern for Object Relational Model (ORM), which consists in an object oriented API to access relational databases such as PostgreSQL. One of the main advantages of this pattern is that programmers do not have to mix the object oriented language they are programming in and the SQL used to query the database.

In this pattern queriesreturn Records that represent lines of a table in the database. The columns are represented by attributes of the record and their values are accessed through getters. The values of the columns can be set through the use of setters. The modified values are only saved back to the database when the method `save()` is called on a record.

In our application the AST is the database and the nodes are the records. The program fetches the records of nodes from the AST and manipulates them. Instead of saving each record by hand, the transformation object stores all accessed records and saves them all at once at the end of the transformation.

The transformation object is the only way to obtain a record for a node. Searches created by the Search API are sent as arguments to the `find( search )` method of a transformation. The resulting records are stored in the transformation object along side the records that represent new nodes. This is why the only way to create or access existing nodes is by calling methods on a transformation object.

At the end of the transformation the `apply()` method is called which results in the creation of a Prolog query that represents all the manipulations and creations of records. The accessed records are checked for changes. If any of its attributes was changed, the corresponding Prolog transformation is generated. Similarly for each record that represents a creation, there is a Prolog query generated. All modifications are concatenated into the same Prolog query which is sent to the JTransformer plugin.

Contrary to the original approach, the ORM approach forces the AST to be accessed before the final transformation is generated. For example, in order to access the class name, it needs to be fetched from the factbase at the time that the transformation is being specified. This means that several individual queries are made to the factbase before they would be all joined together in the final transformation query.

The advantage of this approach is that it is easier to understand for the user. Moreover, with this approach the user can access the values of the nodes during the creation of the transformation, which allows him to use the JavaScript control structures. This allows the user to find additional nodes that he would not be able to find with the Search API alone. We make use of this feature on the evaluation during the Atomic Catches transformation (Section 6.3.2)

5.3 Problem of order of terms

In the Search API, the act of generating the final query string for a search object is a matter of generating the string for the current search object and concatenating it with its context. However, the order in which this concatenation happens affects the search space for the Prolog engine. The terms that have less possible solutions should appear before the ones that might have more.

Note that in general the number of possible solution of a query does not necessarily have a direct relation with the size of the search space. For example, if the query includes a procedure, the complexity of the query depends of the complexity of that procedure. However, in our case most of the AST information is encoded as facts, which means that we can more easily predict the size of the search space of a query.

In the next sections we are going to present how search objects and helper predicates are concatenated and the reasoning behind our choice.

5.3.1 Suffixing or prefixing the term to the context

In the case of concatenating the context of a search object with its term string, we cannot know in every instance which is better: if to add the term before or after the the context. Listing 5.13 has the two possible Prolog queries to be generated from `_method().parent(_class().name('Main'))`. On first option which is the result of prefixing the term of the search object to its context, the fact for the method will have as solution all the methods in the program. This means that the fact for the class with name `'Main'` will be searched for every method. On the second option the class term will only have as solutions the facts of classes named `'Main'`, giving a concrete value to the `C` variable reducing the search space to solve the method term.

```
_method().parent(_class().name('Main'))
// could produce:
//  "methodT(M, C, _, _, _, _, _, _), classT(C, _, 'Main', _, _, _)"
//  "classT(C, _, 'Main', _, _, _), methodT(M, C, _, _, _, _, _, _)"
```

Listing 5.13: Tow ways of concatenating search objects

There is no way of knowing for sure that the context will reduce the number of unifications for the term (a user could pass as argument a search that does not have concrete values for an attribute). Nevertheless, the objective of calling a setter with a search object as argument is to further narrow the number of facts that would be unified by the original search object. Therefore it may be a better approach.

5.3.2 Suffixing or prefixing the helper to the context

The helpers are added before or after the term, but always after the context. They are added before the term if they will reduce the number of solutions of the term, and after if they either need the term's parameters to be bound or if they compute faster in that case.

For example the helper `search.in()` will use the predicate `optEnclosing(Node,Parent)`. If the number of edges between `Node` and `Parent` is smaller than the number of facts for `search`, putting the predicate before will reduce the search space of the query. This was the approach taken because we wanted to optimize the cases where the user wants to find a node like an `if` or a `for` inside a specific method. We took this approach in order to speed up the cases were we were using the

helper `search.in()`. On the other hand the helper `search1.isNot(search2)` is implemented using the predicate from Prolog `not/1`. This helper ensures that the main variable of `search1` is not true in `search2`. In this case the `not/1` predicate requires that the variables of both searches are set, therefore the helper is added after the term.

Examples of these helpers and their corresponding Prolog queries can be seen in Listing 5.14. The first example is a query that uses the helper `search.in()` and the second is a query that uses the helper `search1.isNot(search2)`.

```
operation().op('==').in( ifSearch )
// "ifT(If, _, _, _, _, _),
//  optEnclosing(Op,If),
//  operationT(Op,If,_,_,'==',_)"

_class().isNot( _class().name('Main') )
// "classT(C,_,_,_,_,_,_,_,_), not(classT(A,_,'Main',_,_,_,_,_,_))"
```

Listing 5.14: Helper examples and the corresponding Prolog query

5.4 Console interface

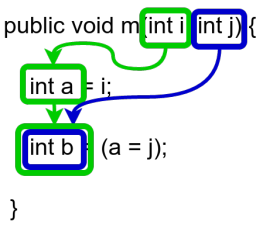
The console interface was integrated into Eclipse. It uses the API for creating a console view inside the development environment. The interactive loop work as follows:

- wait for user input;
- ask the Nashorn instance to evaluate the user input;
- save the result into the variable `_`;
- call the function `printResult(_)` that implements the rendering of the result.

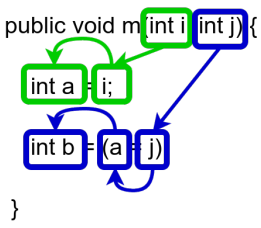
The function `printResult(_)`, which is defined in the JavaScript sources, generates the string to be printed on the console for a given result. Its default behavior is to call `.toString()` on the argument, which is a method present in JavaScript objects. This function makes a special case for search objects. Because we want to make it easier for the user to know the results of a search, the function `printResult(_)` will run the Prolog query of a search object in order to show the class, method and line where the source of the node is located. The information about the source location of a node is computed by a predicate which reads the source corresponding to a node from its source file. An example of this special case can be seen in Section 4.8.

To reduce the time that the user waits for a result we used the asynchronous API of JTransformer to be able to present results as soon as they are computed. Without this optimization the user would have to wait that the Prolog engine had computed all the results of the query before printing results. Because the JTransfomer API for asynchronous queries is complex, we hide that complexity inside a method of the RefactoringApi class (as mentioned in the previous chapter).

Another advantage of asynchronous queries is that we can chose to only pretty print the first few results, before the engine finds all other matches. The limit is there to prevent a console cluttered with too many results and to allow the user to refine his query without having to wait for all the results. If the user wishes to see all the results he can turn off the limit for the shown results. The limit of results works only on the pretty printing of searches. On transformations the search returns a record for each result.



(a) Ignoring inner assignments.



(b) Taking inner assignments into acount.

Figure 5.2: Data flow diagram for the variables `i` and `j` when Figure 5.1a the data flow does not takes into acount assignments inside expressions and when it does Figure 5.1b

5.5 Control Flow and Data Flow

We added also a way of letting the user know where the value of a variable propagates to. This is essentially following the path of a data flow graph, starting at the variables declaration. The implementation of the data flow calculation is done using a definition of control flow that takes into consideration the order of evaluation of the expressions. Calculating the data flow is then a matter of following the control flow path starting on the variable declaration and picking the nodes that reference the data in that variable, registering the other variables whose value depend on that variable.

Control flow is used to know the order in which the AST nodes are executed. The data flow registers the variables that depend on the value of the chosen variable. This dependency is propagated through assignments and method calls.

By taking the order of evaluation of expressions into account in the control flow, it is possible to follow data flow inside expressions. For example in Listing 5.15, the assignment inside the initialization of the variable `b` will make `b` data independent of `i` (Figure 5.1b). Without processing the AST of the expressions we would not detect the change in dependency of the variable `a` (Figure 5.1a).

```
public void m(int i, int j) {
  int a = i;
  int b = (a = j);
}
```

Listing 5.15: Example where the control flow inside expression helps to calculate a better data flow (see Figure 5.1a).

5.5.1 Control Flow

The main part of calculating the control flow is to determine the control flow successor for a given node. We start by describing how to calculate control flow only for statements and then we will introduce expression nodes. This is our predicate `cfSucc(Node, Successor)`. For the time being we only consider these two arguments. Further down the line we will add more arguments to this predicate to include state from previous steps in the flow path.

The simplest case is when the node is a statement in a block, where its successor is either the next node in the block or the statement after the block. If it is the last statement then it goes recursively to the parent node until it finds an ancestor block or until it reaches the method where the original node is in.

```
blockCfSucc(Node, Successor, Method) :-
  \+ Node = Method,
  ast_edge_value(X, parent, Parent),
  (
    blockT(Parent, _, Stmts),
    nextOnList(Node, Stmts, Succ), !
  );(
    blockCfSucc(Parent, Successor)
  ).
```

Listing 5.16: Flow successor of a statement in a block. The predicate `nextOnList` finds the element on `List` that is after `Element`.

The nodes that are exceptions to this rule are flow forks like `if` nodes, `for` nodes etc. These nodes have more than one successor. The node that is executed if the condition is true and the node that is executed if the condition is false.

```
ifCfSucc(Node, Successor) :-
  ifT(Node, _, _, _, Succ, _);
  (
    ifT(Node, _, _, _, _, Succ),
    % check if the node as an else block
    \+ Succ = null
  )
```

Listing 5.17: Flow successor for `if` nodes

```
forCfSucc(Node, Successor) :-
  forT(Node, _, _, _, _, _, Succ);
  ( % loop the last statement of the block back to the for node
    forT(Successor, _, _, _, _, _, Block),
    blockT(Block, _, _, Stmts),
    last(Node,Stmts)
  ).
```

Listing 5.18: Flow successor for `for` nodes

Note that we did not add specifically the calculation of the successor of those nodes in the case when the condition is false, nor for the successor of the last statement of their blocks. Those cases are given by the predicate `blockCfSucc`. The last statement of the `for` block has as successor the `for` node itself to represent the control loop.

We can combine the successor predicates together to form a simple control flow predicate.

```
cfSucc(Node, Successor) :-
  ifCfSucc(Node, Successor);
  forCfSucc(Node, Successor);
  (
```

```
    ast_edge_value(Node, encl, Method),
    blockCfSucc(Node, Successor, Method)
  ).
```

Listing 5.19: Control flow predicate.

To introduce expressions into the control flow we define that the expressions of a statement are evaluated before the statement. For example the initializer expression precedes the variable declaration. This means that one can see the order in which the expressions are executed by following the control flow. This will be used when defining the data flow.

We can no longer rely on the predicate `blockCfSucc` to be the general case to calculate a control flow successor. The successor of a statement now has to be the inner-most expression on the next statement. For example, in Listing 5.15 the successor of the declaration of variable `a` should be the reading of variable `a` on the initialization expression of variable `b`. Instead we add an additional computation after the control flow successor is found. If the new successor is a control flow fork the successor is changed to be the flow forks condition. We call this computation bypass.

The bypass computation is also responsible for redirecting the flow from a statement to its inner most inner expression. Because we only want to bypass the statement once we need to know if the flow is going inside the statement or outside the statement. For that we need to know both the computed successor and the original node.

```
bypass(Initial, Final, Result) :-
  bypassAux(Initial, Final, R) ->
      bypass(Final, R, Result)
    ;(
      (ast_node_type_for_id(Initial, TypeI),
       \+ expressionType(TypeI),
       ast_node_type_for_id(Final, TypeF),
       expressionType(TypeF)) ->
        firstInnerExpr(F, Bypass)
        ;
        Bypass = F
    ).

bypassAux(Initial, Final, Result) :-
  ifT(F, _, _, Bypass, _, _);
  forT(F, _, _, [Bypass|_], _, _, _);
  forT(F, _, _, [], Bypass, _, _);
  forT(F, _, _, [], null, _, Bypass);
  blockT(F, _, _, [Bypass|_]);.
```

Listing 5.20: Bypass predicate.

The predicate `bypass` first searches recursively if the node is possible to bypass. When there is no bypass for the node then it checks if the flow is going from the statement to the expression. If it is the case then the flow is redirected to the first inner expression to be evaluated.

