Linköping University | Department of Computer Science
Master thesis, 30 ECTS | Datateknik
2016 | LIU-IDA/LITH-EX-A--16/031--SE

Semi-automatic code-to-code transformer for Java

– Transformation of library calls

Halvautomatisk kodöversättare för Java

– Transformation av biblioteksanrop

Niklas Boije, Kristoffer Borg

Supervisor : Ola Leifler
Examiner : Tommy Färnqvist

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

Having the ability to perform large automatic software changes in a code base gives new possibilities for software restructuring and cost savings. The possibility of replacing software libraries in a semi-automatic way has been studied. String metrics are used to find equivalents between two libraries by looking at class- and method names. Rules based on the equivalents are then used to describe how to apply the transformation to the code base. Using the abstract syntax tree, locations for replacements are found and transformations are performed. After the transformations have been performed, an evaluation of the saved effort of doing the replacement automatically versus manually is made. It shows that a large part of the cost can be saved. An additional evaluation calculating the maintenance cost saved annually by changing libraries is also performed in order to prove the claim that an exchange can reduce the annual cost for the project.

Acknowledgments

We would like to thank Sergiu Rafiliu at Ericsson, who was our supervisor and mentor throughout our thesis work. He pointed us in the right directions when in doubt and had a lot of insight that was of great value. We would also like to thank our supervisor at Linköpings university, Ola Leifler, for keeping us on track with planning, giving us feedback and for asking challenging questions. Also, we would like to thank Tommy Färnqvist our examiner, for giving valuable feedback on our work and how to improve the report. Finally we would like to thank the Spoon research group at INRIA Lille, for letting us use some of their material in form of pictures and for patiently answering our questions about their tool.

Contents

List of Figures

List of Tables

1 Introduction

Manual replacement of code is expensive in both time and money, besides that it is also prone to human errors. A way to get around this problem is to make the replacements to be performed semi-automatic, which in this case means that some manual work will be needed to instantiate the replacements. To be able to do this task a translator or transformer is needed to transform the code to the desired output. One way this can be done is by looking at and altering the Abstract Syntax Trees which the code is partly compiled from.

1.1 Motivation

An ongoing software project is always evolving, new functions and refactoring of the code is done on a daily basis. In some projects even the libraries have to be replaced, it may be due to new functionality, change of programming paradigms or that the new libraries will be generated and updated automatically. In the last case it is not uncommon, probably more of a requirement, that the new library have at least the same functionality as the old library. To have the same functionality as before does not mean that everything is done in the same way as before, more that the same tasks can be performed and that the end results will still be the same as they were for the old library. If it is a requirement that the libraries contain the same functionality, then replacements of the library invocations could be made throughout the project and still keep the same behaviour as before. At Ericsson where this thesis were made, this was exactly the case, where an old library were to be replaced with a new auto-generated library. When this thesis started, both libraries were used and the main reason to remove the old library was to lower the maintenance cost, mostly by not using both libraries but also by the fact that the new library is updated automatically and would decrease the work for the developers and by that save money.

The change of invocations to a library could be very extensive in a bigger project. A project can have thousands of invocations to a library and if this library is to be replaced, all the calls need to be changed. This could take weeks, months or even years to do manually depending on the size of the project. The project at Ericsson using the old library is over 400 000 lines of code with over 4800 invocations to the old library, this would take months, up to a year, for one person to change manually. If a project is constantly evolving, like the project at Ericsson, a problem will occur when manually replacing the libraries. Because the manual replacement takes such a long time, the code that uses the library would have developed. This means that

there will be trouble merging the changes done to replace the library, a lot of the code will have to be changed again and the same problem will probably occur more than once.

What may come to mind, regarding the change from one library to the other, is to change small portions of the project incrementally, this may not be possible since the use of a library often have a high coupling throughout the project. This leads to that if a change of library functionality is done in one place, it will affect several other places and there will be compilation errors. As a result of this all the replacements need to be done at once.

To both lower the time spent on changing invocations, from months to weeks, together with making it possible to do all the changes at once, a semi-automatic tool for doing the replacements is proposed in this thesis. The idea of this tool is to semi-automatically write a file with rules and then transform the source code according to these rules. The reason for having the file written in a semi-automatically fashion is that some rules need a lot of logic, which is hard to do automatically, or the two libraries are so different from each other that the rule need some manual input.

1.2 Aim

This thesis looks at semi-automatic code transformation as a possible means to move the boundary of when a piece of software is too large to be replaced efficiently, together with reducing the effort spent on transforming large pieces of software. The aim is to evaluate whether semi-automatic refactoring is a possible way of doing library replacement in an efficient way, with lower cost and less time spent on interchanging, compared to manually replacing the library.

The purpose of this thesis is to both study and implement a code-to-code transformer. This transformer shall take some part of a code base and transform it to new code. An example from reality is that there are two different libraries which are almost the same in terms of functionality. To make programming easier, to simplify the maintenance and lower the maintenance cost, the references to the older library should be replaced with references to the new library. The transformer shall do the replacement semi-automatically which will speed up the replacement of libraries and minimize the manual work together with the faults that comes with manual replacements. In order to replace code with its equal counterpart automatically, methods must be found, not just by its name but also by the class containing the method. Methods throughout the code can have the same name but originate from different classes or even libraries, then it is crucial to know which class and library each method have so the right methods are chosen. A way of doing this is to look at the Abstract Syntax Tree which have this kind of information. The aim is to remove as much of the dependencies as possible from the old library.

1.3 Research questions

1. How much of a transformation between two libraries is it possible or economical to automate?
2. How can a partly automated tool for code transformation decrease maintenance cost in terms of needed work in person-months?

1.4 Delimitations

This thesis tries to narrow the scope of the transformation to the case of library replacement. This means that a new library will replace an old one in the entire code base. All functionality used in the old library must exist in some form in the new library.

The project at Ericsson where this thesis were made consists of Java code. This is considered, together with the fact that Java has the advantage of using abstract syntax trees when

parsing the code. Therefore the thesis will focus on library replacements in Java. These syntax trees can be altered and that will result in altered source code, more about this in chapter 3. However, abstract syntax trees are not exclusive to the Java language and the concept of library replacements talked about in this thesis should be applicable in other languages too [46].

Another delimitation that is done is that this thesis studies a transformation that is primarily done on a set of test cases. Test cases should be relatively independent of each other which will make the transformation less cumbersome.

Refactoring can be divided into some different activities and these are shown in section 3.3. One of those activities talks about maintaining all software artifacts in a project, this is however out of the scope in this project. This thesis only look at refactoring code, mainly because all other artifacts like documents are already in place for the new library.

2 Background

The work behind this thesis report were made at a company called Ericsson, which is a big company within communications networks, telecom services and support solutions. About 40% of all mobile traffic in the world passes through network equipment provided by Ericsson [16]. To be able to provide all these services, a large code base is needed and in this thesis report a part of this code base is considered. This small part is still about 400 000 lines of code and require a lot of maintenance. Today there are two libraries in this part of the code that essentially does the same thing, this comes from the fact that Ericsson used to write their own libraries by hand but today generates them automatically from a meta-model. When updating the meta-model, the new library will also be updated, this is a great improvement over the old library where all the work with the source code had to be done manually by the developers and maintainers. Besides just doing the updates of the library automatically, there were two main reason to create this new library. The first reason was that the design started to erode away, which is common in software that evolves over time [38]. The second reason was to cut down the cost and effort of maintenance, this depends heavily on the fact that the new library is made automatically and will lower the time spent on altering the source code by hand.

The work replacing the libraries is perceived as too big to do by hand, as there might be hundreds of method calls to the old library in just one file. There are specific method calls that are used many times each and rather than to do copy paste operations on all the places where a replacement is needed, it is logical to try to do the replacements in a more orderly fashion. A tool could replace all old method calls that it recognizes as it goes along in some unit of code, while manual copy and replace would simply change all calls of a certain type, leaving other old calls untouched. There is also the possibility of using a script of sorts to replace strings, in order to do the transformation, these can be made to cover all the cases of method calls. However, this thesis aims at using some of the built in functionality of the Java compiler to do a deeper analysis of the code to be transformed. This will give the transformer tool a way to reason about things like types, methods and argument types in an active context. With this reasoning an implementation of the code transformer will be easier and more complete than doing a script that is only looking at strings.

There are however some requirements to be able to do these transformations. The new library that is going to replace the old, need to at least have all the functionality that the old

library have. If this is not the case, some transformations cannot be made and the code will still have dependencies to the old library.

Ericsson have several similar situations, where two libraries provide the same functionality. Therefore, another reason to make a tool for transformation is that it can be reused multiple times to solve similar problems.

As said before, the libraries considered in this thesis are very similar, not only in the functionality that they provide but also in their structure. This makes it easier for a programmer to switch from using the old library to the new one. As an example from the actual code, the methods in the old library are as follows:

```
createIkev2PolicyProfileMO(Object id):Ikev2PolicyProfileMO
createEthernetPortMO(Object id):EthernetPortMO
getEthernetPortMO(Object id):EthernetPortMO
```

Listing 2.1: Old library

The equivalent methods in the new library:

```
createIkev2PolicyProfileMo(String id):Ikev2PolicyProfileMo
createEthernetPortMo(String id):EthernetPortMo
getEthernetPortMo(String id):EthernetPortMo
```

Listing 2.2: New library

Here, both the similarities and the typical differences are seen. Both libraries have their own types, which are not interchangeable with each other. Secondly, even though the shown methods from both libraries takes strings as arguments, for some reason the old library allows any Java object as input to its version of the shown methods.

Another example from the actual code shows a difficulty in the transformations, here the the older version of the library method plainly takes in an int as argument, listing 2.3, while the new version uses a struct class to wrap a BigInteger, listing 2.4.

```
public setCommittedBurstSize(int cbs): ShaperMO
```

Listing 2.3: Old library

```
public setCommittedBurstSize(CbsInShaperStruct committedBurstSize): ShaperMo

public class CbsInShaperStruct extends AbstractStruct {

    public getBytes(): BigInteger

    public setBytes(BigInteger bytes):CbsInShaperStruct

}
```

Listing 2.4: New library

3 Theory

The chapter will start with some background about context-free grammars, which is a part of the abstract syntax tree (AST) in the next section. AST is the basis for all the transformations done in this thesis. This will be follow by some background of software refactorings. Refactorings have a direct correspondence to graph transformations [32]. A program can be seen as a graph and the refactorings are seen as graph production rules. Then a selected and performed refactoring will correspond to a graph transformation. When an implementation of a source code analysis tool is made with the Eclipse JDT library later in the thesis, it uses ASTs to retrieve information used in the analysis. JDT is a plug-in to Eclipse and section 3.4 goes deeper into this plug-in. After the analysis part, the next thing handled is how to do the actual transformations, these will be done with the help of a library called Spoon. The Spoon library is made, among other things, to transform Java source code. Spoon code is written in plain Java, to read more about Spoon look at section 3.6. To know if and how transformations can help with maintenance cost, some calculations are needed. This can be seen in the last section of this chapter.

3.1 Context-free grammars

Context-free grammars are a way of describing a special kind of languages, the context-free languages. The context-free grammar consists of 4 parts, a set of terminal symbols, a set of non-terminal symbols, a set of productions (or rules) and a designated non-terminal start symbol. This can be expressed as:

$$G = (N, \Sigma, P, S) \tag{3.1}$$

where

- N is the non-terminal symbols
- Σ the terminal symbols
- P the productions

and

- S the start symbol

The terminal symbols (Σ) are symbols that cannot be changed by the productions and are the basic symbols that forms the strings. In figure 3.1 the terminal symbols (Σ) are written in bold and the keywords if and else are examples on such terminal symbols (Σ). Non-terminal symbols (N), in contrast to terminal symbols, are symbols that can be replaced by the productions (P). These symbols represent sets of strings and helps to define the the language created by the grammar. Each production (P) or a rule has a head symbol, also called left-hand side which can be seen as the left side of the arrows in figure 3.1. These head symbols are strings that can be replaced by following some grammar in the body. The body, also called the right-hand side, can be seen as the right side of the arrows in figure 3.1. It is a collection of terminals (Σ) and non-terminals (N) and describes a way of how the head symbol can be constructed. A production (P) is a way to describe the relation between a non-terminal head symbol and one or more terminal and non-terminal tail symbols separated by separator signs. Each of the symbols in the tail of a production denotes a choice and impacts the final string.

A context free grammar can be used in two ways, to generate a string in the language represented, or to parse an existing string to see if it belongs to the language. By starting at the start symbol (S), which is a non-terminal symbol (N), and following some path of productions, expanding every non-terminal (N), the result might end up with a string with only terminals (Σ) which is a string within the language. The parsing works the other way around, taking symbols from a string and matching them against the tail of a production, giving a non-terminal (N). If all the non-terminals (N) adds up to the start symbol (S) then that string is in the language. However, that there are no productions leading to the start symbols (S) from configuration does not mean that the string is not in the language, rather, all possible paths from the final string back through productions must be taken in order to assure that the string is not in the language [3].

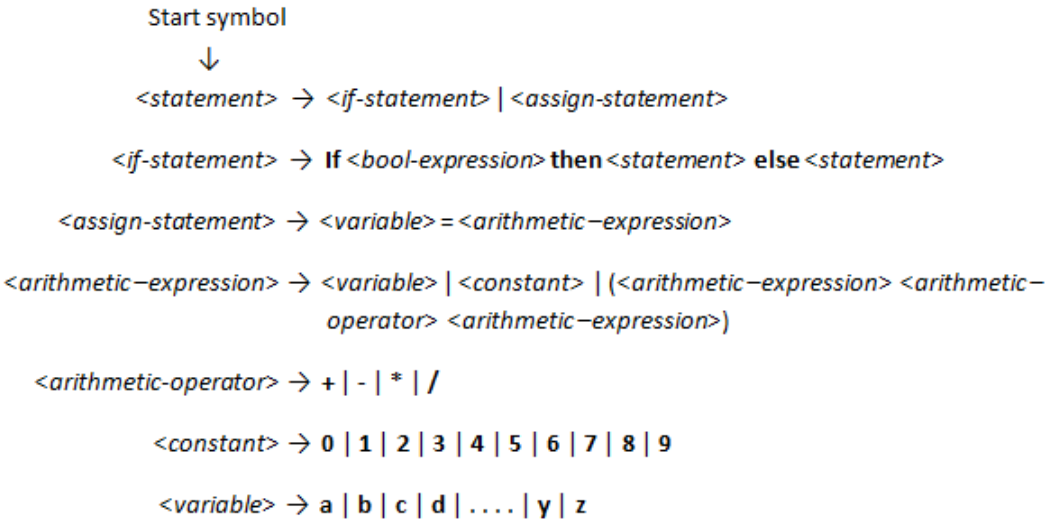


Figure 3.1: An example of a context-free grammar [30]

3.2 Abstract Syntax Trees

Abstract Syntax Tree (AST), is a tree that describes how a particular string inside a context-free language is derived from the grammatical rules of the language. This tree can be used to represent the structure of source code on an abstract level. In compilers, the parser often

produces ASTs where they represents the hierarchical syntactic structure of the source code [3]. The purpose of this is to serve as an intermediate representation for the compiler and can be used in many steps, like the intermediate code generator.

In an AST the start symbol of the grammar is the root and edges going to other nodes are parts of the productions. In figure 3.2 the root can be seen as the - with edges to two other nodes where on is a leaf node represented by a letter (a or c). The leaf nodes in the AST represents the terminals in the context-free grammar [3]. Branches in the AST arise when the production rule used has concatenated symbols. In this case, each of the symbols gets its own edge and must be terminated.

ASTs is not only used for representing easy expressions like the one in figure 3.2, it can express a whole program. However not all information from the code is included in the AST, for example comments, parentheses and brackets is not included. In listing 3.1 a pseudo code of the algorithm insertion sort is shown and in figure 3.3 the corresponding AST is shown. The AST is read like a depth-first traversal that visits the children in left-to-right order. This traversal starts at the root and then visits each children recursively in a left-to-right order [3]. In figure 3.3 this means that it starts at "statement sequence", then goes down the left branch until it reaches "var: i". Then it goes back up to the "compare op: <" and down to "var: a", it keeps on doing this traversal until all the nodes are visited.

