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

João Rodrigues
joao.a.rodrigues@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

May 2017

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

1. Introduction

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 object oriented programs, has been shown to be the solution to avoid the erosion of the programs structure.

However as described in [2], when refactoring one must take into 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' [3]. 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 have 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 [2] are mitigated: the time spent refactoring and the risk of introducing new bugs is significantly reduced.

1.1. Goal

The goal of this work is the development of a transformation tool that combines both the expressiveness of logic for searching with imperative paradigm for transforming the AST. We use JavaScript, instead of Prolog, for scripting refactorings. 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, similar to JTransformer, 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.

2. Background

2.1. Program Querying

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 [18].

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 on 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 [4] to describe queries on the AST and on the program dependency graph (PDG).

2.2. Program Slicing

The source code that implements a given feature may be distributed over several files and methods. This makes it difficult for the programmer to understand the flow of the program. Program slicing is the process of extracting the lines from the source code that influence the values of a given expression. To attain this information the tool must construct a program dependence graph (PDG), which stores information about the dependency of the statements on a program. It can be constructed only having the AST.

2.3. Tool Aided Transformations

On [14] 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 [8] defined a set of meaning-preserving transformations that could be made to a program dependency graph without changing its functionality. From this 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 [5] implement a large set of the Fowler's refactorings. However, as the study [12] 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. Related Work

There are several program transformations tools. On the Table 1 we establish a comparison for the ones relevant for our work. 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

	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 1: Table comparing the source code transformation tools presented.

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 [11] and Stratego/XT [21] 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 instead of the all AST like on Stratego. 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 [10], GenTL [13] and JunGL [20] 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 more simple 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 other hand 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 [17, 9] and InjectJ [9, 7] 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 au-

dience 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.

3. Developed Tool

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 [15] 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 section 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.

3.1. 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. application of the modifications to the source code

We use the JTransformer plugin that implements this 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. This is 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 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 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 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.

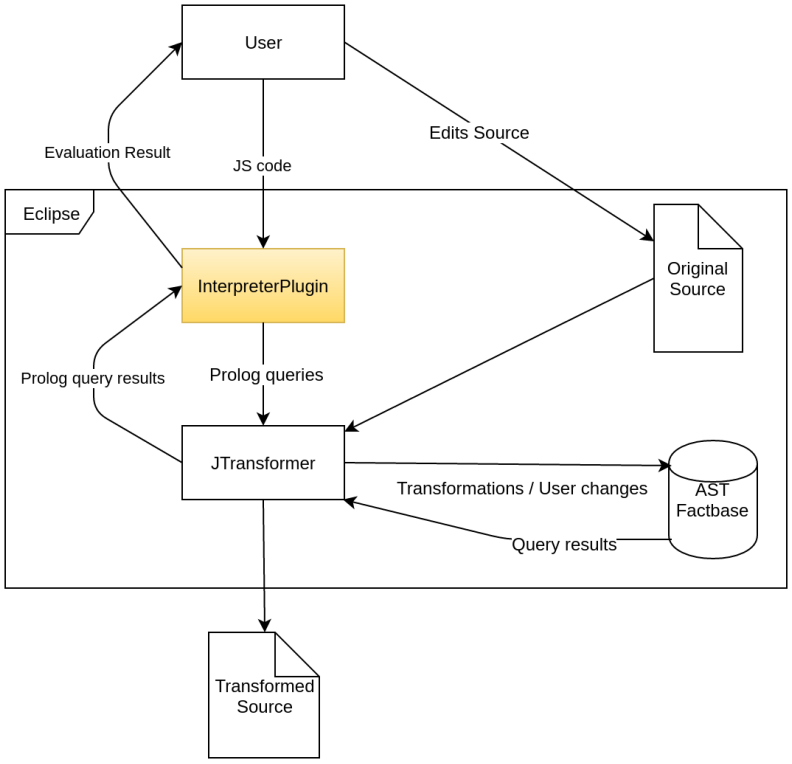

Figure 1: Diagram of how the Interpreter interacts with the user and JTransformer.

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

3.2. Search and Transformation APIs

On this subsection we present the JavaScript APIs for creating searches and transforming the AST, which is the main contribution of this thesis. These APIs will be gradually introduced while creating a script that replaces string comparisons that use the operator `==` instead of calling the `equals` method.

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.

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 in Listing 1.a.

The underscore is there to avoid possible conflicts between the JavaScript keywords when one wants to search for nodes like `if` (Listing 2.b) or `for` (Listing 2.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.