Now that the control flow may receive expression nodes we need a way of calculating the successor of an expression. The successor of an expression is the next expression to be evaluated.

In Java, expressions are evaluated from left to right, bottom up. Which means that subexpressions are evaluated first and from left to right, only then the parent node is evaluated. The predicate `subExprs` (Listing 5.21) calculates a list with the children expressions in the order they are evaluated.

```
subExprs(Expr, SubExprs) :-
    operationT(Expr, _, _, SubExprs, _, _);
  (
    callT(Expr, _, _, Receiver, ExprArgs, _, _, _),
    (\+ Receiver = null ->
      append([Receiver],ExprArgs, SubExprs)
      ;
      SubExprs = ExprArgs)
  );
    newT(Expr, _, _, _, SubExprs, _, _, _, _);
  (
    (assignT(Expr, _, _, Lhs, Rhs);
     assignopT(Expr, _,_, Lhs, _, Rhs)),
     SubExprs = [Lhs, Rhs]
  );(
    conditionalT(Expr, _, _, Cond, Thenexpr, Elseexpr),
    (SubExprs = [Cond, Thenexpr]; SubExprs = [Cond, Elseexpr])
  );(
    (ast_edge_value(Expr, expr, InnerExpr);
     fieldAccessT(Expr, _, _, InnerExpr, _, _)),
    SubExprs = [InnerExpr]
  ).
```

Listing 5.21: Predicate that finds the children expressions ordered by order of evaluation.

For a given expression we need to find its sub expression that is executed first. That is achieved by recursively calling `subExprs` on the head of the sub expression list. The algorithm is implemented on the predicate `firstInnerExpr` (Listing 5.22). The predicate `evaluationLeaf` checks if a node has no sub expressions. By considering array indexing to be an expression leaf we exclude their sub expressions from the control flow.

```
firstInnerExpr(Expr, Expr) :- evaluationLeaf(Expr).
firstInnerExpr(Expr, FirstInner) :-
  subExprs(Expr, [SubExpr|_]),
  firstInnerExpr(SubExpr, FirstInner).

evaluationLeaf(Expression) :-
  ast_node_type_for_id(Expression, Type),
  (
    Type = identT;
    Type = indexedT;  % not supporting arrays
    Type = newArrayT; % not supporting arrays
    Type = literalT;
```

```
    Type = selectT;
    Type = staticTypeRefT;
    Type = typeCastT
  ).
```

Listing 5.22: Predicate to find the first expression to be evaluated for an expression node.

Now we have the necessary predicates to define the next expression to be evaluated. If the expression has siblings then the next expression is the inner most expression of that sibling. Otherwise the next expression is the parent. This logic is implemented in the `nextExpr` predicate in Listing 5.23.

```
nextExpr(Expr, Next) :-
  ast_edge_value(Expr, parent, Parent),
  ast_node_type_for_id(Parent, ParentType),
  expressionType(ParentType),
  subExprs(Parent, InnerExprs),
  (nextOnList(Expr, InnerExprs, Sibling) ->
    firstInnerExpr(Sibling, Next)
    ;
    Next = Parent
  ).
```

Listing 5.23: Control flow successor for an expression.

The control flow successor for an expression is implemented by the `exprCfSucc` predicate (Listing 5.24). We only add a few more checks before calling the `nextExpr` predicate. The first one is to check if the given node is an expression. The second is to check if the expression's parent is a statement. If that is the case, the control flow successor will be computed by the appropriate statement successors. For example, the `ifCfSucc` determines the successor for its condition.

```
exprCfSucc(X, Succ) :-
  % verify that x is expression
  ast_node_type_for_id(X, XType),
  expressionType(XType),

  % Only calc successor if X is not the root node of
  % an expression.
  ast_edge_value(X, parent, Parent),
  ast_node_type_for_id(Parent, ParentType),
  expressionType(ParentType),

  nextExpr(X, Succ).
```

Listing 5.24: Control flow successor for an expression.

Now that expressions are part of the control flow we can follow method calls (`callCfSucc`). The successor of a method call is the first statement of the target methods body. We can also better determine the successor of a `return` node or the last node on a method body. In both cases there are two

alternatives: either the call site is known and the successor is the corresponding node, or the successors are all places where that method is called.

To keep track of the call site we add the argument `CallStack` to `cfSucc` to store a stack of method calls. The `callCfSucc` predicate besides finding the first statement of the method being called also updates the call stack with the current node.

```
callCfSucc(Node, Succ, CallStack, [Node|CallStack], FollowCall) :-
  FollowCall,
  callT(Node, _, _, _, _, Method, _, _),
  methodT(Method, _, _, _, _, _, _, Succ).
```

Listing 5.25: Predicate to find the control flow successor of method calls.

Note that `callCfSucc` does not directly point to the first statement of the method body, but rather the block of statements. We let the `bypass` predicate, that will be called later, to deal with that. The predicate also has a parameter (`FollowCall`) to determine if the call is to be followed into the method body. This will be used later to make the successor of the call node to be the expression successor instead.

There are two possible cases when finding the control flow successor of a return. If we know the call node that originated the call we can point the flow to it. On the other hand we have to chose to either stop the control flow there or fork the control flow into all method calls of the returning method. We opted to fork the control flow, because this provides a more complete information about the control flow.

When returning from a method call site the control flow successor will be the successor of the method call, but this time the call node will be treated as any other expression. We will need to change the original node from the return statement to the call in order for the bypass predicate to work as expected. For this we add a new parameter `MiddleNode` that will be set to the call node.

Returning to the call site is handled by the `returnToCallSite` predicate. This predicate is used in both the `returnCfSucc` and `blockCfSucc`.

```
returnToCallSite(Method, Succ, [], [], MethodCall) :-
  callT(MethodCall, _, _, _, _, Method, _, _),
  cfSucc(MethodCall, Succ, CallStack, CallStack, false, _).

returnToCallSite(Method, Succ, [Call|CallStack], CallStack, Call) :-
  callT(Call, _, _, _, _, Method, _, _),
  cfSucc(Call, Succ, CallStack, CallStack, false, _).
```

Listing 5.26: Predicate that determines the successor given a stack of calls.

```
blockCfSucc(Node, Successor, Method, CallStack, ResultSack,
            MiddleNode) :-
  ast_edge_value(X, parent, Parent),
  (
    blockT(Parent, _, Stmts),
    (
      (nextOnList(Node, Stmts, Succ),
       ResultCallStack = CallStack,
```

```
        MiddleNode = Node, !);
      blockCfSucc(Parent, Method, , CallStack, ResultCallStack,
                  MiddleNode)
  );(
    Node = Method,
    returnToCallSite(Parent, Succ, CallStack, ResultCallStack,
                     MiddleNode)
  ).
```

Listing 5.27: Block successor that takes into account previous calls.

```
returnCfSucc(X, Succ, CallStack, ResultCallStack, MiddleNode) :-
  returnT(X, _, Method, _),
  returnToCallSite(Method, Succ, CallStack, ResultCallStack,
                   MiddleNode).
```

Listing 5.28: Return successor that takes into account previous calls.

Before conditions became control flow forks, nodes like `if` and `for` would have two successors: the successor when the condition is true and the successor when the condition is false. In both cases the case where the condition is false is given by the `blockCfSucc` (except when the `if` node has an else block), which generally calculates the successor of a statement in a block. The predicate `blockCfSucc` was also used in conjunction with the previous case to calculate the successor of the last statement on `then` and `else` blocks and for bodies.

When conditions become the control flow forks, `if` and `for` nodes only have one successor: the condition in the case of the `if` and the initializer in the case of the `for`. This means that `blockCfSucc` no longer can calculate the successor for flow fork statements. To solve this we chose to remove the control flow nodes from the flow, and instead add bypass rules to the `bypass` predicate. Now the flow is bypassed directly to the control flow condition.

However we still want to use the default case of `blockCfSucc` which is to generally go up the tree until it finds a successor or the method. In the case of loops we do not want the default behaver so we added a new condition to the `blockCfSucc` to avoid that.

Finally we adapt the `cfSucc` predicate to make use of a provided stack and to return a new one if either a call was followed or it returned from a call. Besides the stack parameters we also add a parameter to specify whether we want to follow the method call (`FollowCall`) and an out parameter to know if the successor returned from a method call (`Returning`). These last two parameters will be used when calculating the data flow. The `FollowCall` parameter will be set to false when none of the method call arguments are data dependents, i.e. they are not in the data flow. By not following the method call we avoid unnecessary control flow calculations. The `Returning` parameter is a more efficient way of knowing if the flow is returning from a function than comparing the original and resulting call stacks. The data flow predicate uses this information to update the context variables. The final code for the predicate `cfSucc` is in Listing 5.29.

```
cfSucc(Node, Successor, CallerStack, ResultStack, FollowCall,
       Returning) :-
  (
    (
```

```
    (
      (callCfSucc(Node, Succ, CallerStack, ResultStack,
                  FollowCall), !);
      (
        ( % find next node on the flow
          (callCfSucc(Node, Succ, CallerStack, ResultStack,
                      FollowCall), !);

          exprCfSucc(Node, Succ, FollowCall);
          execT(Succ, _, _, Node);
          localT(Succ, _, _, _, _, Node);
          returnT(Succ, _, _, Node) ),

        ResultStack = CallerStack
      )
    ),
      FinalNode = Node,
      Returning = false
  );(
    ifCfSucc(Node, Succ, CallerStack, ResultStack, FinalNode,
             Returning);
    forCfSucc(Node, Succ, CallerStack, ResultStack, FinalNode,
              Returning)

  );(
    (
      % If the return is the last stmt then it could have a block
      % successor. Therfore if the node has return successor then
      % it should not attempt the block successor branch.
      (returnCfSucc(Node, Succ, CallerStack, ResultStack
                    MiddleNode),
       Returning = true, !);


      blockCfSucc(Node, Succ, CallerStack, ResultStack,
                  MiddleNode, Returning)
    ),
      FinalNode = MiddleNode
  )
),
bypassNode(FinalNode, Succ, BypassedSucc),
bypassExpr(FinalNode, BypassedSucc, Successor).
```

Listing 5.29: Final control flow predicate

The predicate `cfSucc` calculates the control flow successor for any given node and can receive state about the call stacks. To follow the control flow it is necessary to keep the state of the stack between

calls to `cfSucc` and store the visited nodes in order to avoid to be stucked in cicles. The predicate `cfSuccPath` makes use of `cfSucc` to return all the nodes in the control flow path of a given node.

```
cfSuccPath(Node, Successor) :-
  cfSuccPath(Node, Successor, [], []).

% control flow that avoids visiting the same node infint times
cfSuccPath(Node, Successor, Visited, CallStack) :-
  cfSuccPathStep(Node, NewSuccessor, Visited, CallStack, true, _,
                 NewVisited, NewCallStack),
  (
    Successor = NewSuccessor
  ;
     (!, cfSuccPath(NewSuccessor, Successor, NewVisited,
                    NewCallStack))
  ).


cfSuccPathStep(Node, Successor, Visited, CallStack, FollowCall,
               Returning, [Node|UpdatedVisited], NewCallStack) :-
  \+ member(Node, Visited),
  cfSucc(Node, Successor, CallStack, ResultStack, FollowCall,
         Returning),
  (
    % update visited if the callstack changed
    (nonvar(ResultStack), !,
     length(CallStack, LengthCallStack),
     length(ResultStack, LengthResultStack),

     % following a call to a method body
     ((ResultStack = [HeadCall |_],
      LengthCallStack < LengthResultStack) ->
      UpdatedVisited_a = [HeadCall|Visited];
      UpdatedVisited_a = Visited),

     % if returned from call visted nodes during that call
     ((CallStack = [HeadCall|_],
      LengthCallStack > LengthResultStack) ->
      dropUntil(HeadCall, UpdatedVisited_a, UpdatedVisited);
      UpdatedVisited = UpdatedVisited_a)
    );
      UpdatedVisited = Visited
  ),(
    var(ResultStack) ->
      NewCallStack = CallStack
      ;
      NewCallStack = ResultStack
```

```
).
```

Listing 5.30: Predicate used to get all nodes that belong to the control flow path of Node.

The predicate `cfSuccPath` behavior is mainly encapsulated in the `cfSuccPathStep` predicate. This last predicate does not simply store a list of visited nodes. It takes function calls into account so that it does not stop the flow the second time a method is called. That would happen if we stored just the ids of the visited nodes. Consider the example where the flow starts at the beginning of a method: if that method would call other method twice, then the same nodes would be visited twice before reaching the end of the original method's body.

