Eötvös Loránd University

Faculty of Informatics

Department of Computer Algebra

Quality Analysis of Test System Source Code

Advisor:

Dr. Attila Kovács

Associate Professor

Author:

Bianka Flóra Békefi

MSc student in Computer Science

Budapest, 2018

Eötvös Loránd Tudományegyetem
Informatikai Kar
Komputeralgebra Tanszék

Tesztrendszerek forráskódjának minőségi elemzése

Témavezető:
Dr. Kovács Attila
egyetemi docens

Szerző:
Békefi Bianka Flóra
programtervező informatikus MSc

Budapest, 2018

Table of Contents

1. Introduction

During the development of a software, the members of the group that work on the program can change, new members come, old ones leave, the new ones might bring new ideas or knowledge. At the same time new software design patterns might appear. Any of these events might make the developers realize that the software architecture is not ideal, it could be organized in a better way.

In the case of TTCN-3 projects one more reason which might cause the need of refactoring the software is the change of the standard. The TTCN-3 standard is continuously maintained and changed. Those practices that were perfectly fine in the previous version might turn out to cause disorganized and hardly maintainable codes in the new version. The program has to be refactored in order to create a maintainable, comprehensible, well-structured software.

The first part of this work defines the necessary terms and expressions and summarizes the different solutions and results regarding the function moving problem. In the second part an implementation of a refactoring technique is proposed for identifying functions in TTCN-3 source codes that are not at their ideal location and for moving them to the optimal one.

1. Bevezetés

Szoftverrendszerek fejlesztése során változhat a csapat összetétele, új emberek érkeznek, régebbiek távoznak, az új csapattagok új ötleteket és tudást hozhatnak a csapatba. Mindezzel egyidőben új design pattern-ök is megjelenhetnek. Ezen események közül bármelyik előfordulása ráébresztheti a fejlesztőket, hogy a szoftver szerkezete nem ideális, jobban is lehetne szervezni a kódot.

TTCN-3 projekteknél egy további ok, ami a szoftver refaktorálásának igényét idézheti elő, a szabvány változása. A TTCN-3 szabvány folyamatosan karban van tartva és rendszeresen változik. Azok a megoldások, amik egy korábbi verzióban még megfelelőek voltak, az új verzióban már szervezetlen, érthetetlen és nehezen karbantartható kódot okozhatnak. A program refaktorálásra szorul, hogy egy jól karbantartható, érthető és világos struktúrájú szoftvert állítsanak elő.

A diplomamunka első felében a megértéshez szükséges alapfogalmak kerülnek definiálásra és az irodalmi áttekintés egy átfogó képet ad a függvények mozgatására vonatkozó eddig javasolt megoldásokról, eredményekről. A második felében egy refaktorálási mód implementációja kerül bemutatásra TTCN-3 nyelven írt kódbázison a függvények áthelyezésére vonatkozó probléma megoldására.

2. Background

First, it is well-advised to provide definitions for some expressions that are necessary for understanding the rest of the thesis.

2.1. TTCN-3

TTCN-3, which stands for Testing and Test Control Notation Version 3, is a programming language developed by ETSI (European Telecommunication Standards Institute) for testing different projects, especially conformance testing for telecommunication systems. The next sections contain explanations of some of the basic terms of the TTCN-3 testing language. [9]

2.1.1. Module

The modules are the top-level units of the language which are either libraries or test suites. They have two main parts, the definitions and the module control part, both of which are optional. A module can import visible identifiers and definitions from other modules.

2.1.2. Definition

In a module, we can define types, constants, module parameters, functions, signatures, test cases, altsteps, imports, groups, external functions and friends. Their visibility can be either private, public, or friend. Imported definitions are private by default, while friend ones can only be private. Group definitions are always public and the rest of the definitions are public by default but their visibility can be modified.

2.1.3. Component

Components are execution entities on which test cases or functions are executed. They are defined as types and become active during execution. A component represents a computer, thus with multiple components a distributed system can be simulated. The behavior of a component is defined by the functions that run on it.