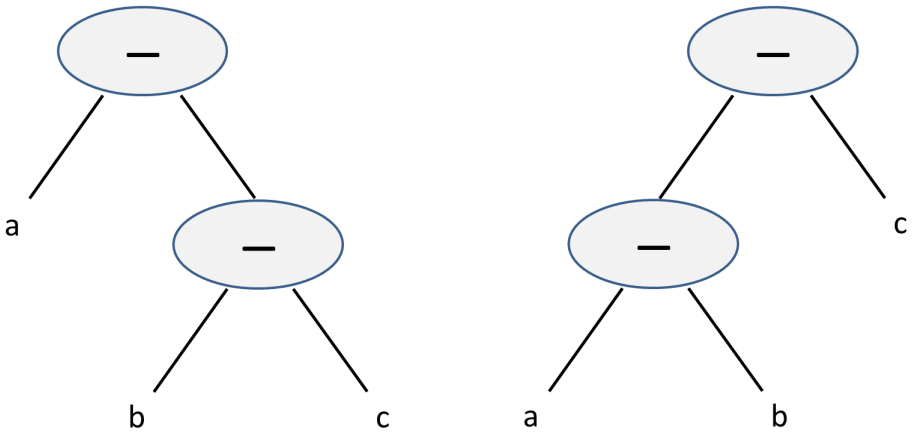


Figure 3.2: Two ASTs with different interpretations of expression $a - b - c$

```
for i == 1 < length(a)-1{
    j = i
    while j > 0 && a[j-1] > a[j]{
        swap a[j] && a[j-1]
        j = j - 1
    }
}
return a
```

Listing 3.1: Implementation of insertion sort

Some compilers does indeed use AST:s to understand computer programs. Therefore it is convenient to make the computer language unambiguous, so that there is no doubt about what the programmer intended. An unambiguous language is one where, for every string in the language, there is only one valid AST for that string. Therefore, there is no doubt about the AST interpretation of the string. For example, there is much difference between the interpretation $a - (b - c)$ and $(a - b) - c$ of the expression $a - b - c$ as seen in figure 3.2. Of course the compiler can choose one of the interpretations arbitrarily, but it is better to make the choice consequent, if only to make it choose one of the two interpretations every time.

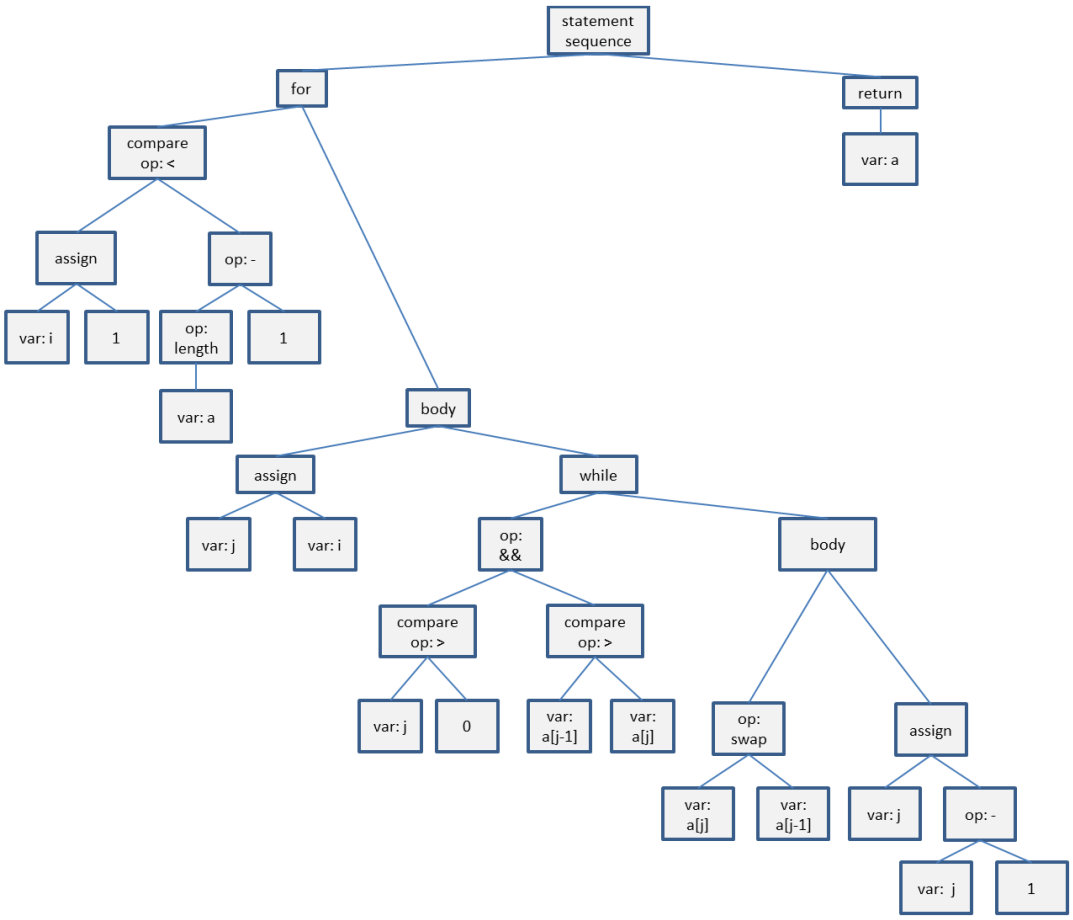


Figure 3.3: AST for the insertion sort algorithm in listing 3.1

3.3 Software Refactoring

In traditional software engineering, requirements are often made for the system to be developed and heavily depends on documentation. In later years agile development approaches have been popular, where an open dialog between the business people and the developers is a corner stone. Instead of requirements, the agile way is to make user stories which are descriptions of features that will provide business value for the customer [33]. When the development starts, a decision is made to either use the traditional way of software development with requirements or to use agile development strategies with user stories. The user stories are more flexible and often changes after dialogs with the customer. As the project proceeds there is a big chance that it will get more requirements or added user stories, code will get altered and the intended code design will fade. The code will not longer follow a good practice and one way to get the code back to a well-design code is to do refactoring. Refactoring is to improve the internal structure of the software system but not to change the external behaviour [38].

Refactoring can be divided into six different activities [32]:

1. Find where to do the refactoring.
2. Which refactorings shall be made on the identified places.
3. Guarantees for preservation of behaviour.
4. Apply refactoring.

5. Evaluate the effect from the refactoring by quality characteristics or the process.

6. Maintain the compatibility between all software artifacts and the new code.

In this thesis project, where the change of libraries are the main task, the focus will not be on item 5 and 6. Item 5 talks about the quality characteristics. It is a part of why this library refactoring or transformation is performed in this thesis. Instead of having dependencies to two libraries in the code, after the refactoring, the code will only have a dependency to one of these libraries. This will lower the complexity and also increase the maintainability. However, it is assumed that the replacement will increase the overall quality of the software, but an evaluation of the full effects are beyond the scope of this thesis. Item 6 is not in the scope of this project and the new library will already have documentations that can be used instead of the documentations to the old library. Therefore the activities in item 1-4 will be considered a little more thoroughly.

3.3.1 Identify places for refactoring

The first difficulty with refactoring is where to apply it. Refactorings can actually be performed in more levels than in the source code level. It can also be preformed in more abstract software artifacts such as requirement documents or design models. In this section the focus will however be on source code and how to find code that needs refactoring. One approach to discover refactoring candidates is to look at program invariants. An invariant is a condition, or a set of assertions, that must always be true during the lifetime of an object. If this holds, a program is said to be valid. Another way of putting it is to say that it is a condition that shall be contained to ensure some desired effect. An example of this can be that the state of an object shall remain the same from the end of the constructor to the start of the destructor if no method is executed to change its state. The invariants can then be used to find parameters that are no longer used in a method body, the value is constant or the information can be computed by other information in the code. From this information, the parameters that are no longer used can be removed. Both computations of invariants together with identification of the candidates for refactoring can be made automatically with good results [29].

Identification of bad smell in code is a good way of detecting parts of a program that needs refactoring. Bad smell in code is structures that have the possibility of refactoring or that really should be refactored because it does not follow the convention of the project or programming standard. Bad smells includes duplicate code, long methods, large classes or long parameter lists to name a few [38].

To find duplications, also known as clones, an analysis tool can be used. Magdalena Balazinska et al have developed such an analysis tool where the analysis builds upon a matching algorithm for matching code fragments [7, 6]. The algorithm aligns syntactically unstructured entities like tokens. Tokens are the variables, operators, delimiters etc. that are used in, for example, the compiler when processing source code. The tokens are then used to measure the distance between the two fragments by looking at how many inserts and/or deletions that are necessary to make the transformation from one fragment into the other, the distance between these fragments, calculated in tokens, are called the difference. The result from the algorithm is shown as the tokens that needs to be inserted and/or deleted. When the smallest difference is obtained, the sequence of tokens need to be linked to their entities in the programming language at an appropriate level of abstraction. A good choice is an AST, which can be used for this purpose because it is easy to analyze and extract entities [6]. In this AST, the tokens forming the differences are linked to their corresponding elements and each token only have one node in the AST. When tokens in a consecutive order belongs to a single difference, that is the tokens needed to be added or deleted, the first ancestor node is found

corresponding to those tokens. Now the set of differences, all the difference between clones, can be obtained as:

$$P(Trees_1 \cup Trees_2) \tag{3.2}$$

where $P(s)$ is the power set of s, in this case s is the union of $Trees_1$ and $Trees_2$. Each AST consists of sub-trees, which themselves are ASTs, and the two trees, $Trees_1$ and $Trees_2$, are all the sets of sub trees belonging to the ASTs [6].

Together with this, a context analysis are made to look at context dependencies to influence the choice of refactoring. How much a refactoring will cost in the sense of transformations between common code, particular code or all the code and also to show how much coupling there is between shared functionality. Low coupling will make transformations possible without a big overhead, on the other hand, high coupling will make the transformations harder. To remedy this, differences could be encapsulated and then decoupled from the shared code, then the transformations would be easier [6].

3.3.2 Refactoring methods

Refactoring can be made on the model level as well as on the source code level. Models are used to raise the level of abstraction, as a result, complex activities such as refactoring also moves over to the model level. On the model level the refactoring is called model refactoring and is the design level equivalent to source code refactoring [42]. The principle of model refactoring is the same as for source code refactoring, the model gets restructured to improve some attributes but preserves the behaviour. Model refactoring is enabled by inconsistency detection and resolution, some examples of inconsistencies are incompatible declaration, inherited association reference or missing instance declaration as can be seen in table 3.1.

Good examples of refactorings that are applicable on both the source code- and the model level are the extract super class and pull up method refactoring. The extract super class refactoring takes two classes with similarities in behaviour and creates a common ancestor for them [34]. The 2 classes for which the new class is created should be related as well as have similar behaviour in order for the refactoring to improve structure. Alternatively, the two classes could already have a common super class but a new super class is wanted in an intermediate layer between the classes and the old super class [34]. The pull up method refactoring can be used if two classes with a common super class share a method with the same name, signature and behaviour [34]. Then, this method can be moved upward in the class hierarchy tree. A situation when the pull up method is a bad idea is when the nearest shared super class should not contain the method in question. Then, the create super class refactoring provides a solution by inserting an intermediate super class. The pull up method and the create super class works in tandem, if no sufficient super class exists for the classes for which one wants to do the pull up refactoring, then it can be created by using the create super class refactoring.

An approach to do model refactoring with inconsistencies is called inconsistency resolution. With this approach an UML-model is refactored in user specified ways and when a refactoring step has been completed, a set of queries are sent to the model. The queries describe standard inconsistencies that have been identified by Ragnhild Van Der Straeten et al. and can be seen in table 3.1 [42]. If the model is found to be inconsistent, an attempt to resolve the inconsistency is made. This is done by asking the user for guidance. The user gives a description of a rule for how the inconsistency is to be handled. If a particular inconsistency reoccurs at another place in the model, the rule given by the user can be reapplied. In this way, the users own preferences for inconsistency resolving are woven into the model. It can happen that new inconsistencies are created when an old one is resolved. These are managed iteratively in the same way as before, by asking the user for preferences or if these are already given, automatically resolving the issue. The model is said to be consistent when it is syntactically correct and some specific behavioral properties are preserved.

	Inheritance	Incompatible declaration	Incompatible behaviour	Unconnected type reference	Inherited association reference	Missing instance declaration
Add Parameter	X		X	X		X
Extract Class	X	X	X		X	X
Move Property		X			X	X
Move Operation	X	X	X		X	X
Pull up Operation					X	X
Pull down Operation			X		X	X
Extract Operation	X		X			
Conditional to polymorphism	X		X		X	X

Table 3.1: Model refactoring and inconsistencies [42]

The similarities between these refactorings and the approach taken in this thesis is that both uses user input to get preferences about how to change a piece of software and that rules are used for the transformation/refactorings. The differences consist of that this thesis try to change the software at a lower level, in the AST. Also, the software transformation is not really a refactoring in the traditional sense, even if the goal of this thesis still is to rearrange the code and giving it better maintainability characteristics without changing its behaviour.

There are no standards for doing transformation between models, however some categories for transformations have been proposed [14]. The categories classifies different transformations and a way to do the classifications and describe the different model transformations is to look at whether they have the following properties:

1. Use of determinism
2. Scheduling per transformation opportunity or per transformation type
3. The scope of the transformation
4. The relation between source and target
5. Iterative and recursive transformations
6. If the transformation can have different phases
7. Whether the transformation is bidirectional

Use of determinism means that applying the same transformation any amount of times on the same piece of code yields the same result. Scheduling per transformation opportunity means going through the model and taking transformation opportunities as they come along, while scheduling per transformation type means taking all instances of a a certain type of transformation at a time. The scope of the transformation tells whether a transformation applies to all parts of the model or if it is restricted to some part of the model. The relation between the source and target property tells if the source and target models are the same model or if a new model is created in the process of transformation. The transformation may have a single or multiple phases. If the transformation has several phases, only certain transformation processes might be available in every single phase. Finally, a transformation might be unidirectional or bidirectional, depending on whether the transformation rules applied has

an inverse transformation or not [14]. These are thought to include the majority of all model transformation, but other work introducing other classifications are mentioned. The differences in the model transformations classified are in how model elements are represented, treated and handled. Some transformation tools for doing model transformations, fitting in one or several of the classifications are also identified. [14].

3.3.3 Rules

The model transformation classification mentioned in section 3.3.2 are primarily concerned with rule based model transformations. Rules can be used to describe single program transformations. Rules can naturally be grouped into compound rules. Strategies is a way to decide where certain rules shall be applied and in what order a set of rules should be applied at some position in a model, a tree or in code [43]. Strategies are also used to tell in what phase of the transformation a type of rule shall be applied, given that the transformation has different phases[43, 14]. Rules can also be used for inconsistency resolution when transforming code[42].

3.4 Eclipse JDT

Eclipse JDT is a set of plug-ins that provides APIs to the Eclipse platform, which adds functionality in order to provide a fully-featured Java IDE. IDE is short for Integrated development environment which is a software application with tools for the programmer, some common examples are a source code editor and a debugger. One of the plug-ins in Eclipse JDT is JDT Core, this plug-in has infrastructure for modifying and compiling Java code and is the most frequently used in this report. In JDT Core there is a Java Model and an API that lets programmers to navigate through the Java element tree. In this plug-in there is a package called *dom* which has support for examining ASTs and also a package called *dom.rewrite* that supports rewriting of these ASTs [20].

3.4.1 Java Model

To get objects that can be used for creating, editing and building programs in Java, a model is needed. In JDT this model is called the Java model and is a set of classes that implements Java specific behaviour for resources. With these implementations Java resources can be decomposed into model elements [21]. The model elements, also called Java elements can then be used to traverse and query the model. There are a 17 different Java elements in the JDT Core that all represents different variables, parameters, methods etc. In Eclipse IDE some Java elements is shown in the package explorer. A project is seen as an IJavaProject with the IJavaModel as the root Java element corresponding to the workspace. Then the ICompilationUnit is seen as the representation of a Java source file.

3.4.2 AST API

Modifications and analysis of source code in JDT is done with the CompilationUnit which is the root of an AST [21]. To create a CompilationUnit when having existing source code you use the ASTParser, which is a parser that takes Java source code and creates ASTs. When parsing, the resulting AST will have all the elements from the source code and they will be in the right positions in the AST corresponding to the source code. Before parsing some different options can be set. Two crucial options are the source path and the class path, if these are set-up correctly bindings can be activated when making the ASTs. Bindings simply provides binding information for all the nodes in the AST and can be seen as connections drawn between the different parts of the program. The bindings have a lot of useful information that is crucial to make transformations, it provides declaring class, return type, parameter types

among lots of other information. To make bindings is however a costly operation and should not be used more often then necessary [22].

There are two different ways to traverse an AST to find a specific node out of the different kinds of nodes that it is composed of, or to perform some sort of calculation upon the tree. Before traversing the syntax tree however, it is necessary to create the AST from the code and to parse it into a CompilationUnit. When the CompilationUnit has been created, the AST can either be traversed by recursively or iteratively extracting children of a particular node or by the use visitors. When traversing the AST recursively a method that takes a node is used. The method can operate on the node in order to perform calculations. It also find all the children of the node in order to call itself with these children as new arguments. To be able to traverse the tree in this way, the complete structure of all of the ASTs nodes must be known. For example, to get to a method declaration inside a class, it is needed to go through the abstract type declarations to get a type. The type can then be used to get the types body declaration. Finally, the body declaration of the type contains the method declarations which can then be retrieved and the specific method declaration can be found. There are many kinds of nodes for describing the Java language, and the granularity difference is vast. If children to nodes are forgotten when iterating over the structure, those corresponding branches in the AST are never reached and cannot be operated upon.