The predicate `cfSuccPathStep` stores the visited nodes in a stack. At the top of the stack is a list of the nodes visited in the current method call. If the flow goes into a method call an empty list is pushed into the stack. If the flow is returning from a method then the stack is popped and the checks of visited nodes are now made with the new head of the stack.

Note that the consecutive returns of `cfSuccPath` do not determine that both nodes are control flow successors. Only that they are in the control flow path that starts at the starting node. This could be a problem but not in the application we are using it for, which is presenting to the user the nodes in the control flow of another node. Also by using backtracking we can take advantage of the asynchronous API of JTransformer for Prolog queries. That API does finds all the solutions for a Prolog query and returns as soon it finds a result. Besides enabling us to present results to the user without having to wait for all the computations to finish (contrary to a solution that would return a list of nodes). It has the added advantage that, when using it on scripting transformations, we can run JavaScript code at the same time we run a Prolog query.

5.5.1.1 Simplifications

In this implementation of the control flow we only take into account the `for` and `if` nodes. Also `if`, `for` and `block` nodes do not make part of the control flow, because they are bypassed. This could be a problem if an algorithm was to be implemented using `cfSucc` that needed those nodes to be part of the control flow. In our case the main use of this predicate is to show to the user the lines of code that are executed. By removing the nodes that contain several lines of code inside them we simplify that process.

5.5.2 Data Flow

The data flow is a graph that represents the dependency of data. In our implementation it represents the dependency on the value initialized to a variable. Therefore the root of our data flow is a variable declaration. The most obvious candidates to belong to the data flow graph are the expressions that reference the said variable and the statements they are embedded in. The dependency on a variable can be propagated to another through an assignment and to a parameter of a method call. A variable is excluded from the data flow when it is assigned a value that is not on the data flow.

In our algorithm we keep a list of variables that are data dependent. If a new variable is assigned an expression that is dependent on the value of any of the variables on the list, then the new variable is added to the list. If any of the variables on the same list is assigned a value that is not data dependent, then that variable is removed.

When following a method call to that methods body, the arguments that are data dependent determine which parameters will be considered data dependent. Note that we are excluding anonymous instance creation and object fields from our data flow.

The predicate `exprIsDataDependent` (Listing 5.31) determines if a given expression node is on the data flow. There are three possible instances for an expression to belong to the data flow. Either it is a reference to a variable on the list of data dependent values (`VarsWithValue`), it is an ancestor of such variable, or it belong to the list of dependent nodes.

```
exprIsDataDependent(VarsWithValue, Expr, ) :-
  (
    identT(Expr, _, _, Var),
    member(Var, VarsWithValue)
  );(
    ast_ancestor(Child, Expr),
    identT(Child, _, _, Var),
    member(Var, VarsWithValue)
  ),!.
```

Listing 5.31: Predicate that determines if an expression is data dependent.

Knowing which expressions are data dependent allows us to know which nodes are data dependent. The predicate `updateVArsWithValue` (Listing 5.32) checks to see if `Node` updates the current list of data dependent variables. In that case the list is updated and the predicate is successfull, and fails otherwise. The types of nodes that can change the list of data dependent variables are assignments and declarations of variables.

```
updateVarsWithValue(Node,VarsWithValue(Node, VarsWithValue,
                    UpdatedVars) :-
  assignT(Node, _, _, Ident, Expr),
  identT(Ident, _, _, NewVar),
  (member(NewVar, VarsWithValue) ->
    (

      \+ exprIsDataDependent(VarsWithValue, Expr),
      delete(VarsWithValue, NewVar, UpdatedVars)
    );(
      exprIsDataDependent(VarsWithValue, Expr),
      UpdatedVars = [NewVar|VarsWithValue]
    )).


updateVarsWithValue(Node, VarsWithValue, VarsWithValue) :-
  identT(Node, _, _, Var),
  member(Var, VarsWithValue).

updateVarsWithValue(Node, VarsWithValue, [Node|VarsWithValue]) :-
  localT(Node, _, _, _, _, Init),
  exprIsDataDependent(VarsWithValue, Init).
```

Listing 5.32: Predicate that is responsible to update the list of data dependent variables.

We also encapsulated the main logic of data flow into a step function that is responsible to update the stack of dependency lists. That predicate's code is in Listing 5.33.

```
dataFlowStep(Node, IsOnDataFlow, [DependentVars|RestContexts],
             Returning, FollowCall, UpdatedContexts) :-

(callT(Node, _, _, _, Args, Method, _, _) ->
  (  % Find arguments that are data dependent on the variables
     % on the context.
    findIndicesWhere(exprIsDataDependent(DependentVars), Args,
                     Indices),
    (Indices = [] ->
      (  % There are no arguments dependent on the context so do
         % not follow call.
        FollowCall = false,
        IsOnDataFlow = false,
        UpdatedContextsAux = [DependentVars|RestContexts]
      );(
        % add the parameters that are data dependent to the context
        FollowCall = true,
        IsOnDataFlow = true,
        methodT(Method, _, _, Params, _, _, _, _),
        selectIndices(Indices, Params, DependentParams),
        UpdatedContextsAux =
           [DependentParams, DependentVars | RestContexts]
      ))
  );(
    FollowCall = true,
    % update the dependent variables if needed
    (updateVarsWithValue(Node, DependentVars, UpdatedVars) ->
      (
        IsOnDataFlow = true,
        UpdatedContextsAux = [UpdatedVars|RestContexts]
      );(
        UpdatedContextsAux = [DependentVars|RestContexts],
        (isDataDependent(Node, DependentVars) ->
          (IsOnDataFlow = true,
           UpdatedContextsAux = [DependentVars|RestContexts]);
          (IsOnDataFlow = false,
           UpdatedContextsAux = [DependentVars|RestContexts]))
      ))
  )
),

(Returning ->
  UpdatedContexts = RestContexts
  ;
```

```
  UpdatedContexts = UpdatedContextsAux).
```

Listing 5.33: Step function used to update the state of data flow.

Similarly to the `cfSuccPathStep`, the predicate `dataFlowStep` also updates a stack when the flow goes into a method call and when returning from it. When going into a call it needs to add to the new context list the parameters that are data dependent. This is determined by knowing which argument expressions are data dependent. If none is found to be data dependent then the call will not be followed by the control flow predicate, by setting the variable `FollowCall` to false. If the node is not a call then we check if the node is data dependent and change the data dependent variable list if necessary. Finally we check if the flow is returning from a method call. If that is the case then we discard the current list of data dependent variables.

The `dataFlowStep` is then coupled with `cfSuccPathStep` to define the `dataFlow` predicate whose code is in Listing 5.34.

```
dataFlow(VarDecl, OnDataFlow) :-
  dataFlow(VarDecl, OnDataFlow, [], [], true, [[VarDecl]]).

dataFlow(PreviewsNode, OnDataFlow, Visited, CallStack, FollowCall,
         Contexts) :-
  cfSuccPathStep(PreviewsNode, Node, Visited, CallStack, FollowCall,
                 Returning, NewVisited, NewCallStack),
  (isInterestingNode(Node) -> % Ignore expressions that can not
                              % change the data flow.
      dataFlowStep(Node, IsOnDataFlow, Contexts, Returning,
                   NewFollowCall, NewContexts)
      ;
      (
        NewFollowCall = FollowCall,
        NewContexts = Contexts,
        IsOnDataFlow = false
      )
  ),(
    (IsOnDataFlow,
     OnDataFlow = Node
    );(
     dataFlow(Node, OnDataFlow, NewVisited, NewCallStack,
              NewFollowCall, NewContexts)
    )
  ).

isInterestingNode(Node) :-
  ast_node_type_for_id(Node, Type),
  expressionType(Type) ->
    (
      % the interesting nodes for data flow
      Type = callT;
```

```
    Type = assignT;
    Type = identT
);(
    true
).
```

Listing 5.34: Predicate that returns all nodes on the data flow graph for a given node.

As with the `cfSuccPath` predicate the `dataFlow` predicate has a case with only two arguments which is the starting point of the recursion. The first step of calculating the next data dependent node is to find its control flow successor. That is achieved by calling the `cfSuccPathStep` predicate. That call also provides information to weather the flow went into or returned from a method call. That information is then passed to `dataFlowStep` to update the data dependent lists and to determine if the node is data dependent. The call to `dataFlowStep` is only made if the `Node` is a statement or an expression that might change the data flow. This filtering is used to reduce the number of calls to `dataFlowStep` while the `cfSuccPathStep` walks through a expression tree.

After those computations we already know if `Node` is on the data flow. If that is the case we return the value by binding it to `OnDataFlow`. Backtracking will result in the calculations of the next data flow successor.

5.5.2.1 Simplifications

Note that this implementation of the data flow does not take into account the object fields and anonymous class declarations. These require more complex state keeping while walking on the control flow graph and were left for future work. Still our solution takes into account the order of evaluation of expressions, which makes it a complete solution on the space of cases we are considering.

Chapter 6

Evaluation

Our goal was to create a tool that allows the users to write scripts that transform source code. Whether it is because they want to refactor it, or to adapt the source code to the new version of a library. In this chapter we evaluate our tool in three steps. First we compare our tool with other scripting tools by implementing the conversion from == to `.equals()` from the Chapter 4. Then we implement a more complex example on our tool to convert an `enum` into the `State Pattern`. Finally we present two a real world transformations on the on the Fénix open source project.

6.1 String equals

In this section we are going to present how the string compare refactoring, can be implemented using the tools Recoder, JTransformer and Eclipse LTK. The other tools mentioned on the related work either were not accessible or they did not work. In particular we could not generate code for the transformation we created in Rascal. We do not compare our tool with Stratego/XT because it utilizes a radically different paradigm.

6.1.1 Recoder

Recoder is a transformation scripting tool that uses Java as scripting language. Both the Search and Transformation APIs are imperative. In order to enable the creation of more efficient refactorings, the programmer has to specify which files are needed to be parsed for the transformation.

In order to create a Recoder transformation we subclass the `Transformation` class and write its code on the method `execute`. The `Transformation` class provides some methods to access the registries and factories. in Listing 6.1 there is the initialization of our transformation. Line 7 fetches the `String` class that we will need to reference the `equals` method. In line 12 we get an iterator to go through all the nodes in a program's AST. The search is done by iterating through all the nodes in the AST and applying the transformation when we find an operator == node which in Recoder's AST is represented by the class `Equals`. The iteration cycle start in line 15. Lines 18-23 test if the node represents a comparison between strings.

```
1 CrossReferenceServiceConfiguration
2   crossReferenceServiceConfiguration = getServiceConfiguration();
3 ProgramFactory localProgramFactory = getProgramFactory();
4 ClassType stringClass =
5   crossReferenceServiceConfiguration
```

```
6       .getNameInfo()
7       .getJavaLangString();
8   SourceInfo localSourceInfo =
9     crossReferenceServiceConfiguration.getSourceInfo();
10  List localList =
11    getSourceFileRepository().getKnownCompilationUnits();
12  ForestWalker localForestWalker = new ForestWalker(localList);
13  Method equalsMethod = getMethod(stringClass, "equals");
14  TreePrinter printer = new TreePrinter();
15  while (localForestWalker.next()) {
16    ProgramElement localProgramElement =
17      localForestWalker.getProgramElement();
18    if ((localProgramElement instanceof Equals)) {
19      Equals localEquals = (Equals)localProgramElement;
20      Expression arg0 = localEquals.getExpressionAt(0);
21      Expression arg1 = localEquals.getExpressionAt(1);
22      if ((localSourceInfo.getType(arg0) == stringClass)
23          && (localSourceInfo.getType(arg1) == stringClass))
24      {
25      // .. do transformation
26      }
27    }
28  }
```

Listing 6.1: ”Recoder searching comparisons.”

There is another search that needs to be defined. In Listing 6.1, in line 13 calls a user defined search to find a method with a given name on a given class. The code for that search is presented in Listing 6.2. On an imperative paradigm the searches are made reusable by defining them in individual methods.

```
1  private Method getMethod(ClassType classType, String methodName) {
2    for(Method md : classType.getMethods()) {
3      if(md.getName().equals(methodName))
4        return md;
5    }
6    return null;
7  }
```

Listing 6.2: ”Find method in class.”