2.1.4. Function

Functions work pretty much the same way as in more conventional programming languages. Their purpose is to divide into smaller sections the computation of the module to define the behavior, and to organize the execution of the test cases. Functions can be parametrized, and they can either be void or they might return a value or a template. In functions we can use

local variables or fields from the module or from other imported modules. The same way, function calls from within a function can refer to a method from the same module or from imported ones.

We can limit which components the functions run on. The functions have access to the definitions of the component they run on. Such definitions or variables behave as global variables of the computer that the component represents during its execution. Functions running on different components can run in parallel as the components can be executed simultaneously, thereby creating concurrent test cases. This is useful when non-deterministic behaviors have to be tested.

2.1.5. Import

Top-level visible definitions from a module can be imported into another module. All definitions that are public can be imported and in case the modules are friends, those with friend visibility are visible too. It is possible to import only a single definition from a module, groups of definitions or definitions of a given type such as functions or constants, or even all visible definitions. As each identifier in a module must have a unique name, in case of name clashes, the imported definitions should be referred to with qualified names.

2.2. Refactoring

According to a representative definition by Fowler et al., “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.” [1]

The goal of refactoring is not to implement new features or fix bugs, but to identify code smells and remove them while preserving the functionality of the system. By improving the structure of the software, it becomes easier to understand the code and its complexity is reduced, thus both maintaining and changing the software will require less time. Refactoring improves software quality by removing unnecessary complexity. Finding bugs, understanding and maintaining the software are all easier when the code is simpler and more clear.

Refactoring tactics can be divided into two main groups, floss and root canal refactoring. During floss refactoring, developers refactor the code while simultaneously

changing the behavior as well, such as adding a new feature or fixing a bug. On the other hand, root-canal refactoring is done with the sole purpose of improving the code structure.

2.2.1. Code smells

Code smells usually indicate a deeper problem in the software. This problem is not necessarily a bug in the functionality, it can be a design flaw which increases the probability of bugs in the future. Code smells are not design flaws, they are usually symptoms of a bigger problem in the code that is hard to detect. Usually, code smells can be corrected through refactoring and the type of the code smell indicates the refactoring technique to be used.

Martin Fowler, who first introduced the concept of code smells in object-oriented programs in 1999, distinguished 22 different types, which are presented below. [1] [10] [11]

In case of *Duplicated Code*, identical or similar code structure exists at multiple places in the software. If the duplication is within the same class or in different subclasses of the same superclass, the problem can be resolved with *Extract Method* refactoring. However, if they are in unrelated classes, it is recommended to move the same pieces of code into a new class (*Extract Class*). In most cases, removing the duplications shortens the code and simplifies the structure.

In *Long Method* code smells, a method or a function is so long its length reduces understandability. These methods should be decomposed into shorter ones through *Extract Method* refactoring. However, length is not the only factor in this situation. Functions should be decomposed in such a way that those lines which belong tightly together move together to the new function.

A *Large Class*, also referred to as *God Class*, contains too many fields or methods. As a result, such classes are too complex and takes too much time to understand them. This problem can usually be resolved with *Extract Class* or *Extract Subclass* refactoring. Often, splitting up a big class helps in reduce code duplication.

The problem with a *Long Parameter List* is, that it makes calling the function complicated. Often in object-oriented softwares some of the parameters should rather be fields of the class. Long parameter lists might happen for various reasons, such as making several changes in the method or trying to make it independent from the class. Usually, this

code smell can be removed by *Replace Parameter with Method*, *Preserve Whole Object* or *Introduce Parameter Object* refactoring.

In the case of *Divergent Change*, different changes are made to the class for various reasons as the class is responsible for various behaviors and functions which violates the Single Responsibility Principle. By *Extract Class* refactoring, the different behaviors can be divided into separate classes.

Shotgun Surgery is the opposite of *Divergent Change*, meaning that in case of a change, small changes have to be made to different classes. The solution is to move all the changes into the same class with *Move Method* or *Move Field*. If there is no good candidate for the destination, a new class has to be created.

Feature Envy code smell means that a method uses more data from another class than from its own. If there is only one class from which the method accesses more data, it should be moved there. If there are multiple classes that contain necessary fields or functions, then one destination should be chosen based on heuristics.