3.4.3 Visitors

If visitors are used instead of traversing the AST, as mentioned in 3.4.2, different visitors can be made with arguments that states which node type to look for. The visitors are as the name suggests a use of the visitor pattern. Visitors represent operations that can be made on elements that belongs to a data structure. Defining new operations with a visitor can be done without altering the classes where the elements operate [44].

Visitors work well when the data structure to be traversed consists of many different kinds of objects, and when several algorithms will be applied to the data structure [40]. An example would be if different rules of transformation should be applied for different kinds of nodes.

3.5 String metrics

During the course of this thesis, a measurement for string comparison is used. For example, method names or class names are compared between the old and new library. An algorithm called Levenshtein distance is used for this, the algorithm gives a measurement about how many edit operations are needed to go from one string to another[45].

Other string metrics can of course be used but William Cohen et al. shows that Levenshtein distance has a similar performance to other top of the line algorithms and works particularly well for non-trivial structures [13]. Besides, the algorithm was well known to the authors of the thesis. Levenshtein distance can find near matches that cannot be found when using a direct matching approach. It works for strings of different lengths contrary to the Hamming distance which only compares the difference between two equally long strings [18]. It can also find similarities if letters are inserted or removed in the compared strings.

Finally, with the Levenshtein string distance, one can adjust the tolerance threshold in order to balance between false positive and false negative results. By setting a highest allowed edit distance for two strings to be compared, the best match can be selected if many string comparisons have a smaller distance. If no comparisons has a distance lower than the threshold, no match is found [45].

3.5.1 Levenshtein distance

The Levenshtein distance is an edit distance over pairs of strings. If a pair of strings are compared, the Levenshtein distance gives a measurement of how many edit operations that

are needed to start with one of the strings and end up with the other. There are three different types of edit operations considered when talking about Levenshtein distance, and they can be described as rules of a Context free grammar. The rules are of the form of $A \rightarrow B$ where A and B are nonterminal symbols that map to a single terminal symbol each or to the empty symbol. If neither of A or B in the production are the empty symbol, the edit operation is called a substitution operation. If A is the empty symbol and B is not, the operation is going to be called an insert operation and if A is not empty but B is, the operation is going to be called a delete operation[45]. Certain series S of zero or more of those operations takes the start string and arrives at the end string, there are an infinite set of such series. The Series S has a score related to it that can be seen as a weighted sum of the operations in the series. The Levenshtein distance is then the least score from this set of series. A natural thing to do when setting the weight for edit operations is to give an insert and a delete operation the same weight. This makes the Levenshtien distance the same when going from a start string to an end string as when going back from the end string to the start string. There are several ways of calculating the Levenshtein distance and the performance of these algorithms are $\mathcal{O}(mn)$ where m and n are the lengths of the respective string [45].

3.5.2 Jaccard index and Jaccard distance

Jaccard index is a measurement of how similar two populations are [37]. The Jaccard index is described with the formula:

$$J = \frac{|A \cap B|}{|A \cup B|} \tag{3.3}$$

[24]. Here, A is the attributes present in only the first of the populations and B is the attributes present in the second but not the first. The expression $|A \cap B|$ is size of the intersection between A and B and is the number of attributes shared between the populations. The Jaccard distance is going to be defined as:

$$1 - J = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \tag{3.4}$$

This can be interpreted as the number of attributes in either A or B but not in both divided by the number of attributes in the union of A and B.

3.6 Spoon Tool

Spoon is a tool for code analysis and automatic refactoring of Java code. It uses the JDT library as its backbone but provides some abstraction and extra functionality to make code manipulation easier. Among other things, Spoon provides functionality for filtering elements and making queries, which can be used to find and filter among source code element such as class- and method declarations, expressions and invocations. Just as JDT, Spoon uses visitors to visit all nodes of a certain type in the AST. In Spoon, visitors are called processors, but the principles are the same and the processors are very similar to the visitors in the JDT library. For the visit methods, some granularity of the code element to be visited is chosen, it can for example be classes, methods, blocks or catch clauses that is to be visited. A finer grain generally makes the wanted information for analysis or transformation easier to access. Therefore the visitor will contain less code, but the trade off is that some information that is available in courser grain elements is "peeled off" in the finer grained ones and therefore not visible [35].

Some code refactorings that has been done in Spoon are, for example the insertion of null checks, insertion of try catch clauses and insertion of variable and method declarations. Some more advanced transformation that also has been done is pull-up-method and create super-class. To do a "create super-class" refactoring means that out of two or more classes,

extract a parent class. The parent class will contain the methods that its children have in common. The definition of a "pull-up-method" refactoring is that two classes with the same implementation of a method gets that method extracted to a common super-class. Both of these transformations have been proven to work well in Spoon through a competition that the crew of Spoon attended [34].

3.6.1 Meta model

The Spoon meta model have all required information to be able to derive, compile and execute Java programs. The structure is divided into three different parts, the structural part, the code part, and the reference part [27]. In the Structural part the program elements are defined, as shown in figure 3.4. This part contains interface, class, variable and method declarations and they all inherit from an element interface called CtElement where the Ct stands for compile time.

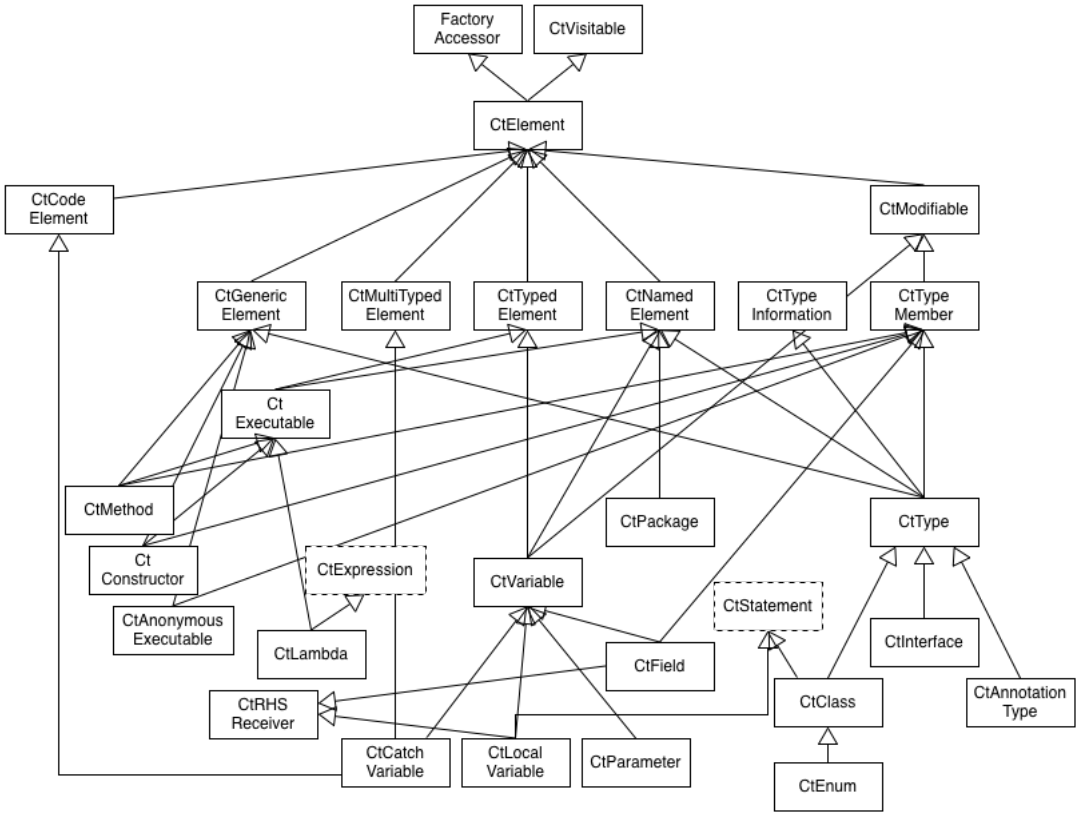


Figure 3.4: The structural elements in Spoon [27]

The Code part is the meta model for Java executable code and here statements and expressions can be found, see figure 3.5. Those two are the main code elements which most of the elements are inheriting from. In a block of code, top-level instructions can be used directly and these instructions are statements. Then CtExpressions can be used inside CtStatements [26]. Some code elements can inherit from more than just one other element like the CtInvocation, which are both a CtStatement and a CtExpression. The CtInvocation is used a lot throughout the implementation of the thesis application because invocations are the main way to access a library.

The last part of the meta model is the reference part, figure 3.6. The references state that referenced elements does not need to be reified into the meta model and can therefore belong

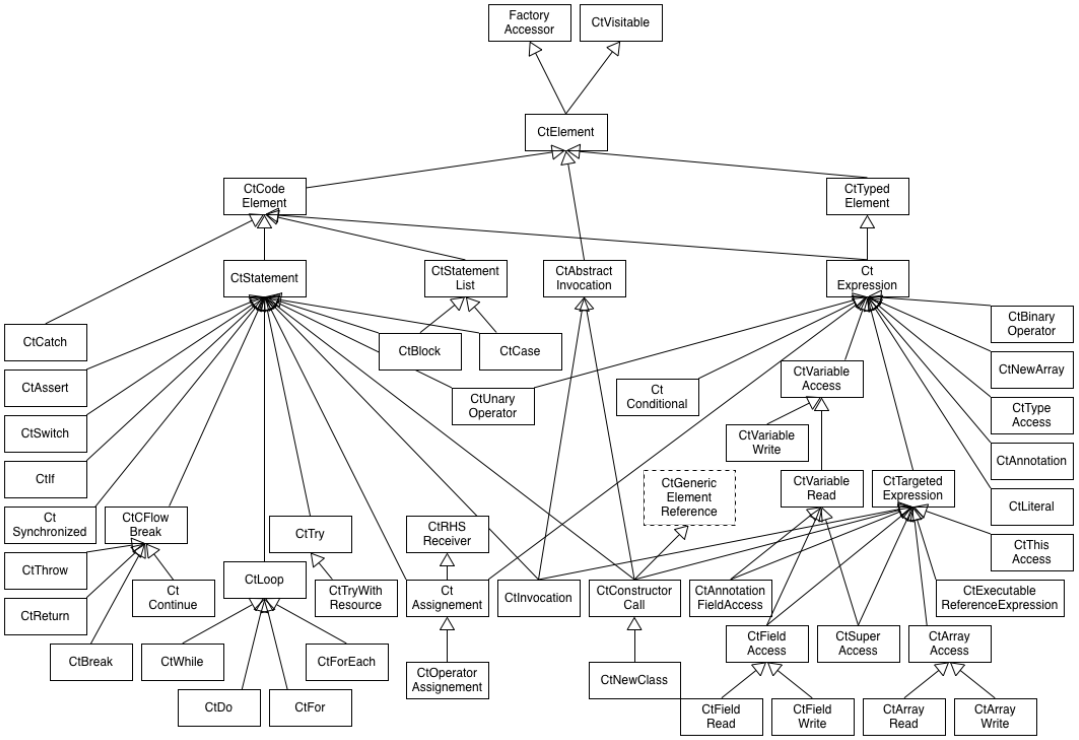


Figure 3.5: The code elements in Spoon [26]

to third party libraries. An example is that String is not bound to the compile time model of String.Java but instead to String. This means that the references between model elements and their reference elements are weak which make it more flexible to alter the program model. From this a low coupling is received but instead you have to chain the navigation, an example is $variable.getType().getDeclaration()$. All references have to be specified before the model is built because they all get resolved at build time, just like in the case of Eclipse JDT.

3.7 Development and maintenance cost of software

COCOMO is a model to do estimations of the effort it takes to develop software [8]. COCOMO defines three models for development effort estimation, a basic model, an intermediate model and a detailed model [23]. The basic model has the appearance of:

$$SDE = a_\alpha \cdot KLOC^{b_\alpha} \tag{3.5}$$

where a and b are constants and $KLOC$ is the number of source code lines in the thousands, that will be delivered. The effort SDE is measured in person work-months. The subscripts indicate that there are several options for the constants, depending on the developing organization. The basic model works best for quick estimations, but gives a fairly rough estimate [23]. a can be seen as a time scaling constant, while b says something about how the effort changes with the size of the project [10].

The Intermediate model has a similar look:

$$SDE = a_\alpha \prod w_i \cdot KLOC^{b_\alpha} \tag{3.6}$$

but uses a product of weights w_i to modify the estimate. The weights are collected from a table and corresponds to values for product, hardware, personnel and project attributes.

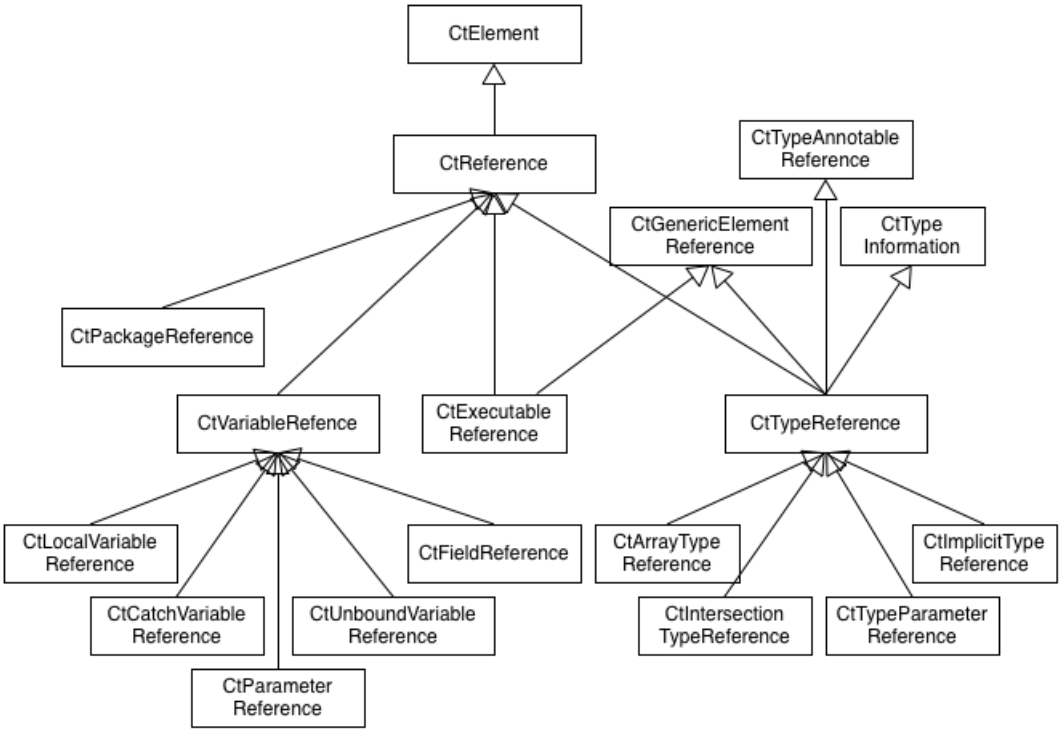


Figure 3.6: The references in Spoon [28]

Type	a	b	Description
Organic	2.4	1.05	Small products, few pre-established requirements.
Semidetached	3.0	1.12	Medium sized products.
Embedded	3.6	1.2	Large products in structured organizations. Requirements are well established.

Table 3.2: Constants for different modes [12, 8]

Weights for attributes not known can in the worst case be set to 1 in order to disregard their impact.

In the detailed version of the COCOMO effort estimation model, phase information are used for all of the attributes to give the model even more detail.

COCOMO also provides a way of estimating the cost of maintaining the software, this calculation uses the result from the development calculation as an input. The cost of maintenance is a large part of a softwares life cycle. There are basically two types of maintenance task done in software, the first is perfective maintenance which is to improve quality, performance and also the maintainability [31]. The second is to add new functionality to the software after its release. Around 60% of the resources for a product is used to do maintenance, which is a large part that a lot of companies do not think about [39, 23]. Models to calculate the maintenance cost have been proposed, and one method appearing in almost all papers studied in this thesis is the COCOMO model, as already talked about earlier in this section [39, 23, 31, 25]. As the COCOMO models for effort estimation, the model for maintenance cost also have three different models, the basic model, the intermediate model and the detailed model. The basic model calculates the basic maintenance cost with the help of

two parameters, annual changes traffic (ACT) and software development effort (SDE). The annual maintenance cost (AME), presented as person-months, can then be calculated as:

$$AME = ACT \cdot SDE \tag{3.7}$$

The basic model gives a rough estimation of the AME, to get a more accurately calculation for your project some weighting factors need to be added. The new model, intermediate model, can be calculated as:

$$AME = ACT \cdot SDE \cdot (\prod_{i=1}^{n} F_i) \tag{3.8}$$

where F_i is the factors presented in table 3.3. The model of COCOMO and its weights is derived from a research of 63 engineering projects done in 1981 to establish a maintenance cost prediction [8]. As seen in table 3.3 there are a lot of variables to take in consideration, some of them are easier to determine and others need qualified guesses or historical data. The third and last version, detailed version, takes each life cycle of the project in account and does estimates from these [23]. Therefore it is not an easy task to determine the cost of maintenance and it can differ a lot from project to project.