The code for the transformation itself is presented in Listing 6.3.

```
1  ASTList<Expression> args = new ASTArrayList<Expression>();
2  args.add((Expression)arg1);
3
4  Identifier identifier =
5    ((MethodDeclaration)equalsMethod).getIdentifier();
6  MethodReference equals =
```

```
 7    localProgramFactory.createMethodReference(identifier, args);
 8  ParenthesizedExpression e =
 9    localProgramFactory.createParenthesizedExpression(arg0);
10  equals.setExpressionContainer(e);
11
12  printer.printTree(equals);
13  ((Equals) localProgramElement)
14    .getASTParent()
15    .replaceChild(localProgramElement, equals);
16  ChangeHistory ch = getChangeHistory();
17  ch.replaced(localProgramElement, equals);
```

Listing 6.3: Code that replaces an `Equals` node with a `MethodReference` to the equals method.

6.1.2 JTransformer

JTransformer uses Prolog as scripting language. Each transformation is defined through the predicate `ct(Name, Condition, Transformation)`:

- **`Name`** - defines the name and possible arguments of the transformation;
- **`Condition`** - is the precondition that has to be met to apply the transformation and acts as a search;
- **`Transformation`** - the transformation will be applied to all the results of the `Condition`.

Listing 6.4 shows the JTransformer `ct` that replaces the operator == between strings with a call to the equals method. Being a more declarative language, Prolog allows to make the iteration through the AST implicit, removing the need to write loops and nested ifs to select the nodes of interest.

The declarative transformations also come with its drawbacks. Because the transformation is applied for each binding that makes the condition true, it is hard to foresee all the cases where the transformation is applied. We solve this problem in our API by allowing the user to imperatively specify which nodes he wants to transform.

```
 1  replaceIdInNode(ParentId, ParentTerm, TranslatedTerm, OldId,
 2                  NewId):-
 3    ast_node_type_for_id(ParentId, ParentType),
 4    ast_term(ParentType, 'Java', ParentTerm),
 5    ParentTerm =.. [ParentType, ParentId | _],
 6    ParentTerm,
 7    replaceIdInTerm(ParentTerm, TranslatedTerm, OldId, NewId).
 8
 9  user:ct(string_smell(TargetClass),
10    ( % search
11      classT(String, _, 'String', _, _),
12      methodT(StringEquals, String, 'equals', _, RetType, _, _, _),
13      classT(TargetClass, _, _, _, Defs),
14      member(Encl, Defs),
```

```
15      operationT(Op, Parent, Encl, [Arg1, Arg2], '==', _),
16      ofType(Arg1, String),
17      ofType(Arg2, String),
18      new_ids([Precedence, EqualsCall]),
19      replaceIdInNode(Parent, ParentTerm, NewParentTerm, Op,
20                      EqualsCall)
21      ),(  % transform
22      set_parent(Arg1, Precedence),
23      set_parent(Arg2, EqualsCall),
24      add(precedenceT(Precedence, EqualsCall, Encl, Arg1)),
25      add(callT(EqualsCall, Parent, Encl, Precedence, [Arg2],
26                StringEquals, [], RetType)),
27      replace(ParentTerm, NewParentTerm)
28    )
29  ).
```

Listing 6.4: JTransfrmer string compare substitution.

6.1.3 Eclipse LTK

The Eclipse LTK is a framework that was designed to allow plugins for different programming languages to provide refactorings. The LTK plugin APIs are complex. They were designed for large range of applications not only refactoring. Besides it requires the user to write code to integrate the refactoring in the visual interface of Eclipse. Therefore it is a system that is hard to learn.

Eclipse LTK has a mechanism that allows the creation of search patterns. However it does not allow the creation of a pattern for comparisons between strings. We use the search pattern mechanism only to search method declarations (Listing 6.5).

```
1  final List<SearchMatch> compares= new ArrayList<SearchMatch>();
2  SearchPattern pattern =
3    SearchPattern
4      .createPattern("codesmell.StringCompare.smellyCompare()",
5                     IJavaSearchConstants.METHOD,
6                     IJavaSearchConstants.DECLARATIONS,
7                     SearchPattern.R_FULL_MATCH);
8  SearchEngine engine = new SearchEngine();
9  engine
10   .search(pattern,
11           new SearchParticipant[] {
12             SearchEngine.getDefaultSearchParticipant()
13           },
14           scope,
15           new SearchRequestor() {
16             @Override
17             public void acceptSearchMatch(SearchMatch match)
18                         throws CoreException
```

```
19              {
20                if (match.getAccuracy() == SearchMatch.A_ACCURATE
21                    && !match.isInsideDocComment())
22                {
23                  IMethod m = ((IMethod) match.getElement());
24                  if(!m.isBinary()) {
25                    compares.add(match);
26                  }
27                }
28              }
29            },
30            new SubProgressMonitor(
31                  monitor,
32                  1,
33                  SubProgressMonitor.SUPPRESS_SUBTASK_LABEL));
```

Listing 6.5: Eclipse LTK search pattern to obtain the method to be changed.

The introspection of the method's AST is done by writing an AST visitor. Listing 6.6 presents the visitor that stores the nodes that represent comparisons between strings on the field `compares` (line 3). The visitor pattern a way of searching the AST in the imperative paradigm. Lines 8- 17 test if the nodes obey the necessary conditions. In line 19 the node meets all the conditions so it is added to the result list.

```
1  public class CompareFinder extends ASTVisitor {
2    public List<InfixExpression> compares =
3                    new ArrayList<InfixExpression>();
4    public boolean visit(InfixExpression node) {
5      System.out.print(".");
6      Object operatorType =
7        node.getStructuralProperty(InfixExpression.OPERATOR_PROPERTY)
8      if(operatorType == InfixExpression.Operator.EQUALS) {
9        InfixExpression ie = (InfixExpression) node;
10       Expression left  = ie.getLeftOperand();
11       Expression right = ie.getRightOperand();
12       String leftTypeBinding  =
13         left.resolveTypeBinding().getBinaryName();
14       String rightTypeBinding =
15         right.resolveTypeBinding().getBinaryName();
16       if(leftTypeBinding.equals("java.lang.String")
17          && rightTypeBinding.equals("java.lang.String"))
18       {
19         compares.add(ie);
20       }
21     }
22     return true;
23   }
```

```
24 }
```

Listing 6.6: Eclipse LTK ast visitor to find the comparisons between strings.

The transformation is very similar to Recoder. LTK adds only one step of complexity to the generation of the source code where the user must construct an object that represents the textual change of the source code.

6.1.4 Tool comparison

The full code of the scripts for our tool, Recoder and Eclipse LTK, for the example of replacing == with `equals`, is in the Appendix B. The JTransformer script is in Listing 6.4. Because the example is simple, the execution time of all scripts is negligible. Therefore, to compare the four tools, we consider the total number of lines of the transformation code, as it can illustrate the complexity of developing with these tools. In Table 6.1 we present the total number of lines, the ones used to search the AST and the lines used for transforming the AST. We excluded from the last two the initialization and boiler plate code, so that we can better compare the logic and imperative paradigms.

The tool that uses less lines of code is JTransformer which uses a logic paradigm. Although our Interpreter plugin is built with JTransformer, it uses less lines of code than JTransformer to describe the search. This is because with our plugin we can express the same computations in a more compact way, without reducing its understandability. The other two, which use the imperative paradigm, need more lines of code to describe the same search.

As for transformations, our tool needs more lines than the others. In JTransformer all the attributes of a new node are set on the same line. While in our tool the user needs to initialize all attributes of a created node, taking one line for each attribute. Recoder and the Eclipse LTK allow the user to create AST nodes from a string of Java code. This way the user does not have to programmatically create the tree node by node.

	Total Number of lines	Searching Lines	Transformation Lines
Recoder	70	13	22
JTransformer	27	8	19
Eclipse LTK	227	22	33
Interpreter plugin	45	2	38

Table 6.1: Number of lines of the code for the transformation that replaces == with a call to `equals`.

6.2 State machine refactoring

Joshua Kerievsky on Refactoring to Patterns [31] expands on the work of Fowler [3] by providing a methodical procedure for programmers to identify smells, and remove them, by refactoring the code to design patterns catalogued on Design Patterns[32]. For our first example we will write a script that automates the refactoring to the state pattern. In order to keep the script simple enough, the conditional code executed for a given state is on a then block of an if where the condition is a comparison between the state field and the state constant. We will be using the code in Listing 6.7 as an example.

```
1 package trafficlight;
2
```

```
3  public class TrafficLight {
4    enum Light { Green, Yellow, Red };
5
6    Light state;
7
8    public TrafficLight(Light light) {
9      state = light;
10   }
11
12   public void tick() {
13           System.out.println("tick");
14     if(_state == Light.Green) {
15         state = Light.Yellow;
16         System.out.println("Yellow");
17     }
18     if(_state == Light.Yellow) {
19         state = Light.Red;
20     }
21     if(_state == Light.Red) {
22         setState(Light.Green);
23     }
24   }
25 }
```

Listing 6.7: TrafficLight class, which is the state machine to be refactored into the state pattern.

We divided the transformation into three transformations:

1. Create the state super class and a subclass for each state.
2. Change enum references to instantiations of the state classes.
3. Change state field assignments with the a call to the state machine setter.

In the next subsections we describe the details of these three transformations.

6.2.1 State superclass creation

The first transformation is responsible for creating the abstract class for the state, its instances and the methods that all states must implement. The transformation allows the user to pass a search as argument that finds the enum that represents the state and the original class that represents the state machine. This adds flexibility for the user to specify the specific targets of the transformation.

6.2.1.1 Create the classes

After knowing the targets of the transformation we can write the search queries to retrieve the relevant information to create the new classes. They can be seen in Listing 6.8. Note that in Listing 6.8 only the searches for the `stateMachine` and `lightEnumRecord` are executed (Lines 10 and 11), while the others will be used later. The main objective of this searches is to ensure that the search for the

state enum and for the state machine class return only one result. This stops the transformation from altering more than one class because of a malformed query. Besides we will be using the records of those classes because we will apply changes to them.

Note how the searches can be defined in terms of other searches. For instance, the search to find the enum instances which uses the search to find the enum class (line 7): in order to search in an imperative paradigm we would have to choose the getter that would give us the smallest collection containing the node that we are looking for and then iterate over that collection. Our declarative queries are more concise because the iteration is done implicitly by the Prolog engine and not by the user.

```
1   function createStateClasses(lightEnum, stateMachineClass) {
2     var transform1 = new Transform();
3
4     var lightEnumName = lightEnum.name();
5     var pkg = _package().enclosing(lightEnum);
6     var lightEnumInstance =
7           lightEnum.def().as(_field).isEnumConstant()
8     var lightEnumInstanceNames = lightEnumInstance.name();
9
10    var stateMachine = transform1.searchOne( stateMachineClass );
11    var lightEnumRecord = transform1.searchOne(lightEnum);
12
13    ...
14  }
```

Listing 6.8: Definition of the searches that will be used on the transformation.

The code for the change of the AST itself is a bit to verbose to be all displayed here. Instead we will show the Java code that results from this transformation in Listing 6.9. The state class and its instances will be defined in separate files which means that we must create a new compilation unit for each one. Furthermore, we have to create a constructor for the abstract class and call it on the sub class constructors.

```
1   public abstract class Light  {
2      protected TrafficLight stateMachine;
3
4      public Light(TrafficLight trafficLight) {
5         this.stateMachine = trafficLight;
6      }
7   }
8
9   public class Red extends Light  {
10     public Red(TrafficLight trafficLight) {
11        super(trafficLight);
12     }
13  }
14
15  public class Yellow extends Light  {
```

```
16        public Yellow(TrafficLight trafficLight) {
17            super(trafficLight);
18        }
19    }
20
21    public class Green extends Light  {
22        public Green(TrafficLight trafficLight) {
23            super(trafficLight);
24        }
25    }
```

Listing 6.9: State classes.

For the creation of the class `Light` we use parts of the AST that define the enum in order to save lines of code. The `Light` sub classes are created by iterating the `lightEnumInstanceNames` as seen in Listing 6.10. The transform object is used to retrieve the concrete names for the enum instance names.

```
1  transform1.search(lightEnumInstanceNames)
2            .forEach(function(className) {
3    var stateClass = transform1.createClass().setName(className);
4    createFileForClass(transform1, pkg, stateClass, folderForClasses);
5    createStateClassConstructor(transform1, stateClass,
6                                constructorMethod, stateMachine);
7
8
9    var extendsClass =
10     transform1.createExtends()
11       .setParent(stateClass)
12       .setSuper(abstractStateClass);
13 }
```

Listing 6.10: State classes creation.

6.2.1.2 State method definitions

We still have to define the methods that will implement the state specific behavior. First we need to add their definition to the abstract class. Then we will move the code from the machine class to separate state instances. New searches are need to identify the methods of the state machine that have behavior dependent on the state and to identify the specific statements that are executed in each state. The code for those searches are presented in Listing 6.11.