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 1.b). As shown in Listing 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 1.c). Combined with the helper method `ofType`, we can reach only the arguments that are of type String (Listing 2.e). For simplicity we assume that `java.lang.String` is the only class named `String`. 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

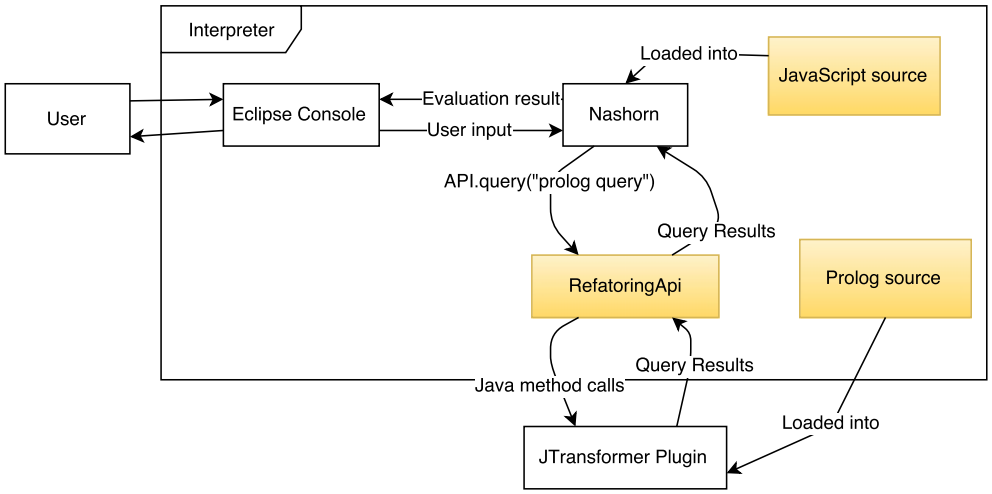

Figure 2: Interpreter diagram

element of that list.

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 these situations that we use the form in Listing 1.d. By passing a function to the selector method, one can call selector methods on it and return the resulting search.

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 use to read and change node attributes. Until the transformation commits the attribute changes are only seen by that the nodes accessed during that transformation, and the AST stays unaltered. We will refer to those proxy objects also as records.

```
// basic search function
a. _<node type>()

// filter by attribute value
b. <node search>.<attribute>(<value>)

// search for the attribute instead
// of the parent
c. <node search>.<attribute>()
d. <node search>.<attribute>(
      <attribute search> ->
            <attribute filter>
   )
```

Listing 1: API for search creation

```
// 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 2: Examples of the use of the search API on 1

The Search API 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 transformation object keeps track of the searched and created records, so that when the transformation is committed, it creates 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 3: Get the records for the desired `==`

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

Listing 4: Example of transformation. It sets the operator of an operation node to `+`

In Listing 3, 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 of their use can be seen in Listing 4. 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 as 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 5: Replacing a `==` 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 5 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. On the case of the `==` node, it demonstrates the use of indexed getters to read the left and right arguments. On 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 on 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, either by replacing the existing node by new ones. The last interactions shown in Listing 5, 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.

4. 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 order to test the capabilities of our tool we compared our tool with other scripting tools by implementing the conversion from `==` to `.equals()`, and we wrote two refactorings for the FenixEdu [6] open source project. The first transformation 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.

4.1. String equals

We implemented the string compare refactoring using the tools Recoder, JTransformer, 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.

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 2 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.

4.2. Fenix refactoring

For a real world application of the tool we chose the open source project FenixEdu [6]. 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 [1].

We applied two transformations in this project with our tool. The first one changed classes that were used in JavaServer Pages (JSP) [16] 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.

We present the statistics for the second transformation. Table 3 lists 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.

Table 4 presents the running time of the search for the methods and the number of problematic methods found in each project. Table 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.

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.

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.

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

	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 3: Factbase statistics for the target projects statistics.

5. Conclusions

Scripting transformations have 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. Over the years the logic paradigm as been shown to be a more successful for program searching. However the most popular languages of the Tiobe index [19], which is an "is an indicator of the popularity of programming languages", use imperative paradigms.

JunGL represents a middle ground between the two paradigms. It provides the more expressive paradigm for searching and the more 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, which can be used in more platforms. 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).

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) [16] 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.

5.1. Future Work

This project can be improved in several ways. 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 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.

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

Table 4: Statistics for the search that finds the problematic methods.

	Problematic methods	Search Duration	Transform Duration
fenixedu-academic	2	498.82s	574.35s
fenixedu-ist-integration	3	22.61s	41.21s
fenixedu-ist-vigilancies	2	9.92s	35.91s

Table 5: Statistics for the transformation to fix atomic methods.

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 [13], 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 are 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 underling refactorings. Moreover it would give the user more assurance that the transformation he is writing is behavior preserving.

References

[1] Universidade de Lisboa. `https://www.ulisboa.pt`.

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

[3] J. Y. B. Foote. Big ball of mud. In *Fourth Conference on Patterns Languages of Programs*, 1997.

[4] 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*, 1(1):146–166, 1989.

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

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

[7] 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.*, pages 254–265, 2003.

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

[9] 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*, pages 357–377, 2006.

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

[11] 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, pages 168–177, 2009.

[12] 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, pages 23–32, 2013.

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

[14] K. B. M. Fowler, W. O. J. Brant, and don Roberts. *Refactoring: Improving the Design of Existing Code.* 1999.

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

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

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

[18] 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, pages 408–419, 2001.

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

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

[21] E. Visser. *Program Transformation with Stratego/XT*, pages 216–238. 2004.