To be able to calculate any of these AME:s, the SDE and ACT variables are needed. SDE is the effort estimation calculated in 3.5 or equation 3.6. Annual changes traffic (ACT) is the measurement of how much source code is changed during a year. A change is when source code is added or modified. To be able to get a value of the ACT, historical data are needed to be able to estimate how much source code is going to be changed in the coming year. The ACT is calculated as:

$$ACT = \frac{KLOC_{added} + KLOC_{modified}}{KLOC_{total}} \tag{3.9}$$

Where $KLOC_{added}$ are all the added source code in terms of thousand lines of code and $KLOC_{modified}$ are the number of modified lines in terms of thousand lines of code. $KLOC_{total}$ are all lines of source code in the project in terms of thousand lines of code [1].

The SDE in equation 3.7 and 3.8 is simply the effort estimation from equation 3.5. The SDE can be calculated with some different constants a and b, as seen in table 3.2. With the constants the equation is proposed as:

$$AME = ACT \cdot a_\alpha \cdot KLOC^{b_\alpha} \tag{3.10}$$

where AME is the effort per year to maintain the software. In this case the effort is given as person-months. From equation 3.10 together with the constants in table 3.2, the model can be tweaked to suit different projects of different sizes and with different requirements.

3.7.1 Function points

Function points are a way of measuring the work effort of a software project. They present an alternative approach to counting the lines of code produced when determining the work effort[5]. The function point measurement is calculated by choosing a piece of software and summing scores of functions inside that piece of software.

Functions are divided into 5 different groups, each group has its own score attached to it. The five groups are: Internal logic file (ILF) which is logically related data that is managed from an external point. External interface file (EIF), which is logically grouped data outside the application that is accessed from within the application. External outputs (EO), which are processes where data derived from internal logic files are crossing the border out from the application. External inquiries (EQ), where unprocessed data from internal logic files passes the border out from the application. Finally External inputs (EI), are processes that input data to a internal logic file[36].

Weight	Very low	Low	Nominal	High	Very high	Extra high
Required software reliability	0.75	0.88	1.0	1.15	1.40	
Database size		0.94	1.0	1.08	1.16	
Complexity	0.70	0.85	1.0	1.15	1.30	1.65
Execution time			1.0	1.11	1.30	1.66
Main Storage			1.0	1.06	1.21	1.56
Volatility of Virtual machine		0.87	1.0	1.15	1.30	
Turnaround time		0.87	1.0	1.07	1.15	
Analysing capability	1.46	1.19	1.0	0.86	0.71	
Application experiance	1.29	1.13	1.0	0.91	0.82	
Programmer capability	1.42	1.17	1.0	0.86	0.70	
Virtual machine experience	1.21	1.10	1.0	0.90		
Programming language experience	1.14	1.07	1.0	0.95		
Usage of programming practices	1.24	1.10	1.0	0.91	0.82	
Usage of software tools	1.24	1.10	1.0	0.91	0.83	
Required development schedule	1.23	1.08	1.0	1.04	1.10	

Table 3.3: Weights for the COCOMO estimation of AME [8]

The formula:

$$UAF = \sum F_i \cdot W \tag{3.11}$$

describes how to calculate unadjusted function points from the functions in a software project. W is an individual weight that is assigned to each group. The weights are defined as 4, for the number of EI:s, 5 per EO, 4 per EQ and 10 for master files[4]. Master files consists of the total number of ILF and EIF. These unadjusted function points are then usually recalculated using additional project specific parameters. From the unadjusted function points, function points FP can be calculated by applying the formula in 3.12.

$$FP = 0.65 + (0.01 \cdot \sum w_i) \cdot UAF \tag{3.12}$$

The formula applies 14 weights w_i for project specific parameters in order to calculate the function points from the unadjusted function points [41]. The weights are all between the value 0% for no influence of the parameter and 5% for strong influence [41].

The maintenance cost estimate can then be calculated as in 3.13

$$AME = 0.054 \cdot FP^{1.353} \tag{3.13}$$

where FP are function points. The formula gives an estimation in work-weeks [2] of the maintenance cost.

4 Method

In the project for this thesis a large code base needs refactoring from using an old library to use a new library. The old library consist of code written manually by people at Ericsson in contrast to the new library which consists of automatically generated code. The new library have updated classes and methods of the functionality in the old library together with some added functionality that does not have equivalents in the old library. This added functionality is nothing that is needed to take in to consideration for the transformation. That the new library consist of at least the same functionality as the old library is crucial for the transformations to work, else there are no functions to represent the old ones. With that said, if there are just some functions that are missing, you are still able to do transformations and then manually add the missing functionality. An even better way is to add the functionality before the transformation and make a rule that says what shall be transformed into the newly written functionality, then you do not need to manually change the invocations in the code after you added the new functionality. More about the rules will follow later in this chapter.

Often the names of the classes and methods are almost the same which makes it easy to find equivalents in the new library but sometimes the names are different and then it is much harder to decide what should be used from the new library automatically. A thought was to look at the return values and arguments of methods, but they often differ too much to make any sense in the translations. This is due to many reasons. One reason is that primitive parameters passed to methods in the methods has changed sufficiently between the old and the new library. Another reason is that the non primitive parameters that are equivalent are represented by different types in the old and the new libraries.

To automatically alter source code a set of tools were needed. The tools would have to be able to modify the AST and to link the source code elements to their parents, types, arguments and more. When it is possible to link the pieces of the software together and find where methods and variables are derived from, it is possible to start to do automatic checks and changes. One prerequisite for the libraries and tools used in this project was that they should all be open source and that they could be used at a company such as Ericsson without any legal issues.

A small study were conducted about what options were available as open source or on the market that could be used in the thesis project. The first two found were Spoon[35] and Eclipse JDT [19], which also became the two used at the end. Other transformation tools are Stratego [43], which in fact is a whole language just for transformations and ASF+ SDF

meta-environment [11]. Both Spoon and Eclipse JDT are made with Java in mind and can only analyse and transform Java source code. This makes them bad tools for transforming Java source code to other programming languages, to do such transformations, Stratego and ASF+ SDF meta-environment are better tools. In this project the transformation is going to be from Java to Java which make Spoon and Eclipse JDT possible choices. Both of these tools have a rich set of functions for analysing and transforming Java source code together with the fact that they use Java to write the transformations. This make these tools ideal for this project, with lots of functionality to use without the need to learn another language just for the transformations.

The information from the study was the basis to choose Spoon and Eclipse JDT, but why use both? In the project where the transformations are going to be done there are around 400 000 lines of code so the thought were that a lot of transformations had to be done. Later in the project some statistics were produced which showed that this was the case, over 5000 invocations are made that should be changed.

To do transformations on all invocations automatically is not possible because the method signatures can have big differences from the old to the new library and then it is almost impossible to know which method to chose in the new library and which arguments to send to it. That is why some manual work is needed to be done for the transformations. Instead of a program that tries to make the correct guess for what the invocations should be translated to, a program that takes a set of rules for all the transformations that shall be performed is preferable. These rules can be generated semi-automatically by another program which will fill in suggestions that later can be changed by the one using the program. In the first part, the analysis part, Eclipse JDT was used. This is because it was found easy to start with and because it manipulates ASTs in a more direct than in Spoon, which uses the Eclipse JDT compiler to interact with the AST. To do transformations in the code, Spoon is used instead. This is mainly because the way of doing transformations in Spoon is easier than in Eclipse JDT and the lines of codes that needs to be written is a lot less in Spoon because a lot of the steps needed in Eclipse JDT are done automatically behind the scenes in Spoon.

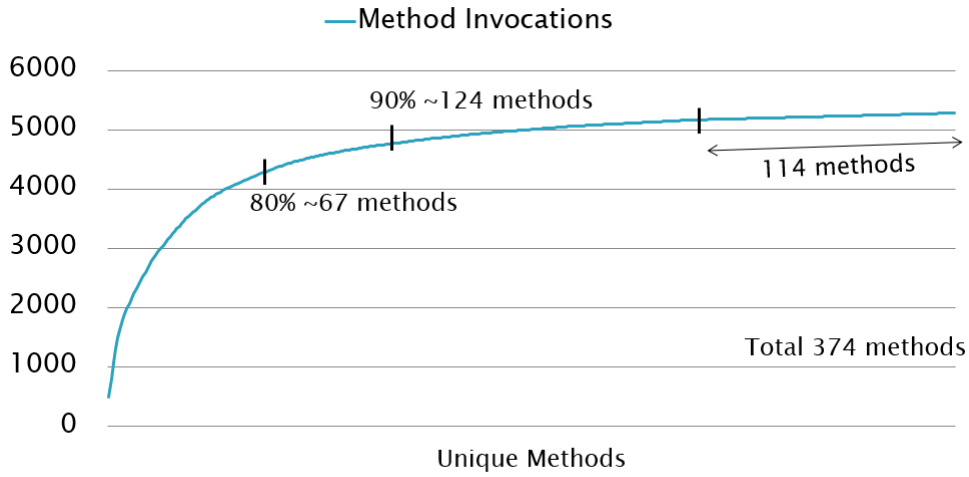


Figure 4.1: Distribution of method invocations.The y-axis show the number of invocation and the x-axis show the number of methods.

The work started with looking at the Eclipse JDT, which is a set of plug-ins for Eclipse with APIs to add functionality to use Java IDE. One of these plugins is called JDT Core, a plugin that has lot of measures for doing code analysis. One of the more useful features are the AST visitors. Listings 4.1, 4.2 and 4.3 show some different AST visitors and their usage. The visitor

pattern works by giving the programmer access to all nodes of a type in a data structure, as talked about in chapter 3. In this case, the data structure is a tree and the nodes visited are type declarations in listing 4.1, method declarations in listing 4.2 and method invocations in listing 4.3. It is relevant to look at visitors using visitors to handle their work. In this case, since a subtree of an AST is also an AST, a visitor can apply another visitor to a node from a subtree. This will make the second visitor to only be applied to the sub-tree. This gives the developer the power to only have to be concerned by the node types that are interesting in the AST.

```
ClassVisitor extends ASTVisitor{

    public boolean visit(TypeDeclaration node){

        ITypeBinding binding = node.resolveBinding();
        if(binding != null){
            classKey = binding.getName();
        }
        node.accept(new MethodVisitor());
        return true;
        }
}
```

Listing 4.1: Visitor visiting all type declarations in a specified AST

```
MethodVisitor extends ASTVisitor{
    Set<String> dataCollection;

        public MethodVisitor(){
            dataCollection=new HashSet<String>();
        }

        public boolean visit(MethodDeclaration node){

            //do some method statistics
            //for method invocations.
            //fill the dataCollection.
            return true;
        }
}
```

Listing 4.2: Visitor visiting all method declarations in a specified AST. In this case it is called from the ClassVisitor

```
MethodInvocationVisitor extends ASTVisitor{
   String InvocationInfo;

        public boolean visit(MethodInvocation node){

            //compute some information
            //about the invocation,
            //and the return it.
            return true;
        }

        public String returnInfo(){
            return InvocationInfo;
        }

}
```

Listing 4.3: Visitor visiting all method invocations in a specified AST

4.1 Finding equivalents

The very first thing done during this thesis was to study the code base manually. This was done by simply traversing parts of the code base and looking into the relations between different parts and by looking at the structure of the libraries. This was done for two reasons:

1. To give insight on an abstract level about the structure of the code base.
2. To provide information about what transformations to expect of the the automatic analysis tools.

Similarities between the two libraries in terms of structure and representative transformations of methods and types, between the libraries, were recorded on paper.

The next step taken was to try to automate the process of finding similarities between the libraries. Since most of the similarities in the libraries that are considered in this thesis are similarities in namings and structure it makes sense to primarily look at these when trying to match parts of the libraries together. The matching between the libraries had the purpose of giving an overlook as well as to provide some automatically generated input to the next phase of the thesis. The reason is that there was a need for manually written rules to describe how the source-code was to be changed. Much could potentially be won by automatically finding replacements for the parts of the old library used. Given that a part of the equivalences between the libraries where found, these similarities could be used as rules. Then, these similarities would not have to be given as manual rules.

Specifically classes and methods are the two important items that must be matched. In order to match classes, some characteristic of the class can be extracted and compared. The size of the class, the classes number of methods, nesting, complexity and method matching can be considered. The most obvious way of matching classes however, and the first attempt made in this thesis, was to match class-names. This was done with the Levenshtein metric as discussed earlier in section 3.

It was also considered to look at arguments to methods as well as their names. Two alternatives were possible. Either an argument distance is extracted from two candidate methods and combined with the string distance. Alternatively the string distance can be considered first. If there are several best matching candidates for a method, the method candidate with the smallest argument distance would be preferred. The method candidates for the equivalence could potentially have the same name.

To determine the argument similarity,or rather the distance, between two methods, a metric similar to the Jaccard distance described in section 3.5.2 was used. In this thesis, all occurrences of all types in the old method argument was counted as well as the ones in a method proposal from the new library. The argument distance between the old method and the new method proposal was then taken to be the sum of differences over all the types used. The difference of a type was calculated as the difference of the usage in the old argument list and in the new one. The difference between this method and the Jaccard distance is that several arguments of the same type are seen as separate. Therefore the difference of the old and the new argument are summed. This is because if one of the methods takes 1 integer as an argument and the other takes, say 8, then they should not be considered the same. Furthermore, the argument distance in this thesis is not normalized in the way the Jaccard distance is, by dividing by the number of arguments in $A \cup B$. This is so that a missing argument adds the same distance independent of how many of the other arguments that match. In the end, the thesis added up not using this distance.

Finally, the automatically generated rules that resulted from a mapping with a Levenshtein distance greater than zero were separated out from the set of rules with the distance zero. This was because the exact matches were deemed to be mappings to the correct method name albeit not necessarily with the correct signature. The inexact matches needed a bit

more of controlling so that a wrong method mapping was not used as a rule. This was done by manually checking inexact matches suggested by the analysis tool. For the cases where it was not obvious if a mapping was correct or not, the source code had to be investigated. In the cases where it still unclear whether two different methods were related, an expert at Ericsson was asked to confirm their equivalence.

4.1.1 Call graph

An experiment to find method groups in the code base that were unrelated to each other were devised. Here, unrelated methods will mean that two methods do not contain any invocations to the same method. Two unrelated method groups will mean that a group of methods does not contain any method such that it is related to any other method in the other group. The name transitive hull will be used for this relation. A tool for examining these relations and printing call-graphs was constructed using the Eclipse JDT library. The reason behind this transitive hull experiment was to see if there would be subsets of the transformations rules that could be applied on the whole code base and not leave any transformations of compilation units half finished. In that case, some set of methods from the code base would use a set of methods from the old library and no other set of methods from the code base would have an overlapping invocation set. However it was seen that no such subsets existed for the code base, which means that no set of rules less than the whole set could be applied on the entire code base and leave all test cases complete.

4.1.2 Retrieving Parameter Information

Analysis was also made upon the signature pairs of the invocations to be transformed. This since it was understood that the manual work needed in the source code to fix all argument conversions were quite massive. Therefore, analysis for grouping conversions of methods under specific signature conversions were tried. The goal was to group transformations with similar signature handling together, in order to write less code for converting the signatures during the transformation.This analysis was done by taking different method signatures for old library methods and matching them against all signatures of the corresponding new method. Several method conversions used the same signature conversion and by looking at the signatures, method transformations could be grouped together.

4.2 Design

This section will take a closer look at the design for the transformer tool that was implemented. Starting with the requirements for the tool, what is needed for the tool but also what the tool shall produce, and then the design decisions that were made. The design followed a set of requirements for the tool.

4.2.1 Requirements

The requirements for the tool have been developed together with the people at Ericsson. It covers both what functions the transformer would have and also limitations of what was possible to achieve. This was developed through discussions and small experiments in the beginning of the project. The requirements were:

- Be able to transform code from using one of two libraries with equivalent functionality.
- A one to one relation between classes and methods in the libraries was necessary in order to perform the transformation.

- Have a set of files as input. They shall contain the old and new libraries, a jar-file with dependencies, the files where the changes are to be applied and rules explaining the transformations.
- Shall be able to translate arguments from an old method invocation to the new one. This given that the new method has a permutation of the old arguments or that the new method has arguments that are themselves translations of the old methods arguments.
- The tool shall not do the changes directly on the source. Instead there shall be copies of the files being transformed. This make it possible to review the files before replacing the old ones.

4.2.2 Design decisions

In this section the decisions made about the design of the transformation tool will be shown. How the design decisions were taken will also be presented.

Spoon as transformation tool

Out of the two tools selected for the AST manipulation performed in this thesis, Spoon was preferred for doing the transformation part. It has the advantage of being more powerful and this is especially notable when changing source code.