For simplicity we assume that all state related code is in the `then` block of an `if` whose condition compares the state field with the state enum. The variable `stateField` holds the search to find the field that holds the state. We assume that the field that holds the state is the only one in the state machine class that is of the type of `Light`.

The search attributed to variable `ifsWithState` will find the comparison between the state field and the enum constant if it is the root of the condition expression or if it is inside a more complex

expression. There is again the need to compare with a subtree to find the comparison between the state field and the access to the state constant. This is expressed by `_if().cond().enclosing( ... )` which will verify that the pattern passed in `...` matches with a sub tree of the condition node. The comparison is expressed by the query `_operation().op('==').arg( ... ).arg( ... )`. The `arg(...)` method specifies that the operation has an argument that matches a specific pattern, without ensuring the order of that argument. This allows us to define a pattern that matches both the `_state == Light.Green` and `Light.Green == _state` cases.

```
1  var stateField   = stateMachine.def().as(_field).type(lightEnum);
2
3  var ifsWithState =
4    _if().cond()
5         .enclosing(
6                     _operation().op('==')
7                       .arg(_fieldAccess().ref(lightEnumInstance))
8                       .arg(_fieldAccess().ref(stateField)) );
9
10 var methodsWithIf = stateMachine.def().as(_method)
11                                  .enclosing(ifsWithState);
```

Listing 6.11: Using the Search API to search the state machine class and the methods that have state dependent behavior.

The above query is an example of the complexity of the AST patterns that can be described with simplicity by our API. By using a logic paradigm to make assertions about the nested nodes we allow the user to define a sequence of implicit nested `if` conditions and loops that otherwise would have to be hand written. We present in Listing 6.12 the pseudo code for an imperative search on the AST that would have the same meaning.

For example the iteration over all `if` nodes in the AST in an imperative approach one would have a for each similar to what we can see in line 3. Our API abstracts that loop in the function `_if()`. The next line on the imperative approach iterates over the subtrees of the if condition which is abstracted on our API in `.cond().enclosing()`. Our API enables the user to treat attributes that hold single nodes the same way he treats attributes that are single nodes. Hiding the cases where an iteration is made and the cases where a simple getter is called.

When a query is passed as argument to another query for (ex: lines 5-8) the imperative approach uses an `if` in order to test if the node matches the given pattern. For example `_operation().op('==')` is equivalent in the imperative approach to the condition in line 5. Lines 8- 12 make the conditions expressed by `.arg(_fieldAccess().ref(lightEnumInstance)).arg(_fieldAccess().ref(stateField))`. In this case the reason for the reduced amount of code in our API is due to the fact that the `.arg(...)` pattern matches any permutation of arguments for the operation. The imperative approach benefits from the fact that the operation == only has two arguments to hard code the condition for all the possible arguments. Still it is more verbose than the same query expressed using our API.

Another aspect in which our API is also better over an imperative approach is the ease with which one can compose queries. On an imperative approach one would have to divide the pieces of the query into separate functions. For example, the functions `isLightEnumInstance` and `isStateField` which represent the queries `lightEnumInstance` and `stateField` defined in the Listing 6.8. In

our approach we do not have to explicitly separate the queries into functions. They can be simply assigned to a variable and then used by the following queries.

```
List<IfNode> ifsWithState() {
  ifs = new List(); // holds the resutl
  for(Node n : AST.getIfNodes())
    for(Expression e : n.getCond().getSubNodes())
      if(e.getOp().equals("==")) {
        arguments = e.getArguments();
        if(arguments[0].nodeType == "fieldAccess"
           && arguments[1].nodeType == "fieldAccess")
          if((isLightEnumInstance(arguments[0].getRef())
              || isLightEnumInstance(arguments[0].getRef()))
             && (isStateField(arguments[1].getRef())
              || isStateField(arguments[1].getRef())))

            ifs.add(n);
        }

  return ifs;
}
```

Listing 6.12: Imperative way of searching the ifs that depend on the state.

Finally on the search `methodsWithIf` we use the `ifsWithState` search to filter the methods of the state machine that depend on state. The part of the search `stateMachineSearch.def().as(_method)` limits the search to the methods defined on the state machine class. From those we want the ones that have behavior dependent on the state. This is done by the rest of the search `(...).enclosing( ifsWithState ).`

Now that we have defined all the queries that we will need we can add the state methods to the state super class and the state specific code to the respective state class. The transformation code will be added to the already used to create the state classes.

We will add an abstract method for each state method on the state super class. The code for that transformation is in Listing 6.13.

```
transform1.search(methodsWithIf).forEach(function(method) {
      var abstractMethod =
        transform1.createMethod()
          .setBody('null')
          .setParent(abstractStateClass)
          .setName(method.getName())
          .setType(voidType)
          .setModifier('abstract');

      abstractStateClass.addToDefs(abstractMethod);
    });
});
```

Listing 6.13: Definition of the abstract methods common to all states.

The results of the query `methodsWithIf` is also iterated to define the methods on the state instances. The code in Listing 6.10 is updated to contain the code that defines the state methods in each state subclass (Listing 6.13). Furthermore we move the code specific to each state to the correct state class.

For each state class we need to iterate the methods that depend on state. For each one of those we get the statements that are run for the corresponding state and move them to the body of the method on the state class. In Listing 6.14, lines 14- 19 define the query that retrieves the `if` nodes whose then block is executed for the current state. Lines 21- 26 contain the transformation that moves the statements present on the then block to the body of the new state method.

```
1   transform1.search(lightEnumInstanceNames)
2             .forEach(function(className) {
3     var stateClass = createClass().setName(className);
4     createFileForClass(t1, pkg, stateClass, folderForClasses);
5     createStateClassConstructor(t1, stateClass, constructorMethod,
6                                 stateMachine);
7
8     transform1.search( methodsWithIf ).forEach(function(method) {
9
10      ...
11      var methodName = method.getName();
12
13      var thisEnumInstance =
14            _fieldAccess().ref(lightEnumInstance.name(className));
15      var ifs = _if().in(method.toSearch())
16                     .cond(
17                        function(cond) {
18                          return cond.enclosing(thisEnumInstance );
19                        });
20
21      transform1.search( ifs ).forEach(function(ifz) {
22        ifz.getThen().getStmts().forEach(function(stmt) {
23          stmt.setParent(body);
24          stmt.setEncl(newMethod);
25          body.addToStmts(stmt);
26        });
27      });
28    });
29  });
```

Listing 6.14: Definition of the abstract methods common to all states.

The first stage of the refactoring is complete. We show in Listing 6.15 the result of our transformation. We only show the `Red` state code because the other state classes are similar to this one. The state

machine class (`TrafficLight`) does not have the code specific to each state. Instead it uses the state instance implementation of the method while keeping the common code.

In line 16 we can see the implementation of the tick method specific to the `Red` state. There we can see that at the current state the code does not compile. We need to change the `state` assignment to reflect the state change on the `stateMachine` and replace the enum references to instantiations of state classes. The first change is described on the next section (Section 6.2.2), while we will explain the other in Section 6.2.3.

```
1  public class Light {
2    protected TrafficLight stateMachine;
3
4    public Light(TrafficLight trafficLight) {
5      this.stateMachine = trafficLight;
6    }
7
8    abstract public void tick();
9  }
10
11 public class Red extends Light {
12   public Yellow(trafficLight trafficLight) {
13     super(trafficLight);
14   }
15
16   public void tick() {
17     state = Light.Red;
18   }
19 }
20
21 public class TrafficLight {
22   Light state;
23
24   public TrafficLight(Light light) {
25     state = light;
26   }
27
28   public void tick() {
29     System.out.println("tick");
30     state.tick();
31   }
32 }
```

Listing 6.15: Source after first transformation.

6.2.2 State instantiations

Replacing the enum references with the corresponding state class instantiation is trivial. Now that we have eliminated the cases where the state enum is a referenced in comparisons, we are left with the cases where the state enum is referenced in changes the state. We also know that the enum references are on methods of the state classes. This gives us access to the `stateMachine` field that we need to pass to the constructor of state classes.

The code for this transformation is shown in Listing 6.16. As the previous transformation, this one is also parameterized with searches that return the enum instances and the state machine class. After writing the search to fetch all the enum references in the state classes (Lines 5- 10) we iterate them to apply the transformation in each node. Line 17 defines the search that returns the state class whose instatiation will replace the enum reference. Line 20 defines the search to find the constructor to be called. Finally, in line 23 we replace the old enum reference by the expression that creates a new object of the corresponding state class.

```
1  function replaceEnumInstanceAccesses(lightEnum, lightEnumInstance,
2                                       stateMachineClass) {
3    var transform2 = new Transform();
4    var stateClass =
5      _class().name(lightEnum.name()).isNot( _class().isEnum() );
6    var stateClasses = _class().extends(stateClass);
7    var stateMachineField =
8      stateClass.def().as(_field).name('stateMachine');
9    var lightEnumReference =
10     _fieldAccess().ref(enumFields).in(stateClasses);
11
12   transform2.search(lightEnumReference)
13             .forEach(function(fieldAccess) {
14     var method = fieldAccess.getEncl();
15     var stateName = fieldAccess.getRef().getName();
16     var classOfNewState =
17       transform2.searchOne(_class().name(stateName));
18     var newParamFieldAccess = transform2.createFieldAccess() ...
19     var constructorForInstatiation =
20       _class().name(stateName).def().as(_constructor);
21     var newStateExpr = transform2.createNew() ...
22     newParamFieldAccess.setParent(newStateExpr);
23     fieldAccess.replaceOnParentBy(newStateExpr);
24   });
25
26   transform2.apply();
27 }
```

Listing 6.16: Transformation that replaces the enum references with the correct state class instation.

6.2.3 Changing the state

Note that in the method `tick` of class Light (see Listing 6.15) there is an assignment of a variable `state` that used to reference the field of the `TrafficLight` class. Because the `state` field is no longer accessible in the state classes that we need to change state in other way. This is why the state classes have a field that holds the state machine they belong to, such that we can change the state of the state machine from the state classes.

The transformation in Listing 6.17 replaces the broken assignments with calls to the state setter of the state machine. Again this transformation is a replacement of nodes (line 43). The state assignments are given by the search hold by `stateFieldAssignInState`. For each of those nodes we retrieve the value being assigned and move it to the new setter call (line 30).

```
1  function replaceOldFieldWrites(stateMachineClass, stateClass,
2                                 stateField) {
3    var stateFieldAssign =
4      _assign().lhs(function(lhs) {return lhs.ref(stateField); });
5    var stateClasses = _class().extends(_class().name('Light'));
6    var stateFieldAssignInState = stateFieldAssign.in(stateClass);
7    var stateMachineFieldSearch = _class().name('Light')
8                                          .def()
9                                          .as(_field)
10                                             .name('stateMachine');
11   var stateSetterSearch =
12     stateMachineClass.def().as(_method).name('setState');
13
14   var transform3 = new Transform();
15
16   var stateMachineField = stateMachineFieldSearch;
17   var stateMachineType  = stateMachineField.type();
18   var stateSetter       = trasform3.searchOne(stateSetterSearch);
19   var setterType        = stateSetter.getType()
20   transform3.search(stateFieldAssignInState)
21             .forEach(function(assign) {
22     var method = assign.getParent().getEncl();
23     var argExpr     = assign.getRhs();
24     var fieldAccess = assign.getLhs();
25
26     var setterCall =
27       transform3.createCall()
28         .setEncl(method)
29         .setReceiver(fieldAccess)
30         .addToArgs(argExpr)
31         .setRef(stateSetter)
32         .setType(setterType);
33
34     fieldAccess
35       .setParent(setterCall)
```

```
36          .setEncl(method)
37          .setRef(stateMachineField);
38
39        argExpr
40          .setParent(setterCall)
41          .setEncl(method);
42
43        assign.replaceOnParentBy(setterCall);
44      })
45
46      transform3.apply();
47  }
```

Listing 6.17: Use setter method of the state machine to change state.

With this final transformation the refactoring is finished. The only classes that changed with the last two transformations were the state classes. As an example, we show the final code of the `Green` state class in Listing 6.18.

```
1  public class Green extends Light  {
2    public Green(TrafficLight trafficLight) {
3      super(trafficLight);
4    }
5
6    public void tick() {
7      stateMachine.setState(new Yellow(stateMachine));
8      System.out.println("Yellow");
9    }
10 }
```

Listing 6.18: The code of the class Green after the state refactoring.