Data Clumps are identical groups of data items or variables which appear in various places indicating that they should be grouped together into a new class. As a result, the code becomes more organized and shorter.

Primitive Obsession is the use of primitive types even when a small object would be more convenient. By using an object it becomes more obvious what operations are available. If there is a group of such variables, they should be moved into a new class.

Switch Statements is the use of complex switch or if statements. The problem with them is that they often result in duplicated code, so when a new case is added to the switch, it has to be added at every occurrence. This problem can be resolved by moving the switch statement to the relevant class and using polymorphism.

Parallel Inheritance Hierarchies is a case of *Shotgun Surgery*. Whenever a new subclass is made to a class, you have to make a subclass to another class as well. To remove the duplication and improve the organization of the code, the instances of one of the hierarchies should refer to the other one's, then with *Move Method* and *Move Field* one of the hierarchies disappears.

A *Lazy Class* is a class that does not do much work. As trying to understand them takes too much time compared to the work they do, they should be removed by inlining the class or collapsing the hierarchy in case of subclasses that do too little work.

Speculative Generality is when a code is too general in case it will be useful in the future, although at the moment it is not necessary. Unused abstract classes, parameters, fields and methods should be removed.

Temporary Fields are fields which are assigned values or used occasionally, but for the rest of the time they are empty, making it difficult to understand them. Such fields should be extracted into a new class with all the code that uses them which increases the clarity of the code.

Message Chains are created when a requested object asks for another object from a different class which again asks for another one and so on which increases the dependency between classes. Usually the refactoring technique used in these cases is *Hide Delegate*.

A *Middle Man* is a class that delegates tasks to another class but has no other work to do. Such classes may be created when breaking down *Message Chains*. If the class does additional work, then delegation can be transformed into inheritance. However, if the class is responsible almost exclusively for delegation, then it can be removed.

Inappropriate Intimacy is when classes have too much information about or use the private methods or fields of other classes. These classes should by separated by moving fields or methods, and the fields and methods used by both can be extracted into a new class.

Alternative Classes with Different Interfaces are classes that perform the same functionality although they have different interfaces or method names. Functions with the same behavior should have the same signature (name, parameters, etc.) and in order to remove duplicated code they can be extracted as a superclass of the classes that share those methods.

In the case of an *Incomplete Library Class*, the class does not perform exactly that task which is required, so it seems necessary to change it. However, it is often impossible as libraries tend to be read-only. There are possibilities to solve this problem, for example using

Introduce Foreign Method for adding a few new methods or *Introduce Local Extension* for adding a larger amount of new behaviors.

A *Data Class* is a class that has only fields and access methods. Other classes manipulate them, they only serve as data holders. In case of a new class this is not a problem, although as time goes by most of the methods or parts of those methods that access the data should be moved into the *Data Class*.

Refused Bequest means that the subclass uses only some of the fields or methods inherited from its parents. This is not among the most severe code smells, removing it is not absolutely necessary. Nevertheless, a way to fix it is to move the unused fields and functions from the parent class to a new subclass.

Most of the time, *Comments* are not code smells. However, they can indicate a problem with the program. When there are long explanatory comments for a class or a method it usually means that the code is too complicated and incomprehensible.

2.2.2. Refactoring techniques

In his book, Fowler described 72 kinds of refactoring techniques for removing code smells. The most commonly used ones are explained below. [1][2][3]

During *Rename variable/method/class refactoring*, the name of the feature is changed so that it better reflects its behavior, thereby improving code readability. If there is inheritance involved, the changed features should be renamed in the subclasses and superclasses as well.

Move Method and *Move Field* describe the situation when a method or field uses more data or is used by more features from a class other than its own, so it is moved to the other one to reduce coupling between the classes.

Inline Method refactoring is done when the method call is replaced by its function body because it is very clear, simple and short. It is especially useful to reduce the occurrence of unnecessary delegation in the code.

Extract Method means dividing too long or complex methods into multiple smaller ones while paying attention so that both the original method's and the extracted ones' names

describe the work they do. As a result of *Extract Method* the code will be easier to understand and its reusability increases.