Visitor design pattern

The transformation tool was designed to use the visitor design pattern to visit all classes in a specified project. The decision to use the visitor design pattern was a directly influenced by using Spoon. Spoon is built so that you have a processor that is called and this processor inherits a visitor. This visitor can be used for visiting classes or methods to name a few. The first time the processor is to be run, before all the visits are done, a setup is made to load dependencies and tell where to look for transformation rules or descriptions.

Structure of the tool

The initial decision of how to structure the tool was to have a main tool which took rules from a file that could be altered by the user. The main tool should ask the user to specify which folders or files that should be transformed together with a jar of class files that contains all the dependencies needed by the files to be transformed. The dependencies in this case are all the classes used by both the new and old library, even the files that shall not be transformed. This is needed because when transforming the files, bindings for all the elements are resolved and if they do not have dependencies the program will get an exception and crash.

In figure 4.2 an overview over how the program works and what dependencies each part of the tool has. The overview shows what was just talked about, that the transformer tool depends on both the analysis tool and the external library of Spoon. The analysis tool only have one dependency and that is to the external Eclipse JDT library. The tool also have an output to some rule file which later on is being read by the transformer tool.

4.3 Implementation

This chapter will describe the implementation of the transformation tool. As mentioned in the design chapter, 4.2.2, the transformation tool is taking help of another tool, the analysis tool. Both of these tools will get their own subsection because they are built with two different libraries and are separate programs.

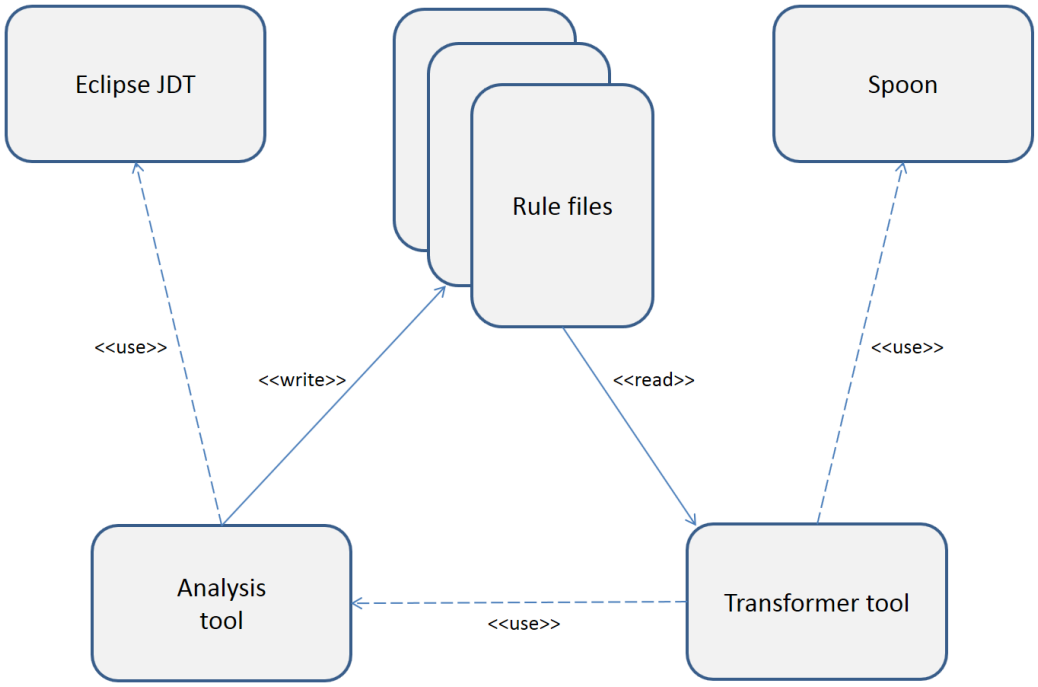


Figure 4.2: Overview of the transformer tool and its helper libraries

4.3.1 Analysis tool

The analysis tool depends on the external library called Eclipse JDT, especially the package Core which have all the transformation tools for the AST. Core includes *Core.dom* that supports examination of the AST structure all the way down to the statement level and *dom.rewrite* that supports rewriting of the AST to manipulate the structure, all the way down to the statement level [19]. As stated in chapter 4.2.2, the rewriting part is left to the Spoon library and therefore only the *Core.dom* is used from the Eclipse JDT.

The tool was built in a way where you specify the folders of files that you want to transform, the folders with the files containing the new methods and also the folders containing the old methods. The folders with new and old methods are needed to match old methods with their equivalent in the new library. The folders to transform are needed to be specified because the transformation is only to be performed for the parts of the code that have invocations to the library. Also, specifying the folders gives the ability to perform a transformation on a part of the source code. When all these files are added one last thing is needed, that is a jar with classes for all the dependencies the added files have. This is very important, without the dependency classes, Eclipse JDT will not be able to resolve bindings and most information used when examining the AST is gone. From all files added to the tool it can now use them to parse the source code into ASTs where each source file gets its own AST, the root of each AST is called a compilation unit. When making the compilation units, each unit will be put in a hashmap with its qualified name as key and the unit itself as the value. The qualified name is one of the two mainly used names for classes in the AST, the other one is simple name. An example of a qualified name is *Java.lang.String* and the corresponding simple name is *String*. The old and new compilation units are saved in different maps. The setup can be seen in figure 4.3

After the map of compilation units is created, the tool compares the old and new classes. The comparison is made by taking out the class names from the qualified name and do a string compare, if and only if the strings, as lower case, are the same they will be regarded

as equal. This is done because when comparing methods, only the ones in the equivalent classes should be compared. This make sense because comparing method in classes that are not equivalents would only lead to wrong methods matches. All the classes that have equivalents are put in a new hashmap with the old library class name as key and the new library class name as value. Having a map like this makes it possible to find the new class directly when an invocation is found and present as a key in the hashmap.

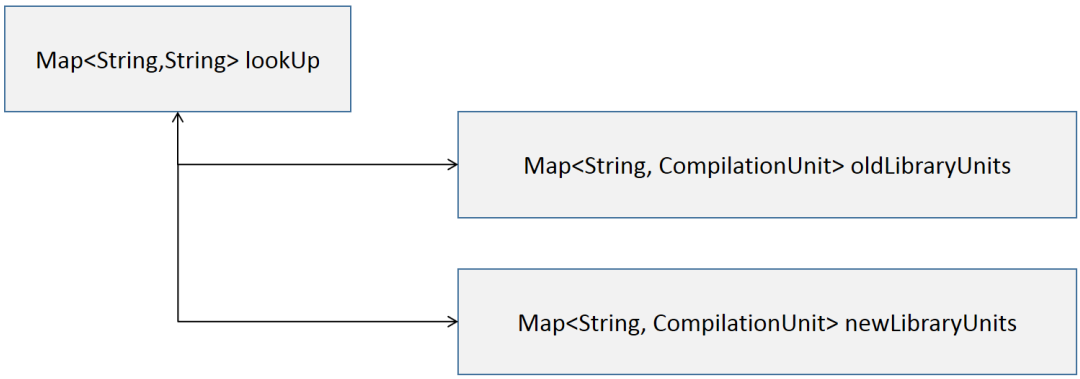


Figure 4.3: Mapping between the two libraries

Next step was to suggest equivalents for the methods. The first step was to iterate over all the files that need to change the invocations. When iterating through the files, a compilation unit was taken out for each file. Every compilation unit will accept a visitor which is going to look for method invocations. If any of the method invocations are to the old library they will be put into a list. A hashset is also present with unique methods from all the classes which all invocations will be put if they are not already present in the set. When all the methods in the code base residing in the old library have been found, all of the equivalent methods in the new library are found. For each class in the files that are going to get transformed all method invocations are taken out. Because the method invocations have bindings to the classes where the invoked methods are derived from, the qualified name for the class can be taken out and used to get all the methods in the new class. Each method in the new class are then compared against the old method with the Levenshtein distance on the names. The one method with the lowest score, is then chosen to be the equivalent, lower scores are better with 0 as the best score. By using this method the equivalents are matched correctly to 85.5%. This was measured by manually going over the methods equivalents to see if the equivalence suggested was correct. All the methods found is put into a map with the old method and class as key and the new method and class as value. Two different rule files will be made, one with old and new classes which is taken directly from this map and then another with rules based on signatures. The rule file with classes have a structure as seen in listing 4.4

```
class: Ospfv2MO ## Ospfv2Mo
class: DscpToPCPMapMO ## DscpToPCPMapMo
class: BrMMO ## BrMMo
class: QoSClassifierMO ## QoSClassifierMo
```

Listing 4.4: Example of the structure for the class rule file

The signatures are presented in the following ways. A list of possible signature conversions are written to a file. Together with the possible signature conversion, a number of how many times this conversion occurs in the code base. The numbers for the possible signature conversions are ordered so that the more frequent possible conversion appears at the top of the file. The file also contained the possible method invocations for a possible signature conversion, an example of the signature file can be seen in listing 4.5. It was now possible to check and select the wanted signature conversions between methods.

The signature calculation works by traversing the code base a second time. It looks at the method names and arguments of old invocations and uses the method mapping to look up the method bindings for the new methods. These class and method mappings are retrieved from earlier computations. In the compilation unit for the new method binding, all signatures for that method are extracted. Counters for the method conversions are incremented. When the print is finally done, first the counters and then the signatures followed by their methods are printed, see listing 4.5.

Since some of the method matches are inexact, a division of exact and inexact matches are made, listing 4.5. This is because of the heightened chance that the inexact matches are wrong, while pretty much all of the exact matches are correct, for some argument.

Sets of pairs are used to save signatures with arguments as well as return types. Wrapper classes contain lists of method pairs, one old and one new and their corresponding lists of return types. The length of the method pairs and the list of the return types are equally long.

```
--() -> () 307
--(int,) -> (int,) 86
--(String,) -> (long,) 37

() -> ()
  getEventType#FmAlarmTypeMO -> getEventType#FmAlarmTypeMo
  getLastRestoredBackup#BrmBackupLabelStoreMO -> getLastRestoredBackup#
      BrmBackupLabelStoreMo
  getSource#FmAlarmMO -> getSource#FmAlarmMo
  getAdminDistance#NextHopMO -> getAdminDistance#NextHopMo
  getDpdTime#Ikev2PolicyProfileMO -> getDpdTime#Ikev2PolicyProfileMo
  getAddress#NextHopMO -> getAddress#NextHopMo
  getIpsecProposal#IpsecProposalProfileMO -> getIpsecProposal#IpsecProposalProfileMo

(OperatingMode,) -> (EthPortOperatingModeEnum,)
  setAdmOperatingMode#EthernetPortMO -> setAdmOperatingMode#EthernetPortMo

(InterfaceIPv4BfdStaticRoutes,) -> (BfdStaticRoutesInterfaceTypeEnum,)
  setBfdStaticRoutes#InterfaceIPv4MO -> setBfdStaticRoutes#InterfaceIPv4Mo

---------- NOT DIRECT MATCH ----------

() -> ()
  getProgressReportPercentage#BrmBackupManagerMO -> getProgressReport#
      BrmBackupManagerMo
  getReportProgressProgressInfo#SwMMO -> getReportProgress#SwMMo
  getAvailStatus#EthernetPortMO -> getAvailabilityStatus#EthernetPortMo

(BaseMO,) -> (String,)
  setEncapsulationRef#InterfaceIPv4MO -> setEncapsulation#InterfaceIPv4Mo
  setBfdProfileRef#InterfaceIPv4MO -> setBfdProfile#InterfaceIPv4Mo

(ManagedElementMO,Object,) -> (String,)
  create#TransportMO -> createPtpMo#TransportMo
```

Listing 4.5: Example of the structure for the signature rule file

4.3.2 Transformation tool

The design uses several of Spoons custom filters to filter out nodes from the AST. Important nodes that are filtered are variable declarations, method invocations, and types from the libraries. An example to get all the invocations in a method block can be done by just making a filter like the one in listing 4.6.

```
List<CtInvocation> invocations =
    method.getElements(new TypeFilter<CtInvocation>(CtInvocation.class))
```

Listing 4.6: A way to filter out invocations in a method

When visiting the classes from the new library, which is done in the setup phase, method declarations are visited and stored away. The mapping between classes, methods and arguments are stored in a number of hash maps.

When designing the data structure for storing the equivalences, special care was taken so that the map could be filled with data without using a file with the rules. This was done so that it was possible to fill the maps from the visitor constructor. The reason behind this was to make the code testable earlier in the implementation. The design uses one hash map to store class equivalences and one to store the method equivalences. Both maps uses strings as identifiers. The method mapping is of the form shown in the table 4.7.

```
<"classname"+"methodname","newclassname"+"newmethodname">
```

Listing 4.7: Mapping

This makes it possible to map methods of a class to methods of many classes and to map types separately from its methods. Moreover it makes it so that methods in different classes with the same name can be mapped separately of each other.

A special SpoonLoader class was designed to give the option of how to visit the libraries. This since it is important to have the libraries and their bindings active when doing the transformation. The extra information given by the AST is only available when the bindings are active.

Depending on the signatures, different actions should be taken for the methods to be transformed, it was decided to use the signature conversion to determine what strategy would be used for the rule for a certain transformation as mentioned earlier in 3.3.3. Strategies are used to decide how to apply rules. In this case the strategies describe signature and parameter conversion. This in line with the suggested use of strategies in [43]. These large groups of signature conversion strategies are likely to have one single, or a few different ways of translating the arguments, depending on what method is transformed. for example Object to String or String to long type conversions need some extra logic which was be coded into a Java rule file.

For empty signature conversions, as well as for integers to longs, no action more than to give the old signature to the new invocation is needed. When the signature is transformed from Object to String, much more additional logic has to be applied. The Incoming *Java.lang.Object* can for example be of type Integer or String or another non primitive type. In this case $Integer.toString(object)$ is applied.

In listing 4.8 some examples of how Spoon filters are used, abstract filters can be used to construct user specific filters.

```
List<CtInvocation> invocations = method.getElements(new TypeFilter<CtInvocation>(
    CtInvocation.class));

List<CtField<?>> fields = classElement.getElements(new TypeFilter<CtField<?>>(
    CtField.class));

List<CtTypeReference> typeRefList = ctClass.getElements(new TypeFilter<
    CtTypeReference>(CtTypeReference.class));
```

Listing 4.8: Queries with filters

```
ctClass.getAllExecutables();
ctClass.getAllFields();
ctClass.getAllMethods();
```

Listing 4.9: Queries using getAll methods

As well as filters, Elements of the AST model can be queried with such methods as seen in listing 4.9. These returns all code elements of type executeable, all fields and all methods residing in a class respectively.

As suggested by Visser. E, pattern matching and term construction was decoupled from the transformation rules and their signature scopes [43]. This is done in order to be able to apply different signature conversions in different contexts.

Because of the double libraries and their incompatibility, duplication of methods are found in several places in the helper classes, the classes that contain functionality that are not tests but is not in the meta model for the libraries. Since these helpers were transformed, the duplicate functionality lead to duplicate methods. The way these were detected was by saving methods whose invocations or types had been transformed to a list. by comparing transformed and unchanged methods by signature and name, the duplicate methods were found. In this case it was desired to remove or mark the transformed duplicate method, so it could later be removed. This since the untransformed one is thought to function correctly with a certainty that is not guaranteed for the transformed one.

4.4 Economy evaluation

In this section some economical aspects of the work will be presented to demonstrate why the usage of the tool is beneficial.

4.4.1 Optimizing work effort

In order to answer the question of if it is possible or economical to do automated transformation for library replacement, the COCOMO model for effort estimation was used. In the theory chapter 3.7 a model for how one can use COCOMO to estimate effort is presented. Library replacement was seen as a developmental task and therefore the effort estimation for development of COCOMO could be applied. The number of method invocations to be changed in the code base was used as a rough approximation of the numbers of lines that had to be changed. The constants $a = 2.4$ and $b = 1.05$ was chosen, which in COCOMO is called the organic mode. Organic mode is the name for the mode used for small products with few pre-established requirements, as can be seen in table 3.2 from the theory chapter. The organic mode was used since the numbers of lines of invocations from the old library were relatively low. The COCOMO consists of three models as mentioned in the theory chapter 3.7, for this estimation the basic model was used since the values for the intermediate and the detailed ones require a lot of information which was not available for this project. The basic model will however make a rough estimation that is good enough to draw conclusions from.

4.4.2 Maintenance cost

The main idea of using the tool developed in this paper is to reduce the maintenance cost. The hypothesis is that if one library can be removed, the maintenance cost will be reduced. This can be shown by using the COCOMO model described in section 3.7 of the theory chapter. To calculate the COCOMO maintenance cost, the annual change traffic (ACT) is needed. This can be calculated by using equation 3.9. The equation have the variables $KLOC_{add}$, $KLOC_{delete}$ and $KLOC_{total}$. At Ericsson they use a team code collaboration tool called Gerrit, and from this tool all changes to the source code can be seen. By adding the changes made throughout the years a mean of 1743 added and deleted lines of code per year was found. By looking at the library it was found that the whole library is 11 309 lines of code.