6.3 Fenix refactoring

For a real world application of the tool we chose the open source project FenixEdu [7]. According to their home page ”FenixEdu is a modular software platform for academic and administrative management of higher education institutions.” Its development started in Instituto Superior Técnico and is now the official academic system for the schools of the Universidade de Lisboa [33].

We are now going to describe the transformations that we created for this project. The first is a refactoring that changes legacy code to adopt a new strategy of dealing with strings in Portuguese and English. The second one fixes a problem detected in transactional code.

The specifications of the machine used in our evaluation is in Table 6.2. The amount of cores does not affect the speed at which the queries run, because that process is single threaded. Nevertheless, the amount of RAM is important, when JTransformer is loading the source code to a factbase.

CPU Model	CPU Architecture	Cores	Threads	RAM	Operating System
i7-5820K	x86_64	6	12	32GB	Linux 4.10.1-1-ARCH

Table 6.2: Evaluation system configuration.

6.3.1 Localized Strings

In the FenixEdu project some classes have a different attribute for each language a string of text is to be shown to the user. For example, there were classes that had one attribute for the text in English and another for the text in Portuguese. In order to increase flexibility instead of two attributes, the new classes use a map that relates a Locale to a string, which is implemented by the class LocalizedString. This new approach was used on the new code, however the old approach is still present in old code which led to the opportunity to test our tool.

The transformation replaces the attributes specific for a language with a single one of type LocalizedString. To alter the attributes one must alter the code that changes them. This includes setters, getters, code that calls setters and getters which span over several files.

Such a refactoring implies several specific cases were the need to create new portions of the AST programmatically. One needs to change the AST in the places where the attributes are read and written to, also there is the need to change the places where a getter is used and where an argument is passed to the setters. This variety of cases, were AST manipulation and creation was needed, resulted in verbose code were AST patterns needed to be created node by node. For future work we recommend an interface that allows the creation of AST patterns with snippets of Java code like GenTL does.

The code being refactored is part of a web application that uses JavaServer Pages JSP [34] to describe the web pages. Because JTransformer does not parse JSP, the tool could not refactor all uses of the transformed classes.

The fields containing the name and the translated name are both collapsed into one single field and transformed into the code. Because the types of the arguments and return types are different we must also change the code that interacts with instances of `Bean`. Listing 6.19 shows the use of the class `Bean` before the refactoring and Listing 6.20 shows the same code after the refactoring. Setters and getters for both languages are collapsed into a single one. The distinction between languages is now made in the client code, using Locales.

```
1  Bean b = new Bean("nome em português","name in english");
2  println(b.getName());
3  println(b.getNameEn());
4  b.setNameEn("Outro Nome");
```

Listing 6.19: Example of the usage of the class before the refactoring.

```
1  Bean b = new Bean(
2    new LocalizedString()
3      .with(Local.PT,"nome em português")
4      .with(Local.EN,"name in english"));
5  println(b.getName().getContent(Locale.PT));
6  println(b.getName().getContent(Locale.EN));
7  b.setName(Locale.PT,"Outro Nome");
```

Listing 6.20: Example of the usage of the class after the refactoring.

6.3.2 Atomic Catches

In complex projects like FenixEdu, where many developers come and go, it is easy for some bugs to be introduced unnoticed. The example we are presenting here is caused by the misuse of the transaction system of the project.

Code for database transactions in FenixEdu is written inside the method annotated with `@Atomic`. If the method throws an instance of `DomainException` the transaction is aborted, else the transaction is committed. When faced with invalid records on the database the team hypothesized that there could be methods `@Atomic` where the exceptions were being caught.

An example of a problematic method is the one in Listing 6.21. The code inside the `try` block performs several writes to the database. If one of those fails an exception is thrown. When that happens the transaction should be aborted. However the exception is being caught before leaving the method annotated with `@Atomic`.

```
1  @Atomic(mode = TxMode.WRITE)
2  private void createInsuranceEvent(Person person,
3                                    ExecutionYear executionYear) {
4    try {
5      final AccountingEventsManager manager =
6        new AccountingEventsManager();
7      final InvocationResult result =
8              manager.createInsuranceEvent(person, executionYear);
9
10     if (result.isSuccess()) {
11       InsuranceEvent_TOTAL_CREATED++;
12     }
13   } catch (Exception e) {
14       taskLog("Exception␣on␣person␣with␣id␣:␣%s\n",
15               person.getExternalId());
16       e.printStackTrace();
17   }
18 }
```

Listing 6.21: Transactional method that catches the exception thrown. Catching the exception inside the transaction will result in the transaction to be committed.

The solution to that problem is to move the block of code that interacts with the database to a separate method and move the `@Atomic` annotation to the new method. The code of the original method still logs the exception when it happens, but the transaction is aborted when the domain code throws an exception. The solution to the example in Listing 6.21 is presented in Listing 6.22.

```
1  private void createInsuranceEvent(Person person,
2                                    ExecutionYear executionYear) {
```

```
3     try {
4       createInsuranceEventAtomic(person, executionYear);
5     } catch (Exception e) {
6       taskLog("Exception␣on␣person␣with␣oid␣:␣%s\n",
7               person.getExternalId());
8       e.printStackTrace();
9     }
10  }
11
12  @Atomic(mode = TxMode.WRITE)
13  private void createInsuranceEventAtomic(
14                       Person person,
15                       ExecutionYear executionYear)
16  {
17    final AccountingEventsManager manager =
18      new AccountingEventsManager();
19    final InvocationResult result =
20      manager.createInsuranceEvent(person, executionYear);
21
22    if (result.isSuccess()) {
23      InsuranceEvent_TOTAL_CREATED++;
24    }
25  }
```

Listing 6.22: Transaction code isolated in method anotated with `@Atomic`.

The first step in building the transformation is to find the `try` blocks that need to be fixed. We know that the domain code that interacts with the database throws instances of `DomainException` when an abort is to be performed. Therefore, we are concerned with catches of super classes, sub classes and instances of `DomainException`. Listing 6.23 presents a search for each of these `catch` nodes.

```
1  var catchSuper = _try().encl( _method().enclosing(atomicAnotation) )
2                          .catcher().param(function(param) {
3    return param.type().as(_class).extendedByRec(domainException);
4  });
5
6  var catchSub = _try().encl( _method().enclosing(atomicAnotation) )
7                        .catcher().param(function(param) {
8    return param.type().as(_class).extendsRec(domainException);
9  });
10
11 var catchDomain =
12   _try().encl( _method().enclosing(atomicAnotation) )
13         .catcher().param(function(param) {
14           return param.type(domainException);
15         })
```

Listing 6.23: Search that find the catches of DomainException's in atomic methods.

Of those nodes that we want to filter out are the catch blocks where another exception is re-thrown (see Listing 6.24). Furthermore, we want to exclude those that are present in methods annotated with `@Atomic(mode = TxMode.READ)`. This Atomic annotation enables us to restrict the transaction to read only operations. The only problematic transactions we are fixing are the ones that write to the database during a transaction that should be aborted.

```
1 ...
2 } catch(DomainException e) {
3   log(e);
4   throw e;
5 }
6 ...
```

Listing 6.24: Caught exception and thrown again. Because the exception is re-thrown the transaction will abort.

This example shows the limits of our Search API. It was designed to express the desired values for the node attributes. In order to filter out the nodes that have characteristics that we do not want we use the Transform API. For example, in Listing 6.25 we define a function that checks if a given catch record encloses a throw node.

```
1 function throws(transform, catchRecord) {
2   return transform.search(
3             catchRecord.toSearch().enclosing(_throw())
4          , 1).length == 0;
5 }
```

Listing 6.25: Function that checks whether a node encloses a `throws` node or not.

We use the second argument of the search method to limit our search to one result. Because the JTransformer API does not allow us to get only one result we use the Prologs predicate `limit/2`. If the search returns any result we know that the catch block throws an exception. The same principle is used to filter the catches in methods where the atomic annotation mode is read only.

The transformation itself is trivial. It is just a matter of moving the try block to a new method and call that new method. Because the extracted block is inserted in a method of the same class, we can assume that we have access to the same fields and imported classes. The only concern left is to pass the variables that are not defined in the `try` block as parameters of the new method.

This transformation was applied to the source in fenixedu-academic and fenixedu-ist repositories. The fenixedu-ist repository holds several projects. To search the problematic methods we were able to load all projects to a single factbase. However, JTransformer could not generate the transformed source code due to the large amount of dependencies of the code. Instead of applying the transformation on the common factbase we created a separate factbase for each of the projects that contained the problematic methods and then ran the transformation in each of them.

The transformation described in this subsection was successfully concluded and is now part of the code running in the FenixEdu instalation at Instituto Superior Técnico.

Table 6.3 presents the loading times of the projects' source to a JTransformer factbase (Loading Time), the number of source files, classes and methods they contained. Fenixedu-ist is a factbase that holds both fenixedu-ist-integration and fenixedu-ist-vigilancies. The projects fenixedu-ist-integration and

fenixedu-ist-vigilancies were loaded individually to apply the transformation. Also it should be noted that we excluded database source classes from fenixedu-ist-integration and fenixedu-ist-vigilancies, when loading them to apply the transformations. Those excluded classes are generated from a configuration file and used to access the database. They were not loaded to search the problematic methods, however JTransformer needs to resolve all references for those classes in order to generate the source code.

Note that the transformation on the fenixedu-academic took longer. This is because more methods matched our search than the ones that needed to be modified. The only way to know that those methods that were not supposed to be changed was with human intervention. Therefore we run the transformation for all of them and then, selected only the problematic ones to be submitted to the repository.

	Loading Time	Source Files	Classes	Methods
fenixedu-academic	4122.60s	11225	11225	82633
fenixedu-ist	4441.28s	8307	14884	92147
fenixedu-ist-integration	86.87s	2267	4489	52704
fenixedu-ist-vigilancies	61.36s	2081	2024	22618

Table 6.3: Factbase statistics for the target projects statistics.

Table 6.4 presents the running time of the search for the methods and the number of problematic methods found in each project. Table 6.5 presents the times for the execution of the transformation. We further show how long the search took to run on the fenixedu-ist-integration and fenixedu-ist-vigilancies. The transformation duration column includes the time to search for the methods and to correct them.

The transformations can take long periods of time to finish, as it is the case with the transformation of the fenix-academic project. In such cases we suggest the following methodology of development of the transformation:

- write a search that finds the nodes where the transformation is to be applied to
- chose a result of the search and retrieve its id so that a faster search can be written
- develop the transformation script that concrete method as target until it generates the desired code for that point of the source code
- apply the transformation to all the cases that match the original search

In this methodology the developer tests a faster transformation that targets only one case of it's application. When the developer is sure that the script will apply the desired source code transformations he can run transformation for all results of a search, freeing him to do other tasks.

	Problematic methods	Search Duration
fenixedu-academic	2	498.82s
fenixedu-ist	5	115.80s

Table 6.4: Statistics for the search that finds the problematic methods.

6.4 Chapter Summary

In this chapter we compared our tool with others through a small example. We conclude that using our tool we can write more compact searches, but the transformations are more verbose. Then we give an

	Problematic methods	Search Duration	Transform Duration
fenixedu-academic	2	498.83s	574.35s
fenixedu-ist-integration	3	22.61s	41.21s
fenixedu-ist-vigilancies	2	9.92s	35.91s

Table 6.5: Statistics for the transformation that fixes the problematic methods.

example of a more complex transformation to show that more features of our APIs. Finally we describe two refactoring done on the FenixEdu project with our tool.

The first one changed classes that were used in JavaServer Pages (JSP) [34] files which can not be parsed by JTransformer. Therefore we could not complete the refactoring.

The second transformation, which fixed a bug, was successfully concluded and is now part of the code running in the FenixEdu installation at Instituto Superior Técnico.

Chapter 7

Conclusions

Scripting transformations has many uses, whether it is to update calls to a library that changed its API, or writing a new refactoring that was not implemented yet. The common activities of scripting a refactoring are searching and transforming the AST. With these activities in mind tools like Stratego/XT, Rascal, GenTL, Recoder, InjectJ, JTransformer and JunGL have been proposed. Stratego/XT and Rascal have a broader scope than just source code transformation. While they were designed to rewrite ASTs they lack the implementation of Java source code generation and a more expressive way to search. Over the years the logic paradigm as been shown to be a more successful for program searching. JTransformer and GenTL besides supporting search queries using logic (JTransformer uses Prolog as scripting language) also allow the scripting of transformations. However, the most popular languages of the Tiobe index [35], which is an "is an indicator of the popularity of programming languages", use imperative paradigms.