A calculation on the new library can be made in the same manner to see how many person-months that are still needed to do maintenance. In this case where both the libraries have been used simultaneously, the interesting part is to see how much that can be saved by removing the old library. Because the new library is made automatically from a meta-model, the maintenance will also depend somewhat on the meta-model. To estimate the maintenance cost in some manner, it is needed to separate the maintenance cost for the old and the new library. In order to approximate how much the maintenance cost would change due to the

transformation, the following method was used. The number of unique methods from the new library used in the source code were counted. After a transformation was performed the methods were counted again. Since commit history was available, an approximation of the change in commits per year could be calculated. Meanwhile, the same type of statistics were available for the old library, making a similar approach possible for calculating the decrease in commits after the transformation.

4.4.3 Maintenance cost with function points

In order to give the reader the ability to access the validity of the result of the cost estimations, Function points were used as a second evaluation strategy. A single file was chosen for function point counting. Later this files effort estimation were used to extrapolate an approximation of the total maintenance cost of the old library. A web-based tool were used for the calculation of function points [15]. The numbers of Internal logical files (ILF) was determined to 2, these were 2 global variables independent of each other. 1 basic external output (EO) was found, other methods that communicated with the outside world used this method as a proxy. There were no external queries (EQ) and neither were there external interface files (EIF). on the other hand, 5 distinct input parameters to methods were concerned to be External Inputs (EI).The external inputs were counted once per distinct parameter rather than once every time they appeared.

5 Results

In this chapter, all the results from the different parts of the study conducted throughout the thesis will be displayed.

5.1 Finding equivalents

The analysis tools developed in this thesis shows the benefit of AST analysis , when analysing source code. The tools were able to investigate the statistics of method invocations, to reveal call hierarchies and to find equivalences between classes and methods.

The Results of this part of finding equivalents show that approximately 63 out of a total of 126 types could be matched exactly, that is to say with a Levensthein distance of 0. These types are guaranteed to be true positives. If one increases the threshold so that all types gets a match, the number of false positives are approximately 38/126. It is useful to point out that there is no magic or even fancy logic, for finding these equivalences, the reason that name matching works so well in this case is that the old and the new library are derived from the same meta model. If the namings of equivalent parts were more different between the two libraries, fewer equivalents could be matched automatically. Then more rules would need to be written by hand. Alternatively, in that case, the approach using rules could be scrapped altogether in benefit of some other approach.

The same high similarity, or even higher, is to be found for methods names, approximately 43 out of the total of 296 used methods were wrong, making approximately 85,5% of the methods in the exactly mapped types to be correctly mapped. The possibility to look at arguments, return types or arguments and return types together (signatures), when trying to find similar methods were investigated. However, when it was understood that there were several practical problems in combination with a very limited benefit of this comparison, the work was discontinued. There were two main problems, the first one was that equivalent methods in the old and the new library seldom took the same arguments and that they used many complex data types. The second problem was to scale different metrics to know which one should be of more importance. When the methods used their own complex data types, it was only possible to match these types with the same kind of statistical success that is mentioned above in this section. The other problem was how to use an argument similarity score together with the string metric score for the method names. Scaling the metrics, possibly with the use of genetic algorithms, to compare these different scores can probably be made into a

thesis of its own. The high name similarity, also makes it less useful to look at other metrics, such as arguments. It is doubtful that in this particular case, how much improvement can be made using argument matching. No improvements could be seen in the brief experiments with the string argument metric that were conducted in this study.

From the study of finding equivalents, except from the string matching results, it was also seen that the code base investigated uses about 5215 method invocations from 374 separate methods, declared in the old library. In the data from this study, it was shown that the distribution between the number of times different methods are invoked is strongly skewed, as seen figure 4.1. This means that a small amount of methods are responsible for a large part of the invocations while many of the methods had very few invocations. The most invoked method in code base from the old library has 506 invocations, while the 114 least used methods only has 5 or less invocations each. In the same spirit, 67 of the 374 methods were responsible for 80% of the total invocations while 124 of the 374 methods constituted 90% of the invocations. The results from the statistics of the method invocations are best shown in Figure 5.1. As seen, the distribution between number of invocations to different methods is heavily skewed.

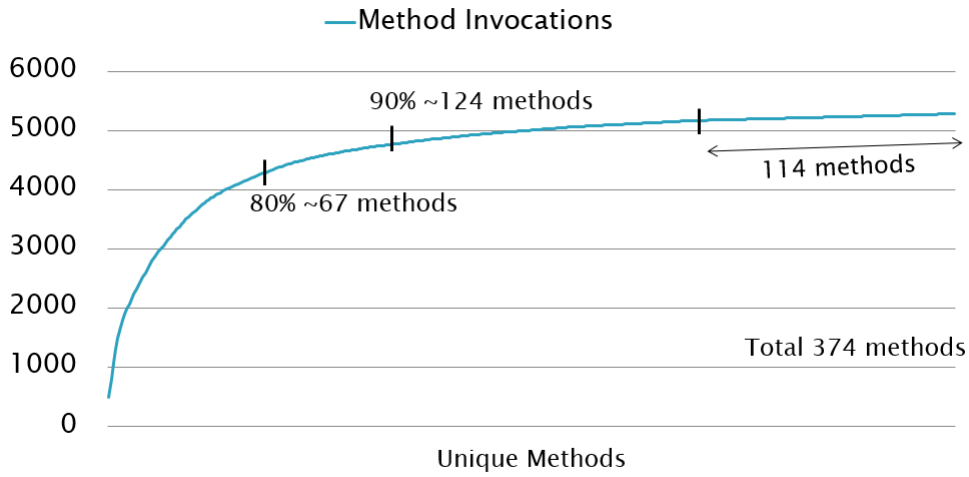


Figure 5.1: The skewed distribution of the method invocations

5.1.1 Prioritizing transformations

Since it might be less work to do some of the transformations by hand than to write rules for them, the information concerning frequency of different method invocations can be very useful. It allows for a selection of what parts of the old library to make transformation rules for. Especially if the aim only is to remove some percent of the total number of invocations to decrease the dependency to an old library, an educated selection of the method invocations to replace might considerably reduce the work.

This thesis shows to extract dependencies and call hierarchies using Eclipse JDT and the Spoon tool. In table 5.1 some of the relations that were extracted are shown. These small pieces of information can then be puzzled together into information about usage of code elements. Redundant code can be found by looking at code elements that are not depended upon and are not part of a test suite.

Eclipse JDT was used to extract the transitive hulls of the changes needed to replace the old library with the new one. It is shown that in this case there existed only one single transitive hull and therefore, the transformation could not be divided into smaller parts.

Super class of class
Interfaces of class
Type reference of methods
Method declarations invocations

Table 5.1: Relations extracted from the abstract syntax tree

During the analysis phase, it was found that some test cases used both the new and old libraries, with the result that the test cases became particularly awkward. These test cases contained duplicate functionality from the libraries in order to function properly. The libraries do not share any data types or classes except from the basic Java types, even though they contain the same functionality. The result is that these test cases using both libraries are filled with workarounds so that the pieces can fit together.

5.1.2 Retrieving Parameter Information

The results from the tries to group method conversions from the analysis are presented here. For every old method invocation to be replaced in the code base, a set of possible new signatures were extracted. These signatures consisted of all signatures from the methods with the smallest Levenshtein distance in name, from a class with an exact match in name, to the class where the method invoked resided. Every possible signature conversion got its own counter and every instance of the possible signature conversions were counted. Some few exceptions were made where the new argument was known beforehand, in these situations, only the correct signature conversion counter was incremented. Some of the signature conversions was inevitably wrong but the conversions that has a high number of possible method conversions attached to them were seen to have a big likelihood of being correct. In listing 5.1 some common signature transformations are shown with the number showing how many times that signature transformation is possible. As seen, String to long have 37 possible invocations. These can easily be fixed with a rule like $Long.parseLong(\text{"}arg\text{"})$. For signature transformation like Object to String it is harder to make a universal rule for the transformation. Then it is required to break it down to smaller sets and look at of what type the object really is. Often in the case of Object to String, the Object has been a String or an Integer which then can have different rules corresponding to which transformation shall be performed.

```
()        -> ():307
()        -> (String):302
(boolean) -> (boolean):56
(int)     -> (int):86
(int)     -> (long):50
(Object)  -> (String):628
(String)  -> (long):37
```

Listing 5.1: Common signature transformation sets

5.2 Design results

The first decision made was going to form how the overall design was going to look like. It was the decision of whether to use Eclipse JDT or the Spoon tool as a basis for the transformation tool. This decision was made after some testing with the both tools were conducted. The first one, Eclipse JDT, has a big tool set for doing analysis and transformations on source code. Using this library is however pretty cumbersome when it comes to code transformations. The Eclipse JDT tool was therefore rejected in favor of the Spoon tool in the implementation of the transformation tool. This is because Spoon is designed explicitly with the goal of code transformations as well as code analysis, albeit the anticipated Spoon transformations were smaller than the ones that is presented in this paper.

In listing 5.2 the name of all method declarations in a class are changed to "NewName" using Eclipse JDT. With Eclipse JDT, all the work with getting the old method and writing to a file have to be done manually, in Spoon this is all made in the background. In listing 5.3 it is shown how the same change of the method declarations names is done in Spoon. As seen it is lot less code and overall easier to do the same transformation in Spoon compared with Eclipse JDT. Spoon also provide ways of accessing the AST and filtering different node types from the AST.

```
ASTRewrite rewriter = ASTRewrite.create(unit.getAST());

for(MethodDeclaration method : methodlist){
    MethodDeclaration methodAccess = unit.getAST().newMethodDeclaration();
    methodAccess.setName(unit.getAST().newSimpleName("NewName"));
    rewriter.replace(method, methodAccess, null);
}

TextEdit edits = rewriter.rewriteAST(document, null);
edits.apply(document);
FileUtils.writeStringToFile(file, document.get());
```

Listing 5.2: Changing method names using Eclipse JDT

```
for (CtMethod method : methodlist){
    method.setSimpleName("NewName");
}
```

Listing 5.3: Changing method names using Spoon

In order to run a transformation, the tool for finding equivalents described in section 4.1, shall be run to create a file with rules. Before the tool starts the transformations it is possible to alter the rules made but also to add more complex rules. A decision about how to structure the rules was made. The rules were partly written in a text file and partly in a Java file. The text file contained rules about simple rules like how to transform method or what signatures should be matched. Complex rules like how the new code would look like in detail were handled in the Java rule file. The pros with having all rules in a text file was that all rules could have been made in a single file and that rules could probably been made easier than in a Java file. The problem with making the rules in a single file is that the tool need some sort of own interpreter where the rules will get translated to the right rules in Java and Spoon. The solution would also have been more locked to specific changes because in order to make the rules as generic as possible, all the rules have to be interpreted into Spoon commands. From this the decision was to use a separate Java file that uses the Spoon library directly when making the rules. The downside with this approach is that the user need to know a little bit about Spoon, this problem can be solved to some extent with having examples to look at.

When the user has specified all the rules, the transformer can execute the transformations with the help of Spoon as said before. The new files will be written to a new folder and then the users will by themselves copy the files and overwrite the old ones in the source file location. This was a conscious choice, because by doing this, instead of overwriting all the old files you can first look at the files and also compare them with a comparing tool like Meld before overwriting. Comparing files is out of the scope for this thesis, that is why a third party program was used to do this. The transformation tool does not care about whether the the result of the transformation is correct or not. Compile errors could be one way of checking this but even if no errors are present, some transformations could have been wrong anyway. Another thing to take notice of is that it is not necessary to perform all transformations using the tool. Some of them may be easier and better to do manually. One such case could be when you have to write a rule for a transformation that is only used once or twice, the rule will probably be longer and harder to make then just change it manually. The best way of knowing that the transformations have been a success is to have a good test base and see that all the tests still pass after the transformation has been performed.

5.3 Implementation results

Several inconsistencies between the libraries were found during the library transformation. Sometimes functionality had to be moved so that it could be used after the transformation was done. The effort to transform the code base to only use the new library led to refactoring of the old library to make it more coherent with the new one. At other times, variables were missing from the new library. In cases where variables declared *staticfinal* were used, the usual solution was to extract these variables from the old library and place them so that they could be reached after the transformation.

Classes containing duplicated functionality for the old and new libraries was found in section 4.3. When transformed, one of the method copies had to be removed. The duplication stemmed from developers trying to make the transition to the new library more straightforward.

Classes with the same name but residing in different packages did also occur in the code base. Because this was not anticipated, the design of the transformation tool had a hard time cooping with these duplicate names. Because methods were saved with their name and class name in the tool, duplicates made the program choose one of the classes at random.

In the beginning of the work, It was thought that method invocations made up the greatest part of the transformation. Testings with the automatic transformation tool shows that type references are just as frequent as method invocations, or even more frequent. Type references occur in definitions, assignments, types to method invocations, but also in expressions and as parameters to methods. Some enumerations also had to be taken into account when doing the transformation, even though they were less frequent than the invocations.

5.4 Economy evaluation

5.4.1 Optimizing work effort

The number of invocations to the old library were at the point of running the analysis tool 4803. After running the transformation in its current state and then the analysis tool again, the number of invocations left were about 170. 170 invocations compares to about 4% of the total invocations. There are about 910 lines of code representing rules for transformations. The average length of a rule is about 3 lines of code. Therefore it is estimated that it is beneficial to keep writing rules until the number of invocations that a rule will fix is about 3. This is assuming that an invocation takes up about 1 line of code. By looking into the skewed graph, graph 4.1, the fraction of methods that are called less or equal to 3 times covers about 6% of all the invocations.

So the optimal number of written rules covers 94% of the invocations and 96% of the invocations are covered with 910 lines of rules. The calculated estimation of the optimal lines of rules are given in equation 5.1.

$$(910/0.96) \cdot 0.94 = 891 \tag{5.1}$$

This is the optima for the numbers of rules to write for a project of the size 4803 lines, or in this case 4803 invocations. By seeing the replacement of those invocations as a development cost, an estimation of the cost of doing the replacement can be calculated. Here, it is assumed that one invocation corresponds to one line of code. By inserting 4803 as KLOC, that will say 4.803, into the equation for effort estimation, equation 3.5, the value is obtained as in equation 5.2.

$$SDE = 2.4 \cdot 4.803^{1.05} \approx 12.5 \tag{5.2}$$

This shows that the estimated replacement cost is 12.5 person-months. The constants used in the estimation are for the COCOMO organic mode. The total amount of lines of code

needed to be written to complete the transformation in the estimated optimal way are 891 for the automated part plus 6% of 4803 that has to be done manually. Equation 5.3 shows the calculation.

$$KLOC = \frac{891 + 0.06 \cdot 4803}{1000} \approx 1.179 \tag{5.3}$$

$$SDE = 2.4 \cdot 1.179^{1.05} \approx 2.9 \tag{5.4}$$

Inserting the result from 5.3 into 5.4 . The assumptions made here are that rules have an average length, that the correct COCOMO mode is used for the constants in the formula and that an invocation takes up one line of code. The constants were retrieved from table 3.2 in the theory chapter.

5.4.2 Savings in maintenance cost

The methodology from chapter 4 was used to calculate an estimation of saved person-months per year. The values for the size of the old library and the number of lines commited during a year were inserted into equation 3.9 in order to calculate the annual change rate. The resulting calculation can be seen in equation 5.5.

$$ACT = \frac{1.743}{11.309} \approx 0.154125 \tag{5.5}$$

This result can be used in equation 5.6. The constants for semidetached mode from table 3.2 in the theory chapter were used.

$$AME = 0.154125 \cdot 3.0(11.309)^{1.12} \approx 6.9957 \tag{5.6}$$

The result here represent the person-months per year needed to maintain the software. From this it is seen that by removing the library a lot of work can be saved, almost 7 person-months each year.

The maintenance estimation was redone with Function Points and the TINY TOOLS function point calculator was used. A great thing about the web-based tool used in the calculation is that it produces function points rather than raw function counts (UAF), this makes for a better approximation of the maintenance effort. For a single file of the old library, external inputs were calculated to 5, external outputs to 1 external interface files and external queries to 0 and internal logical files to 2. These numbers are not exact and depending on how the function points are defined, some of the five counters vary. The 14 adjusting factors were generally scored low with the exceptions of the data communication score and the performance critical score. The resulting function point count score were 32.76. By using the formula $E_w = 0.054 \cdot FP^{1.353}$ proposed by Yunsik Ahn et al. [2], the effort for maintaining this file was estimated to about 6.06 person-weeks per year. Since this particular file were 541 lines of code long and the whole library at the point of measurement were about 11309 line long, the total effort were extrapolated to $6.06 \cdot 11309/541 = 126.73$ work-weeks per year. This is about 29.2 person-months of effort per year.