With simpler APIs than Eclipse LTK, InjectJ implemented on top of Recoder has a purely imperative approach to searching and transformation. Even with more verbose searches the transformations are a matter of imperatively attaching, detaching and creating new nodes.

JunGL represents a middle ground between the two paradigms. It provides the more expressive paradigm for searching and the well-known imperative paradigm to apply transformations to the AST. With this work we had the objective of bringing this ideas originally implemented for Visual Studio to Eclipse. Another important requirement of our project was that it should have an interpreter for allowing the developer to interactively write searches. This enables the developer of the transformation to make sure that he is fetching the right nodes, and if not, he can test the new query without having to compile and run the whole script.

With those goals in mind we chose to use JavaScript as scripting language, and JTransfomer to interpret the queries and perform the transformations. We wrote a Search API that has a simplified logic paradigm and a Transformation API where transformations are similar to database accesses in Object-relational mapping libraries. Both APIs were written in JavaScript and generate Prolog queries that are sent to JTransformer. The Nashorn JavaScript engine provided by Oracle's JDK is used to interpret the JavaScript which calls Java methods.

To present the features of our API we provided an implementation of a refactoring that replaces uses of the operator == to compare strings with a call to the method `equals`. We also provide implementations of the same refactoring using a pure logical tool (JTransformer) and a purely imperative tool (Recoder and Eclipse LTK).

In order to show more features of our APIs we scripted a refactoring that transformed Java code to use the state pattern.

Finally we created two transformations on the FenixEdu source code. One that changes the multi-language implementation on the project, but it was not able to change classes that were used in

JavaServer Pages (JSP) [34] files which can not be parsed by JTransformer. Therefore we could not complete the refactoring.

The code generated by the second transformation, which fixed a bug, was successfully concluded and is now part of the code running in the FenixEdu installation at Instituto Superior Técnico.

7.1 Future Work

In this section we present several ways this project can be improved. The first is removing the dependency from Eclipse. The JTransformer plugin uses classes of the Eclipse's graphical user interface (GUI). Because we use JTransformer as a base for our tool we also need to run the Eclipse GUI. However it would be convenient to run a transformation tool as a console program in order to be easier to use, to be easier to integrate with other tools or to run in environments where there is no access to a GUI like a server or a website.

Another aspect that can be improved is the Prolog generation. We detected that transformations in our API would run slower than on JTransformer's. This is has two reasons: the Prolog generation of the API is not as good as hand written Prolog and the interoperability between the Search API and the Transformation API does not allow the generation of a single Prolog query for a transformation. Gathering all the queries into one Prolog query would reduce the search tree for the Prolog engine.

Although our tool allows the definition of declarative searches, the API for applying the transformations to the AST is still verbose. Moreover the user needs to know the JTransformer AST model in order to be able to use our tool. Like proposed by GenTL [1], it would be ideal to enable the user to express queries and transformations using patterns in the source language (Java).

One of the main goals of this work was to develop a Search API based on the logic paradigm. We relayed on a logic tool as a basis for the implementation of our tool because we thought this was the best way to achieve our goal. However, we did not explore the possibility of implementing the Search API with an imperative tool, which is faster.

Finally, we propose the creation of a library that implements common refactorings. This would lower the barrier of entry for new users because it would allow the creation of more complex refactorings without the user necessarily knowing all corner cases of the underlying refactorings. Moreover it would give the user more assurance that the transformation he is writing is behavior-preserving.

Bibliography

[1] G. K. M. Appeltauer, "Towards concrete syntax patterns for logic-based transformation rules," *Electronic Notes in Theoretical Computer Science*, vol. 219, pp. 113–132, 2008. Proceedings of the Eighth International Workshop on Rule Based Programming (RULE 2007).

[2] F. Tip, "A survey of program slicing techniques," tech. rep., Amsterdam, The Netherlands, 1994.

[3] K. B. M. Fowler, W. O. J. Brant, and don Roberts, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[4] Eclipse. `https://eclipse.org/articles/Article-LTK/ltk.html`. [Online; accessed 05-January-2016].

[5] J. D. W. J. Ferrante, K. J. Ottenstein, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, pp. 319–349, 1987.

[6] W. G. Griswold, "Program restructuring as an aid to software maintenance," 1991.

[7] FenixEdu. `http://fenixedu.org`. [Online; accessed 05-January-2016].

[8] IBM. `http://www-03.ibm.com/software/products/en/z-compilers-optimizer`. [Online; accessed 05-January-2016].

[9] K. Sartipi and K. Kontogiannis, "A graph pattern matching approach to software architecture recovery," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, (Washington, DC, USA), pp. 408–419, IEEE Computer Society, 2001.

[10] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 1, pp. 146–166, 1989.

[11] F. Tip, "A survey of program slicing techniques," *JOURNAL OF PROGRAMMING LANGUAGES*, vol. 3, pp. 121–189, 1995.

[12] "Economics of refactoring." `http://c2.com/cgi/wiki?EconomicsOfRefactoring`, 2009. [Online; accessed 05-January-2016].

[13] J. Y. B. Foote, "Big ball of mud," in *Fourth Conference on Patterns Languages of Programs*, (Monticello, Illinois), 1997.

[14] Eclipse. `http://www.eclipse.org`. [Online; accessed 05-January-2016].

[15] Y. Y. Lee, N. Chen, and R. E. Johnson, "Drag-and-drop refactoring: Intuitive and efficient program transformation," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 23–32, IEEE Press, 2013.

[16] E. Visser, *Program Transformation with Stratego/XT*, pp. 216–238. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[17] K. T. Kalleberg, "Abstractions for language-independent program transformations," 2007.

[18] P. Klint, T. v. d. Storm, and J. Vinju, "Rascal: A domain specific language for source code analysis and manipulation," in *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, (Washington, DC, USA), pp. 168–177, IEEE Computer Society, 2009.

[19] M. Hills, P. Klint, and J. J. Vinju, "Scripting a refactoring with rascal and eclipse," in *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, (New York, NY, USA), pp. 40–49, ACM, 2012.

[20] JTransfomer. `http://sewiki.iai.uni-bonn.de/research/jtransformer/start`. [Online; accessed 05-January-2016].

[21] SWI-Prolog. `http://www.swi-prolog.org/`. [Online; accessed 05-January-2016].

[22] C. C. T. Core). `http://sewiki.iai.uni-bonn.de/research/ctc/start`. [Online; accessed 05-January-2016].

[23] G. Kniesel, J. Hannemann, and T. Rho, "A comparison of logic-based infrastructures for concern detection and extraction," in *Proceedings of the 3rd Workshop on Linking Aspect Technology and Evolution*, LATE '07, (New York, NY, USA), ACM, 2007.

[24] M. Verbaere, "A language to script refactoring transformations," 2008. [Online; accessed 05-January-2016].

[25] M. .NET. `http://www.microsoft.com/net`. [Online; accessed 05-January-2016].

[26] Recoder. `http://recoder.sourceforge.net`. [Online; accessed 05-January-2016].

[27] D. Heuzeroth, U. Aßmann, M. Trifu, and V. Kuttruff, "The compost, compass, inject/j and recoder tool suite for invasive software composition: Invasive composition with compass aspect-oriented connectors," in *Generative and Transformational Techniques in Software Engineering: International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers* (R. Lämmel, J. Saraiva, and J. Visser, eds.), (Berlin, Heidelberg), pp. 357–377, Springer Berlin Heidelberg, 2006.

[28] T. Genssler and V. Kuttruff, "Source-to-source transformation in the large," in *Modular Programming Languages: Joint Modular Languages Conference, JMLC 2003, Klagenfurt, Austria, August 25-27, 2003.* (L. Böszörményi and P. Schojer, eds.), (Berlin, Heidelberg), pp. 254–265, Springer Berlin Heidelberg, 2003.

[29] Oracle. `http://www.oracle.com/technetwork/articles/java/jf14-nashorn-2126515.html`. [Online; accessed 05-January-2016].

[30] JTransfomer. `https://sewiki.iai.uni-bonn.de/research/jtransformer/api/java/pefs/4.1/java_pef_overview`. [Online; accessed 05-January-2016].

[31] J. Kerievsky, *Refactoring to Patterns*. Addison-Wesley, August 2004.

[32] R. H. E. Gamma, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, October 1994.

[33] “Universidade de Lisboa.” https://www.ulisboa.pt.

[34] Orcale. http://www.oracle.com/technetwork/java/javaee/jsp/index.html. [Online; accessed 05-January-2016].

[35] Tiobe. https://www.tiobe.com/tiobe-index/. [Online; accessed 05-January-2016].

Appendix A

Search API - base implementation

```
1  function() {
2    var _queryId = queryIdGenerator++;
3    var _queryContext = "";
4    var _params = genParamsForQuery(_queryId);
5    var query = { // methods common to all types of nodes
6      mainId:   function() { return _params.id; },
7      params:   function() { return _params; },
8      context:  function() { return _queryContext; },
9      queryStr: function() {
10       return _queryContext + termInfo.term
11                            + "(" + printParams(this.params()) + ")";
12     }
13   }
14   termInfo.params.forEach(function(paramName) {
15     query[paramName] = function(nodeQuery) {
16       if(nodeQuery) { // setter
17         if(typeof nodeQuery === "string") {
18           // is setting a value param not a search
19           _params[paramName] = "'" + nodeQuery + "'";
20         } else { // add query to this one
21           _params[paramName] = nodeQuery.mainId();
22           _queryContext =
23             addContextToQuery(_queryContext, nodeQuery.queryStr())
24         }
25         return this;
26       } else { // getter
27         return wrappQuery(this, paramName);
28       }
29     }
30   });
31
32   return query;
33 };
```

```
34 // wraps the search in an object that
35 function wrappQuery(originalQuery, newMainId) {
36   return {
37     params:   function() { return originalQuery.params(); },
38     queryStr: function() { return originalQuery.queryStr(); },
39     mainId:   function() {
40                 return originalQuery.params()[newMainId];
41               }
42   };
43 }
```

Listing A.1: Search API - function that generates a search object. It is parameterized with the configuration for a specific type of node through the `termInfo` variable.

Appendix B

Scripts that replace == with `equals`

```
1  public class StringCompare extends Transformation {
2    public StringCompare(
3      CrossReferenceServiceConfiguration
4        paramCrossReferenceServiceConfiguration)
5    {
6      super(paramCrossReferenceServiceConfiguration);
7    }
8
9    public static void main(String[] paramArrayOfString) {
10     RecoderProgram.execute(
11       new StringCompare(
12         new CrossReferenceServiceConfiguration()),
13       paramArrayOfString
14     );
15   }
16
17   public ProblemReport execute() {
18   try {
19       CrossReferenceServiceConfiguration
20         localCrossReferenceServiceConfiguration =
21               getServiceConfiguration();
22       ProgramFactory localProgramFactory = getProgramFactory();
23
24       ClassType localClassType =
25         localCrossReferenceServiceConfiguration
26             .getNameInfo()
27             .getJavaLangString();
28       SourceInfo localSourceInfo =
29         localCrossReferenceServiceConfiguration
30           .getSourceInfo();
31       List localList =
32         getSourceFileRepository()
33           .getKnownCompilationUnits();
```

```
34        ForestWalker localForestWalker = new ForestWalker(localList);
35        Method equalsMethod = getMethod(localClassType, "equals");
36        TreePrinter printer = new TreePrinter();
37
38        while (localForestWalker.next())
39        {
40          ProgramElement localProgramElement =
41            localForestWalker.getProgramElement();
42          if ((localProgramElement instanceof Equals))
43          {
44            Equals localEquals = (Equals)localProgramElement;
45            if ((localSourceInfo
46                  .getType(localEquals.getExpressionAt(0))
47                           == localClassType)
48                 && (localSourceInfo
49                  .getType(localEquals.getExpressionAt(1))
50                           == localClassType))
51            {
52
53              ChangeHistory ch = getChangeHistory();
54
55              // apply transform
56              Expression arg0 = (Expression)localEquals.getChildAt(0);
57              ProgramElement arg1 =
58                (Expression)localEquals.getChildAt(1);
59
60              ASTList<Expression> args =
61                new ASTArrayList<Expression>();
62              args.add((Expression)arg1);
63
64              MethodReference equals =
65                (MethodReference)localProgramFactory
66                  .parseExpression("(\"2\").equals(1)");
67              printer.printTree(equals);
68
69              ExpressionContainer parentheses =
70                (ExpressionContainer)equals.getChildAt(0);
71              parentheses.replaceChild(
72                parentheses.getChildAt(0), arg0);
73              arg0.setExpressionContainer(parentheses);
74              equals.replaceChild(
75                equals.getChildAt(0), parentheses);
76              printer.printTree(arg0);
77
78              equals.setArguments(args);
79
80              printer.printTree(equals);
```

```
81              ((Equals) localProgramElement)
82                .getASTParent()
83                .replaceChild(localProgramElement, equals);
84              ch.replaced(localProgramElement, equals);
85            }
86          }
87        }
88        return setProblemReport(IDENTITY);
89      } catch(ParserException e) {
90        throw new RuntimeException(e);
91      }
92
93    }
94
95    private Method getMethod(ClassType classType, String methodName) {
96      for(Method md : classType.getMethods()) {
97        if(md.getName().equals(methodName))
98          return md;
99      }
100     throw new RuntimeException(
101       "no method " + methodName + " in class " + classType.getName()
102     );
103   }
104 }
```