In order see if the maintenance cost was really reduced by replacing the old with the new library, the commit history was looked through. Over a one year span 68 respective 20 commits with Java code were made to the old and the new library. 375 methods from the old library, and 850 methods from the new library were used in the source code. After a transformation, the number of methods used in the new library were 888.

By this, it is shown that a reduction of the maintenance cost can be achieved by removing the usages of the old library in the source code. From the study of the economy aspects of the work in section 4.4 and from the calculation above, it is shown that automation of library replacing can decrease the work of changing between libraries significantly and also that there is an overall decrease in maintenance cost when the old library is removed.

The first of the research questions in section 1.3 was, how much of a transformation between two libraries is it possible or economical to automate? The analysis tool were used to find the answer to this question. The analysis tool was first run when no invocations had been replaced and after a transformation when most of the invocations were changed. In the first run the invocations were 4803 and after the replacements it was down to 170, that will say 4% of the total invocations. To calculate the benefit of the transformation, the written rules for doing these changes were counted and used to get the optimal lines of rules to write. The optimal amount of rules ended up to be 891 lines of source code for the 4803 invocations.

By using the COCOMO model the development effort for a project can be calculated [8]. In the study made in this paper it was shown that for all the 4803 invocations, 12.5 person-months were needed to change all the invocations manually. If the tool for changing the invocations automatically is used instead, with the optimal 891 lines of rules and manually change the remaining 6% of the invocations, the resulting number of person-months goes down to 2.9.

The second research question in section 1.3, how a partly automated tool for code transformation can decrease maintenance cost, is answered here. Part of the COCOMO model can be used in order to calculate the maintenance cost of a software project. In the case of this project the library is 11 309 lines of source code and by looking at change history it was seen that around 1743 lines were deleted and added each year. From this, the annual change traffic (ACT) could be calculated and used to get an effort estimation of the maintenance for the library. The calculation showed that almost 7 person-months each year are needed to maintain the library. An alternative calculation using function points suggests that the maintenance cost is about 29.2 person-months per year.

6 Discussion

In this section there will be discussions about the method, results and also about the work in a wider context. The discussion will discuss, explaining and clarifying the results together with taking a critical viewpoint of the work in this thesis.

6.1 Method

In order to make the thesis more manageable, many improvements to try to decrease the workload were thought up. Some of the important items that could or did decrease the workload were automatic rule extraction, method-use frequency extraction, standardized rules for common signature conversions, skipping of transformation for code elements not used in the test cases and parallel update of helper classes to the test cases. The helper library with its helper classes contains functionality that does not fit in the test cases. The test suite depend on the helpers, the helpers depend among themselves and both helpers and test cases depend on the old library. Therefore, it might have been a good idea to replicate the helpers and do the transformation on the replicas separately and then make rules for the replacement of helpers in the test cases. The advantage would have been that the work might have been divided into smaller parts and sub goals could have been set up.

On several places in the code, the work of transformation had already begun. Parts of the old library had been marked deprecated, packages had gotten counterparts equipped to use the new library and test code used both libraries in parallel. Because the libraries are inherently incompatible except from when they use basic Java types, the intermingled usage of both the libraries in a single test case can lead to uneasy or superfluous code solutions. Some test cases had been fully translated to use the new library. The parts in the tests that have been left using the old library were often the parts that required most effort to transform.

There are a few things that should be considered when building a transformer tool for a library replacement. Firstly the AST tool functionality should be well known by the developers beforehand. In this study, Eclipse JDT functionality was learnt as the analysis proceeded and the AST structure was understood with time. Because Spoon is built on Eclipse JDTs compiler the work with Eclipse JDT made the usage of Spoon easier, errors in Spoon was then easier to understand and fix.

6.1.1 Shortcomings

If the problem and solution are well known, that fact will provide the means of a more insightful implementation. It is also necessary to think through the design of the tools to be built, and the scope of their usage. In this thesis several unexpected problems occurred during the progress of the work. Since the scope of the task was not understood fully, much functionality had to be added onto the analysis and transformation tools along the way. Redesigning the software on the fly before new features were implemented, was tried, but it would have been easier if the tools built had been designed knowing the whole scope of the problem.

There might be better options for how to transform the code in the desired way. For example, an Eclipse plug-in for making the transformation might result in more reusability and ease of use. A plug-in that asks a user for inputs on a certain kind of problem and then resolves all problems of the same kind the same way, would be more user friendly. Also, even though it is possible to write new rule-files for new transformations, a plug-in that asks for user inputs would have a higher reusability because no code, or rules in this case, would have to be rewritten between transformations. An Eclipse tool that asks for the users advice can also easily be tested by letting a group of people test it. The rule based transformation approach however, is a bit awkward to teach to people. Therefore, the replacement cost evaluation were performed on a theoretical level rather than on a practical level. The estimation of the effort saved is among others based on the assumptions that writing one line of code with rules is exactly as time consuming as writing a line of code in the software. Another assumption that was made about the effort, was that the tool is reusable. That is to say that no other changes than those to the rule files has to be done when starting on a new transformation.

There are several things to be said about the evaluation methods. Neither COCOMO nor evaluation with function points are that accurate [17]. The reason that they were used in favor of a genetic algorithm is that they are easier to use and no overhead for implementation is needed. Function points are usually used for estimating how much effort must be put into a project by estimating the amount of functionality that the software provides to a stakeholder, the reason why the estimation with function points were so course was that the parameters were set based on guesses rather than expert opinions and that there ambiguities in the function count. Sources of error when the COCOMO model was used stemmed from the fact that that invocations were used as a measurement for lines of code, noise and model errors. The COCOMO model in contrast to function point count can be language specific. This is because the constants used are extracted from software projects written in specific languages. Function points on the other hand are effort depending on functionality provided to a stakeholder, these should in theory be more abstract and independent of program languages.

6.1.2 Replicability

Spoon can be used in some different ways, with Maven, Gradle or directly from the command line. In this thesis a bash-script was built that executed Spoon from the command line. When executing Spoon from the command line only one dependency folder or jar-file can be used. For this project this was not enough, so a workaround was made that loaded more dependencies into Spoon. When a solution like this is needed to make a tool work, it has bad consequences for the replicability. This because it is not intuitive and a lot of knowledge about that tool is needed.

Another threat to the replicability is the results of the work effort and maintenance cost. These two results are calculated based on a lot of experience. If there is no experience, the calculations have to be based on educated guesses. That is since the calculations depends heavily on project parameters and the different parameters that can be chosen. For both COCOMO and function points a set of parameters can, or need, to be chosen. These parameters

corresponds to everything from estimating the programmers capability to deciding how big an impact different software metrics have, like how important performance is for the project.

6.1.3 Reliability

If the study is repeated, there are some things to consider. Firstly, in order to map functionality from an old and a new library, some kind of similarity is needed. In this thesis, it was possible to map the functionality by name. This does not need to be the case but it must be possible to extract the similarity between the libraries in order map the functionality. In order to extract the mapping automatically, the similarities must be extracted from the code base and an analysis tool must be able to find these similarities. Both in the analysis part and in the transformation part, a one to one relationship between the libraries are assumed. This means that classes and methods that exist in one of the libraries usually have equivalents in the other. In this thesis it is small pieces of functionality from libraries that are equivalent, like method invocations or type references. This made the rules relatively simple to write. The evaluation used COCOMO to estimate the cost of replacing the library manually and automatically and both COCOMO and function points to estimate the maintenance cost. Both methods are simple and easy to use but they are also known to vary a bit in precision[17]. An alternative method like a genetic approach can potentially make a more accurate estimation. An implementation of a genetic development or maintenance estimation is however far outside the scope of this thesis. The maintenance estimations showed very different results, the reasons for this can be that the function points method does not take historical data into account like commit history. Since the maintenance was calculated on a library, it might be the case that the cost deviates from the what one would expect of a piece of software. In this case, the COCOMO estimation that makes use of commit history might be the more accurate of the estimations. In order to estimate how much maintenance is saved replacing the libraries, an investigation into the commit history was made. Initially it was thought that the removal of a library from a code base would save exactly that amount of maintenance. As pointed out by several persons however, the maintenance for the new library might go up considerably when its use increases. In chapter 5 the values of the investigation are shown. No formal estimation of the increase of the maintenance from the new library is made, but the numbers seem to point to a very low increase.

6.1.4 Validity

The Levensthein metric for string comparison is widely used and recognized as a good string metric algorithm. COCOMO estimation for development effort and maintenance cost and function point metrics are widely used. Function points are usually also used for development effort but in this thesis it has been used for maintenance cost using a formula presented in [2]. An improvement in the development effort using the COCOMO model could have been to not only look at method invocations but also at other usages of the library. A more thorough investigation of the source code, possibly using an automated tool could have given a more accurate picture of how many lines of code that was going to be replaced. Since method invocations are only one part of the development effort, it is likely that the effort estimation would have risen if a more thorough investigation would have been made.

6.1.5 Source Criticism

The articles dealing with similar types of transformations are a bit sparse. Therefore, this paper tries to find and identify relevant articles about refactoring to fill some of the gaps.

Due to the sparsity, the majority of the articles about refactoring and code transformation are either about code refactoring on a small scale or model refactoring.

The articles about the the Spoon AST tool and its uses are not written of independent researchers, but rather of people that are involved in the development of Spoon. The article on Stratego refers to several papers written by the same person.

Generally, the sources used are articles published in places like IEEE digital library and The ACM digital library. The exception was when referring to Eclipse JDT or Spoon literature. These were taken from their respective homepages. The web tool TINY TOOLS was used to calculate function points out of a function count. The tool seems to be a student project at the university of Michigan. The articles describing string metrics, function points and COCOMO are generally from the eighties while the papers describing refactoring generally stems from the early 2000s. In the Levenshtein distance case the age of the source does not matter since the algorithm has not changed. The COCOMO and newer COCOMO II models share a lot of traits, and therefore the older COCOMO articles are still valid. The references [30], [44] and [3] are books used as student literature at Linköpings university.

6.2 Results

One important result that emerged in the pre-study was the power of analysing the AST. Statistics over method usage, method mappings, call hierarchies and possible argument mappings were extracted. It was possible to tailor a search-query to answer pretty much any specific question about the software under analysis. It was found from experience, that this provided information for the transformation, gave insight in the code structure and gave the opportunity to formulate better questions about the code. Tests similar to the transitive hull test from chapter 4 can also be performed and give information about the structure of a code base. The results significance lies in that it shows that it is meaningful to analyse software automatically before doing big changes even if the change in itself is done manually. The level of insight gained by automatic analysis can be significant.

It has been shown before that tools using the AST can be used to find equivalences when it comes to methods that changes the program state in the same way [34]. An example of this is when two methods in two different classes contains the same functionality. Then a superclass can be made and the methods can be pulled up to this class and merged. A reordering and normalizing of elements in a tree gives a broader set of semantically equivalents that can be found. In this thesis however, the direction of the analysis has been towards naming similarities between elements such as types and methods, rather than finding methods with the same behaviour. This because of the nature of the project code and the library replacement task. Partly, namings are very similar and partly because the atoms of the source-code are different even though they achieve the same things. The analysis part of the thesis clearly shows that equivalents parts of two libraries can be found in an satisfactory way, using the Levenshtein string metric. This is given that the namings of equivalent parts of the old and the new library are similar enough. In the case where the a library replacement is required and the old and new library have equivalent parts that are not named similar, another approach has to be taken. An integrated library replacement tool where the user defines equivalents one by one could be an alternative approach.

In the results it was seen that the distribution between method invocations was heavily skewed. It was unanticipated when first seen, but at second thought, it is probably a common for several tokens in a larger software project, to have a skewed distribution. This skewed distribution makes it possible to select what method translations to write rules for. The more frequent the invocation is, the more beneficial should it be to write a rule for handling the translation.

The automatically generated rules for class and method conversions in this thesis were correct in a majority of the cases. This can be seen in the statistics for the correct guesses for equivalents, that can be found earlier in chapter 5. In the cases where the automatic guesses from the analysis tool are wrong, manual corrections had to be made.

In the case studied, the methods and classes were generally named similarly. If the arguments are empty or are of the same primitive type, there is probably no special rule needed. Neither is a special rule required if the arguments converted from are both of the type String or a wrapper class like Integer. For argument conversions like $int \rightarrow String$ on the other hand, some small rule for converting the number into a String is required. In this study, the case where the arguments represent the same object but has different classes is present. This means that both objects have equivalent functionality but the classes are not the same, as can be seen in listing 6.1, where the new and old library have the same method but are contained in two different classes. Generally, not much extra logic is needed here either. What takes much extra logic and specialization though, is when the arguments in the old and new method declaration represents different objects. Then the relation between the old and new argument has to be expressed.

```
getLocalAddress#IpsecTunnelMO -> getLocalAddress#IpsecTunnelMo
```

Listing 6.1: Example of the same object representation but different classes

The method invocation rules were grouped together under signature conversions. These groups were used as strategies for how apply the rules. In the Java rule file, every signature conversion group applied its own logic for how the transformation was to be performed. Some of the groups had to have different mechanisms for different specific method conversions but generally only one or a few approaches were needed for a signature conversion group. The reason why using strategies worked so well is due to the regularities in the structure of the source code. For example large groups of method conversions has the following signature conversions 5.1. In the case where the conversion $Object \rightarrow String$, the usual action is just to cast the argument to the type $String$.

It was found during the course of the thesis that the two libraries differed more than expected in terms of content in the old library that did not exist in the new one. It was known on beforehand that the new library had functionality that was not present in the old one. That the opposite was true to some extent posed a problem in the transformation. This even more so since the transitive hull test from4 had shown that it was hard to reduce the problem into smaller parts. All functionality from the old library was needed at once for the transformation to be achieved. Therefore, as issues with inconsistencies were found, they were reported so that they could be resolved.

Here an important point comes up. A piece of software that is generated from an UML model or other meta model can cut maintenance costs, but only if it can be used. A possibility is that it was assumed that the use of the new library in the software project would emerge naturally over time. If so, it is an open question how large the time frame was imagined to be for such an event to occur.

The savings in work-months were found to be high, when they were evaluated with the COCOMO model. Because it is an important result, it is relevant to discuss the shortcomings and assumptions made during the evaluation. The values chosen for the COCOMO evaluation are of course a source of errors. This is natural since it is an estimation model, but the choice of parameters can greatly impact the accuracy. The organic mode of the COCOMO was chosen and that is probably a good choice for the replacement of the library which is considered a relatively small development task. It might also work well with Ericsson's team organisation with small agile teams. However no multipliers from the COCOMO II model were chosen, making the estimation a bit courser. It was felt that information about the many parameters that COCOMO II handles was insufficient. Therefore, since the multiplier of 1.0 is the standard multiplier for all those parameters, they are totally disregarded in this evaluation.

The evaluation with the COCOMO model assumes, among other things, that the cost of replacing a line of code from the old library is equal to the cost of writing a new line of code. The simpler of the transformations performed by the automatic tool have been shown before.

Examples of simple rules are given in listing 6.2 while some examples of more complicated rules can be found in listing 6.3.

```
() -> ()
  getEventType#FmAlarmTypeMO -> getEventType#FmAlarmTypeMo
  getLastRestoredBackup#BrmBackupLabelStoreMO ->
  getLastRestoredBackup#BrmBackupLabelStoreMo
  getSource#FmAlarmMO -> getSource#FmAlarmMo
  getAdminDistance#NextHopMO -> getAdminDistance#NextHopMo
  getDpdTime#Ikev2PolicyProfileMO -> getDpdTime#Ikev2PolicyProfileMo
  getAddress#NextHopMO -> getAddress#NextHopMo
  getIpsecProposal#IpsecProposalProfileMO ->
  getIpsecProposal#IpsecProposalProfileMo
```

Listing 6.2: Example of some simpler rules

```
() -> (AbstractMo[],)
    showMacTable#BridgeMO -> bridgeMo.performAction("show␣mac-table").getResult()

(String,) -> (long,)
    showArpCache#RouterMO -> routerMo.performAction("show␣arp-cache␣" + arg).
        getResult()


() -> (String,)
    show#ConfigManager -> performAction#TnConfigHandler (arg1.performAction("show" +
         "␣" + arg2).getResult())

() -> ()
    getSendTimeoutMillis#ConfigManage -> getOperationTimeout#TnConfigHandler
```

Listing 6.3: Example of some simpler rules

Worth mentioning is that the getters and setters for timeouts are in seconds, and that the tool therefore has to do conversions from milliseconds to seconds and vice versa. The transformations seen in listings 6.2 and 6.3 are relatively simple, and that will affect the time saved. More complex transformations will require more transformation rule logic. In the defense of the calculation however, it shall be said that the conversions are not mere copy paste operations, since names for instances of objects in the code base are not always following naming conventions. There are other examples where the instance names differ from file to file. In the case where the transformed code uses an instance of an object that was not needed in the old code it can be required to find the name of the instance. When using rules in a file, as in this thesis, the most common names can be covered by a bit of hardcoding. However it probably is better to try to look up an instance name in the class or file. This can be done by first looking for an instance of the correct type in the local scope if one exists. If not, the global scope of the class and after that possible super classes can be checked. If no instance of the object is found a new instance can be created. These "advanced" kinds of insertions that are somewhat flexible to the code into which they are inserted, are more unpleasant to do by hand. When doing a similar library replacement by hand one will have to find all transformations of a certain type in one sweep. These are potentially spread over hundreds of files in which the transformation takes place. The other strategy when replacing the libraries by hand, is to transform one file at the time, it has another deficiency. Namely that, for every time one comes to a new file, one has to look up the transformations again. This will make the transformation time increase which will probably make it go closer to the time for writing a new line of code. Either if one goes for fixing all transformations of a certain type at the same time or if one file at a time is fixed, the end result has a lot of compile errors produced. In the one-file-at-a-time approach this is because helper classes has to be retrofitted for the new library. This in turn spreads the errors to new places, or at least so in the case studied.

The maintenance cost of the COCOMO model can be compared to the maintenance cost calculated with the function point method. The COCOMO calculation points to about 7 person-months per year while the function point counting points toward about 29.2 person-months using the formula from [2]. This is of course quite a difference. The reason for the discrepancy in these results may stem from the fact that the measurements were taken on a library. A library might not change as much over time as an application does, since the functionality it provides should not change over time. The significance of this is that the COCOMO maintenance model takes historic changes into account. Since historic changes of the library are used, the COCOMO probably gives a more accurate picture of the maintenance effort. Even though the estimations are quite different it is hoped that the function point calculation shall validate the COCOMO calculation. in the degree that it does, the COCOMO estimation of the cost for replacing the old library and the cuts to it by automatic transformation is also validated.

There are other effort estimation models except from COCOMO and function points. But effort estimation can also be performed with genetic models. Typical inputs are lines of code, function points and historic data, such as programs and the effort spent on them [17]. Two ways of doing genetic effort estimation is either to gather functional parameter values and to use it as input to a neural network or to try to match the efforts against similar cases from the past[17]. These ways of estimation probably are at least as good as the COCOMO or function point estimation, the reason that they are not used in this thesis is that they require historic data as well as an implementation overhead.

6.3 The work in a wider context

In a wider context, this thesis shows that analysis tool using ASTs are very useful for revealing the structure of a large piece of software. This is true when introducing new people to the software as well as for system experts. For example the examination revealed differences between the libraries, duplicated functionality in classes, copies of classes and classes with the same name but different behaviour residing in different packages.

The desired societal impact is that the results will make transformations easier, the analysis and overview of large scale software easier. Making design that has solidified due to large numbers of dependencies in large interfaces between parts of the software project easier to refactor, is also a desired goal. In the end, this is in order to increase the functionality that can be developed and maintained in a software project.

The software development process is always pushed to its boundaries. Large software projects always take a long time to mature and programmers cannot keep track of all the software elements or their relation to each other. The programmers can't possibly keep track of all necessary information for doing a required change, which leads to several problems. One being that errors are introduced into the code and another that redundant or bad design is introduced. Inadequate design then in turn leads to more errors being introduced into the code. By improving the design through automatic transformations, transformations that operate on a large scale can be achieved. This makes it possible to do transformations that are tedious to do by hand.

In this thesis we have studied the case when an old library shall be replaced by a new automatically generated one. In cases such as this, a reduction in maintenance cost can be achieved by doing the transformation.

In the method chapter 4, it is shown that the removal of an old library saves maintenance cost. This is however nothing that the developers earn money on or can show new functions that is improving the product for the end-user. Then the question is, who are going to use the tool and put an effort in the transformations? As a software engineer and developer you shall be able to see the whole spectrum of a project and understand what other stakeholders want to get out of the project. To not care about the project as a whole and just trying to get the re-

quirement of the project to pass can be a disaster for both your career and the project [9]. This comes from that it is more important to trace the value propositions then the requirements. Today when software teams work on projects in an agile environment the conversation between stakeholders throughout the project is much greater than before and the requirements can get altered very quick, that is why it is important to look at the value instead and not just the requirements [9]. If the software engineers work with the mindset that the value of the product always shall increase, then using the tool and removing an old library is one way of doing this. From removing a library the engineer shall see that the maintenance work will get down for the project but also for other stakeholders that may use this product, who may also alter the libraries and source code. This is not only done for saving money, even if that always is good to lower the costs, but also to bring more time for the software engineers to do actual improvements of functionality for the product. As shown in section 4.4.2 in the method chapter, in the project looked at in this thesis, almost 7 person-months can be saved from doing maintenance which then can be used for improving or adding new functionality to the project.

7 Conclusion

The purpose of this thesis is to investigate if it is possible and/or economic to try to do large-scale code transformations automatically. The answer is that it is certainly possible, but has an economic threshold. It is only economic for large transformations and/or where the transformation rules can be reused. This is shown in the first research question, "How much of a transformation between two libraries is it possible or economical to automate?". To answer this question, a transformer tool was developed. The transformer tool, taking written rules together with automatically made rules as a base for the transformations, showed that the average rule was 3 lines of code. With the assumption that an invocation is 1 line of code, it is shown that to benefit from making rules there shall be more than 3 invocations that the rule will fix. From using the transformer tool on a real project together with some calculations, it was shown that the work for transforming the usage of one library to another will take about 2.9 person-months. This can be compared with the calculated value of doing everything manually, which corresponds to about 12.5 person-months. Using the transformer tool shows a saving of 77.6% compared to doing the transformations by hand.

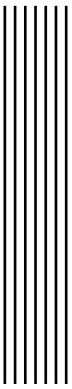
Much like the situation in [38], the use of a team for an activity that does not improve the functionality for a long time is hard to justify. Being able to successfully estimate the size of a software change and the time it will take as well as to cut some of the preparatory work leading up to the transformation will increase the attraction of transformations as well as mitigate the risk.

Doing the transformations by hand is hard or not even possible in some projects, which shows another purpose of doing the transformations automatically. The 5000 invocations seen in the project considered in this thesis may not seem much, and in the perspective of large software systems it is minuscule. However, putting a team on doing the transformations by hand is estimated to take 12.5 person-months, calculated in equation 5.2. Because the project is always evolving, with several new commits every day, it would be hard to merge the transformations after 12 months. Even if a small group of 4 people were working with this, after 3 months a lot of changes would have been made to the code base and a merge would be hard. Instead of doing all the transformations at once and get the problems with merging, one can think of doing the transformations gradually. This is however, not always possible, as in the case of this thesis where the project had high coupling. This means that one change in the code often leads to another change in the code and so on.

The main idea behind the transformation of libraries is to decrease the maintenance cost for the project. This is also reflected in the second of the research questions as "How can a partly automated tool for code transformation decrease maintenance cost in the form of needed work in person-months?". In the case of this thesis where two libraries, with the same functionality, were used at the same time it is easy to see that removing one library will decrease the maintenance. The question is by how much and if it is worth doing. Calculations based on how many lines of code that were added or deleted during a year showed that maintaining the old library had a cost of around 7 person-months each year. This shows that removing the old library can make good savings for the project. This is somehow based on the fact that the new library are already in place and that the maintenance will not rise significantly with the new invocations. In this case the new library only started to use 38 more methods when all the invocations to the old library were replaced. This is not much based on the fact that the new library started at 850 originally used methods. The new library is also auto-generated which minimizes the need for maintenance and the value of changing the library increases even more.

7.1 Continued work

In order for code analysis and code transformations to take off as concepts, new tools are needed to be developed. AST tools must be easier to configure and use, this has been one of the main concepts of Spoon where all the analysis and transformations shall be made in Java to make it easier for developers to use [35]. However some lack of documentation have made it hard to use from time to time. Since ASTs are used for parsing many languages, Java among others, it is fairly certain that analysis and transformation tools building on the AST representation will be developed and tested to a greater extent. This in order to come to terms with code transformations that are so large that they cannot be resolved entirely by hand. Analysis tools can also be used for determining the effort needed to do a certain transformation. In a possible future smart transformation tools can make some of their own decisions in situations with uncertainties.



Bibliography

[1] K.K. Aggarwal and Yogesh Singh. *Software Engineering*. New Age International (P) Limited, 2005. ISBN: 9788122416381.

[2] Yunsik Ahn, Jungseok Suh, Seungryeol Kim, and Hyunsoo Kim. "The software maintenance project effort estimation model based on function points". In: *Journal of Software Maintenance and Evolution: Research and Practice* 15.2 (2003), pp. 71–85.

[3] Alfred V. Aho. *Compilers : principles, techniques, & tools*. Boston : Pearson Addison-Wesley, cop. 2007; 2. ed, 2007. ISBN: 0321486811.

[4] Allan J Albrecht. "Measuring application development productivity". In: *Proceedings of the joint SHARE/GUIDE/IBM application development symposium*. Vol. 10. 1979, pp. 83–92.

[5] Allan J. Albrecht and John E. Gaffney. "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation". In: *IEEE Transactions on Software Engineering* SE-9.6 (Nov. 1983), pp. 639–648. ISSN: 0098-5589. DOI: 10.1109/TSE.1983.235271.

[6] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lagüe, and Kostas Kontogiannis. "Advanced clone-analysis to support object-oriented system refactoring". In: *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*. IEEE, 2000, pp. 98–107.

[7] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lagüe, and Kostas Kontogiannis. "Measuring clone based reengineering opportunities". In: *Software Metrics Symposium, 1999. Proceedings. Sixth International*. ID: 1. 1999, pp. 292–303.

[8] Barry Boehm. *Software engineering economics*. Vol. 197. Prentice-hall Englewood Cliffs (NJ), 1981.

[9] Barry Boehm. "Value-based software engineering: reinventing". In: *ACM SIGSOFT Software Engineering Notes* 28.2 (2003), p. 3.

[10] Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. "Cost models for future software life cycle processes: COCOMO 2.0". In: *Annals of software engineering* 1.1 (1995), pp. 57–94.

[11] Mark GJ van den Brand, Arie van Deursen, Jan Heering, HA De Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, and Jeroen Scheerder. "The ASF SDF meta-environment: A component-based language development environment". In: *Compiler Construction*. Springer, 2001, pp. 365–370.

[12] Kavita Choudhary. "GA based Optimization of Software Development effort estimation". In: *IJCST, September* (2010).

[13] William Cohen, Pradeep Ravikumar, and Stephen Fienberg. "A comparison of string metrics for matching names and records". In: *Kdd workshop on data cleaning and object consolidation*. Vol. 3. 2003, pp. 73–78.

[14] Krzysztof Czarnecki and Simon Helsen. "Classification of model transformation approaches". In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. Vol. 45. 3. USA. 2003, pp. 1–17.

[15] Harvey Roy Divinagracia. *Tiny tools - FP Calculator*. 2000. URL: http://groups.engin.umd.umich.edu/CIS/course.des/cis525/js/f00/harvey/FP_Calc.html (visited on 05/17/2016).

[16] Telefonaktiebolaget LM Ericsson. *This is Ericsson*. 2015. URL: http://www.ericsson.com/res/thecompany/docs/this-is-ericsson.pdf (visited on 03/17/2016).

[17] Gavin R Finnie, Gerhard E Wittig, and Jean-Marc Desharnais. "A comparison of software effort estimation techniques: using function points with neural networks, case-based reasoning and regression models". In: *Journal of Systems and Software* 39.3 (1997), pp. 281–289.

[18] G David Forney. "Generalized minimum distance decoding". In: *Information Theory, IEEE Transactions on* 12.2 (1966), pp. 125–131.

[19] The Eclipse Foundation. *JDT Plug-in Developer Guide*. URL: http://help.eclipse.org/juno/index.jsp?nav=%2F3 (visited on 02/26/2016).

[20] The Eclipse Foundation. *JDT Plug-in Developer Guide - DOM Rewrite*. URL: http://help.eclipse.org/juno/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/rewrite/package-summary.html (visited on 05/30/2016).

[21] The Eclipse Foundation. *JDT Plug-in Developer Guide - Java Model*. URL: http://help.eclipse.org/juno/topic/org.eclipse.jdt.doc.isv/guide/jdt_int_model.htm?cp=3_0_0_0 (visited on 05/30/2016).

[22] The Eclipse Foundation. *JDT Plug-in Developer Guide - Manipulating Java Code*. URL: http://help.eclipse.org/juno/topic/org.eclipse.jdt.doc.isv/guide/jdt_api_manip.htm?cp=3_0_0_1 (visited on 05/30/2016).

[23] Juan Carlos Granja-Alvarez and Manuel José Barranco-García. "A Method for Estimating Maintenance Cost in a Software Project: A Case Study". In: *Journal of Software Maintenance* 9.3 (May 1997), pp. 161–175. ISSN: 1040-550X. DOI: 10.1002/(SICI)1096-908X(199705)9:3<161::AID-SMR148>3.0.CO;2-8.

[24] Hadi Hemmati and Lionel Briand. "An industrial investigation of similarity measures for model-based test case selection". In: *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. IEEE. 2010, pp. 141–150.

[25] Bob Hunt, Bryn Turner, and Karen McRitchie. "Software Maintenance Implications on Cost and Schedule". In: *Aerospace Conference, 2008 IEEE*. ID: 1. 2008, pp. 1–6.

[26] Spirals research group at Inria Lille. *Spoon Code Elements*. 2016. URL: http://spoon.gforge.inria.fr/code_elements.html (visited on 04/13/2016).

[27] Spirals research group at Inria Lille. *Spoon Meta model*. 2016. URL: http://spoon.gforge.inria.fr/structural_elements.html (visited on 04/13/2016).

[28] Spirals research group at Inria Lille. *Spoon References*. 2016. URL: http://spoon.gforge.inria.fr/references.html (visited on 04/13/2016).

[29] Yoshio Kataoka, David Notkin, Michael D. Ernst, and William G. Griswold. "Automated Support for Program Refactoring Using Invariants". In: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. ICSM '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 736–. ISBN: 0-7695-1189-9. DOI: 10.1109/ICSM.2001.972794.

[30] Dexter C. Kozen. *Automata and computability*. New York : Springer, cop. 1997, 1997. ISBN: 0387949070.

[31] Rob Leitch and Eleni Stroulia. "Understanding the economics of refactoring". In: *EDSER-5 5th International Workshop on Economic-Driven Software Engineering Research*. 2003, p. 44.

[32] Tom Mens and Tom Tourwe. "A survey of software refactoring". In: *IEEE Transactions on Software Engineering* 30.2 (2004). ID: 1, pp. 126–139.

[33] Frauke Paetsch, Armin Eberlein, and Frank Maurer. "Requirements engineering and agile software development". In: IEEE, 2003, p. 308.

[34] Gérard Paligot, Nicolas Petitprez, and Martin Monperrus. "TTC'2015 Case: Refactoring Java Programs using Spoon". In: *Transformation Tool Contest*. 2015.

[35] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code". In: *Software: Practice and Experience* (2015), na. DOI: 10.1002/spe.2346.

[36] Denis St-Pierre, Marcela Maya, Alain Abran, Jean-Marc Desharnais, and Pierre Bourque. "Full function points: Counting practices manual". In: *Software Engineering Management Research Laboratory and Software Engineering Laboratory in Applied Metrics* (1997).

[37] Raimundo Real and Juan M Vargas. "The probabilistic basis of Jaccard's index of similarity". In: *Systematic biology* 45.3 (1996), pp. 380–385.

[38] *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-48567-2.

[39] Yongchang Ren, Tao Xing, Xiaoji Chen, and Xuguang Chai. "Research on Software Maintenance Cost of Influence Factor Analysis and Estimation Method". In: *Intelligent Systems and Applications (ISA), 2011 3rd International Workshop on*. ID: 1. 2011, pp. 1–4.

[40] Don Roberts, John Brant, and Ralph Johnson. "A refactoring tool for Smalltalk". In: *Urbana* 51 (1997), p. 61801.

[41] Charles R Symons. "Function point analysis: difficulties and improvements". In: *Software Engineering, IEEE Transactions on* 14.1 (1988), pp. 2–11.

[42] Ragnhild Van Der Straeten and Maja D'Hondt. "Model refactorings through rule-based inconsistency resolution". In: *Proceedings of the 2006 ACM symposium on Applied computing*. ACM. 2006, pp. 1210–1217.

[43] Eelco Visser. "Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5". In: *Rewriting techniques and applications*. Springer, 2001, pp. 357–361.

[44] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. "Design patterns: Elements of reusable object-oriented software". In: *Reading: Addison-Wesley* 49.120 (1995), p. 11.

[45] Robert A. Wagner and Michael J. Fischer. "The String-to-String Correction Problem". In: *J. ACM* 21.1 (Jan. 1974), pp. 168–173. ISSN: 0004-5411. DOI: 10.1145/321796.321811.

[46] Daniel C. Wang, Andrew W. Appel, Jeffrey L. Korn, and Christopher S. Serra. "The Zephyr Abstract Syntax Description Language." In: *DSL*. Vol. 97. 1997, pp. 17–17.