Listing B.1: Script that replaces == operator with a call to `equals` using Recoder.

```
1  public class IntroduceIndirectionRefactoring extends Refactoring {
2
3    private Map<ICompilationUnit, TextFileChange> fChanges= null;
4
5    // info about the project that the refactoring changes
6    private String projectName = null;
7    private IJavaProject project = null;
8    public void updateProjectInfo() {
9        if(this.projectName == null || this.project == null) {
10           IWorkspaceRoot workspaceRoot =
11             ResourcesPlugin.getWorkspace().getRoot();
12           IProject eclipseProject =
13             workspaceRoot.getProject("RefactInt");
14           this.projectName = eclipseProject.getName();
15           this.project = JavaCore.create( eclipseProject );
16       }
17   }
18   public String getProjectName() {
19     updateProjectInfo();
```

```
20      return this.projectName;
21    }
22    public IJavaProject getJavaProject() {
23      updateProjectInfo();
24      return this.project;
25    }
26
27    @Override
28    public RefactoringStatus
29      checkFinalConditions(IProgressMonitor monitor)
30      throws CoreException, OperationCanceledException {
31      System.out.println("--␣checkFinalConditions␣--");
32      final RefactoringStatus status= new RefactoringStatus();
33      try {
34        monitor.beginTask("Checking␣preconditions...", 2);
35        fChanges=
36          new LinkedHashMap<ICompilationUnit, TextFileChange>();
37
38              final List<SearchMatch> compares=
39                new ArrayList<SearchMatch>();
40        {
41          System.out.println("--␣Searching␣for␣==␣--");
42
43
44          IJavaProject project = getJavaProject();
45          IJavaSearchScope scope =
46            SearchEngine
47              .createJavaSearchScope(new IJavaElement[] {project});
48
49          SearchPattern pattern =
50            SearchPattern
51              .createPattern(
52                  "codesmell.StringCompare.smellyCompare()",
53                  IJavaSearchConstants.METHOD,
54                  IJavaSearchConstants.DECLARATIONS,
55                  SearchPattern.R_FULL_MATCH);
56          SearchEngine engine= new SearchEngine();
57          engine.search(pattern,
58                        new SearchParticipant[] {
59                          SearchEngine.getDefaultSearchParticipant()
60                        },
61                        scope,
62                        new SearchRequestor() {
63
64            @Override
65            public void acceptSearchMatch(SearchMatch match)
66                                throws CoreException
```

```
67              {
68                if (match.getAccuracy() == SearchMatch.A_ACCURATE
69                    && !match.isInsideDocComment())
70                {
71
72                    IMethod m = ((IMethod) match.getElement());
73                    if(!m.isBinary()) {
74                      compares.add(match);
75                    }
76                }
77              }
78            },
79            new SubProgressMonitor(
80                monitor, 1, SubProgressMonitor.SUPPRESS_SUBTASK_LABEL));
81
82          }
83
84
85          final Map<ICompilationUnit, Collection<SearchMatch>> units =
86            new HashMap<ICompilationUnit, Collection<SearchMatch>>();
87          for (SearchMatch match : compares) {
88            Object element= match.getElement();
89            if (element instanceof IMember) {
90              ICompilationUnit unit =
91                ((IMember) element).getCompilationUnit();
92              if (unit != null) {
93                Collection<SearchMatch> collection= units.get(unit);
94                if (collection == null) {
95                  collection= new ArrayList<SearchMatch>();
96                  units.put(unit, collection);
97                }
98                collection.add(match);
99              }
100           }
101         }
102
103         final Map<IJavaProject, Collection<ICompilationUnit>> projects
104          = new HashMap<IJavaProject, Collection<ICompilationUnit>>();
105         for (ICompilationUnit unit : units.keySet()) {
106           IJavaProject project= unit.getJavaProject();
107           if (project != null) {
108             Collection<ICompilationUnit> collection =
109                                               projects.get(project);
110             if (collection == null) {
111               collection= new ArrayList<ICompilationUnit>();
112               projects.put(project, collection);
113             }
```

```
114             collection.add(unit);
115           }
116         }
117
118
119
120         ASTRequestor requestors= new ASTRequestor() {
121
122           @Override
123           public void acceptAST(ICompilationUnit source,
124                                 CompilationUnit ast)
125           {
126             try {
127                 fChanges.put(source,
128                              rewriteCompilationUnit(
129                                this, source,
130                                units.get(source), ast, status));
131             } catch (CoreException exception) {
132               RefactoringPlugin.log(exception);
133             }
134           }
135         };
136
137         IProgressMonitor subMonitor =
138           new SubProgressMonitor(monitor, 1);
139         try {
140           final Set<IJavaProject> set = projects.keySet();
141           subMonitor.beginTask("Compiling␣source...", set.size());
142
143           for (IJavaProject project : set) {
144             ASTParser parser= ASTParser.newParser(AST.JLS3);
145             parser.setProject(project);
146             parser.setResolveBindings(true);
147             Collection<ICompilationUnit> collection =
148               projects.get(project);
149             parser.createASTs(
150               collection.toArray(
151                 new ICompilationUnit[collection.size()]),
152               new String[0],
153               requestors, new SubProgressMonitor(subMonitor, 1));
154           }
155
156         } finally {
157           subMonitor.done();
158         }
159
160       } finally {
```

```
161       monitor.done();
162     }
163     return status;
164   }
165
166   @Override
167   public RefactoringStatus checkInitialConditions(
168                                 IProgressMonitor monitor)
169             throws CoreException, OperationCanceledException
170   {
171     System.out.println("--␣checkInitialConditions␣--");
172     RefactoringStatus status= new RefactoringStatus();
173     try {
174       monitor.beginTask("Checking␣preconditions...", 1);
175     } finally {
176       monitor.done();
177     }
178     return status;
179   }
180
181   @Override
182   public Change createChange(IProgressMonitor monitor)
183           throws CoreException, OperationCanceledException {
184     try {
185       monitor.beginTask("Creating␣change...", 1);
186       final Collection<TextFileChange> changes= fChanges.values();
187
188       CompositeChange change =
189         new CompositeChange(getName(),
190                             changes.toArray(
191                               new Change[changes.size()]))
192       {
193         @Override
194         public ChangeDescriptor getDescriptor() {
195           String project= getProjectName();
196           String description =
197             MessageFormat.format(
198               "Remove␣==␣for␣project␣''{0}''",
199               new Object[] { project }
200             );
201           String comment =
202             MessageFormat.format(
203               "Introduce␣indirection␣for␣''{0}''␣in␣''{1}''",
204               new Object[] { "a", "b"}
205             );
206           Map<String, String> arguments =
207             new HashMap<String, String>();
```

```
208           return new RefactoringChangeDescriptor(
209                       new IntroduceIndirectionDescriptor(
210                         project, description, comment, arguments));
211         }
212       };
213       return change;
214     } finally {
215       monitor.done();
216     }
217   }
218
219   public RefactoringStatus initialize(Map arguments) {
220     return new RefactoringStatus();
221   }
222
223     public class CompareFinder extends ASTVisitor {
224         public List<InfixExpression> compares =
225           new ArrayList<InfixExpression>();
226         public boolean visit(InfixExpression node) {
227             System.out.print(".");
228             if(node.getStructuralProperty(
229                       InfixExpression.OPERATOR_PROPERTY)
230                == InfixExpression.Operator.EQUALS)
231             {
232                 InfixExpression ie = (InfixExpression) node;
233                 Expression left  = ie.getLeftOperand();
234                 Expression right = ie.getRightOperand();
235                 ITypeBinding leftTypeBinding  =
236                   left.resolveTypeBinding();
237                 ITypeBinding rightTypeBinding =
238                   right.resolveTypeBinding();
239                 if(leftTypeBinding
240                      .getBinaryName()
241                      .equals("java.lang.String")
242                     &&
243                     rightTypeBinding
244                       .getBinaryName()
245                       .equals("java.lang.String"))
246                 {
247                     compares.add(ie);
248                 }
249             }
250             return true;
251         }
252
253         public void preVisit(ASTNode node) {
254             //System.out.println(node.toString());
```

```
255         }
256       }
257
258       // the refactoring
259     protected TextFileChange rewriteCompilationUnit(
260         ASTRequestor requestor, ICompilationUnit unit,
261         Collection matches, CompilationUnit node,
262         RefactoringStatus status) throws CoreException
263     {
264       ASTRewrite astRewrite =
265         ASTRewrite.create(node.getAST());
266       ImportRewrite importRewrite =
267         ImportRewrite.create(node, true);
268
269       System.out.println("--␣realy␣searching␣for␣compares␣--");
270       CompareFinder astv = new CompareFinder();
271       node.accept(astv);
272       List<InfixExpression> compares = astv.compares;
273
274       System.out.println("");
275       for(InfixExpression ie : compares) {
276           System.out.println(ie.toString());
277           AST ast = ie.getAST();
278           Expression left =
279             (Expression) ASTNode
280                           .copySubtree(ast, ie.getLeftOperand());
281           ParenthesizedExpression pe =
282             ast.newParenthesizedExpression();
283           pe.setExpression(left);
284           Expression right =
285             (Expression) ASTNode
286                            .copySubtree(ast, ie.getRightOperand());
287           MethodInvocation mi = ast.newMethodInvocation();
288             mi.setName(ast.newSimpleName("equals"));
289             mi.setExpression(pe);
290             mi.arguments().add(right);
291           astRewrite.replace(ie, mi, null);
292       }
293
294       MultiTextEdit edit= new MultiTextEdit();
295           TextEdit astEdit= astRewrite.rewriteAST();
296           edit.addChild(astEdit);
297           TextFileChange change =
298             new TextFileChange(
299                   unit.getElementName(),
300                   (IFile) unit.getResource());
301           change.setTextType("java");
```

```
302           change.setEdit(edit);
303
304       return change;
305     }
306       @Override
307       public String getName() {
308           return "Fix␣String␣Compares";
309       }
310   }
```

Listing B.2: Script that replaces == operator with a call to `equals` using Eclipse LTK.

```
1  // query for detecting String comparisons with '=='
2  function stringCompareQuery() {
3    // assuming only one String class on classpath.
4    return _operation().op("==").ofType( _class().name("String") );
5  }
6
7  var toEqualsTrans = new Transform();
8
9  var stringEquals =
10   toEqualsTrans.searchOne(
11     _method().name("equals").parent(_class().name("String"))
12   );
13
14 toEqualsTrans
15   .search(stringCompareQuery())
16   .forEach(function(operation) {
17       var leftArg  = operation.getArg(0);
18       var rightArg = operation.getArg(1);
19       var enclosingMethod = operation.getEncl();
20
21       operation.removeFromArgs(leftArg);
22       operation.removeFromArgs(rightArg);
23
24       var parentheses =
25         toEqualsTrans.createPrecedence()
26           .setEncl(enclosingMethod)
27           .setExpr(leftArg);
28
29       leftArg.setParent(parentheses);
30
31
32       // create the equals call
33       var call =
34         toEqualsTrans.createCall()
```

```
35            .setParent(operation.getParent())
36            .setEncl(enclosingMethod)
37            .setReceiver(parentheses)
38            .addToArgs(rightArg)
39            .setRef(stringEquals)
40            .setType(stringEquals.getType());
41
42        parentheses.setParent(call)
43        rightArg.setParent(call)
44
45        operation.replaceOnParentBy(call);
46        operation.remove();
47      });
48  toEqualsTrans.apply();
```

Listing B.3: Script that replaces == operator with a call to `equals` using our Interpreter plugin.