Extract Class is similar to *Extract Method*. It is done when a class has too many fields and/or methods, is too long and implements too many functionalities. Instead of one class, new classes are made, each of them responsible for a subset of the original behavior.

During *Extract Interface* some of the features of an interface are moved into a new one because multiple classes implement only a subset of the functionalities of the interface. Another reason for refactoring this way is when classes have different interfaces but part of the interfaces' functions is the same. In that case, the common part should be extracted.

Extract Superclass is the solution when classes have similar features which results in code duplication. The common features are extracted into a new class which will be their superclass. *Extract Class* can be an alternative depending on whether inheritance or delegation is the goal.

Pull Up Method is employed in case of inheritance, when multiple subclasses have methods that perform the same tasks. These functions are moved into their superclass, thereby removing the code duplication. It can be used even when the parametrizations of the methods are different but can be changed in a way so that the functions end up being practically the same.

Pull Down Method is the opposite of *Pull Up Method.* When some features of the superclass are relevant only to a part of its subclasses, these methods are moved from the superclass to these subclasses.

3. Problem description

During the development of a software system, its structure diverges from the original design. Although this is not necessarily a problem, usually the changes made hurt the understandability and maintainability of the code. The quality of the code decreases, implementing new features, fixing bugs become more and more difficult and time-consuming. The number of code smells might increase which indicate a deeper problem with the architecture. The need emerges to improve the structure of the program without changing its behavior, known as refactoring.

One of the problems that can arise regarding the structure is *Feature Envy* code smell. *Feature Envy* is when the distance between the current place of some of the functions and the locations where they are used is too big. The unnecessary imports make compilation slower and decrease understandability and maintainability of the code. A solution for this problem is moving the function to a different file. However, where the function should be moved is not obvious, as several aspects have to be taken into consideration. For this reason, there are multiple approaches to selecting the best new location.

4. Literature review

This chapter gives an overview of the motivations and problems regarding refactoring and summarizes the approaches to *Feature Envy* code smell detection and removal.

4.1. Motivations behind refactoring

Although software quality is an important aspect of a product, only 33% of the developers try to avoid anti-patterns and as low as 2% of the them use tools to identify and remove such behaviors [8]. The majority, more than 60%, have never heard or does not care about them. The rate is even lower among testers, as only 1% of them use such tools and only 12% knows and tries to avoid the use of anti-patterns. In another study [22] they found that even if code smell detecting and refactoring tools are available, developers do not refactor the code more often. However, if they are motivated or their attention is drawn to the importance of code quality, their productivity is increased by using such tools.

A survey [7] on refactoring observed that in contrast to the generally accepted definition of refactoring, saying that it does not change the behavior while improves the structure of the code [1], 78% of the developers define it in a way that refactoring improves some aspects of the behavior, e.g. performance. 46% of them did not even characterize it with preservation of behavior, functionality or semantics. The developers asked said, that they do 86% of the refactorings manually, while half of them said they do all of their refactoring manually even though half of them had automated refactoring tools available. Around 77% of the developers thought that refactorings might introduce new bugs. The most common advantages of refactoring were considered to be increased maintainability of the code, improved readability and reduced number of bugs. The main reasons for refactoring were poor readability, poor maintainability, code duplication and difficulties with repurposing the code.

Paloma et al. [14] found that developers generally do not consider some code smells such as *Middle Man*, *Long Parameter List*, *Lazy Class* and *Inappropriate Intimacy* as problems that need to be refactored. Those that are considered more important and concerning were *Complex Class*, *Large Class*, *Long Method* and *Spaghetti Code*.

According to Silva et al. [2], more than half of the developers refactor manually, and only 38% of them use tools provided by an IDE to refactor. The most common reasons for not using a refactoring tool were distrust in the tool, or that the changes needed were so simple that using a tool was unnecessary. Another study [13] found that 90% of the refactorings are done manually, without tools.

Black and Murphy-Hill [12] suggest that the lack of using refactoring tools is due to the fact that developers prefer floss refactoring over root-canal refactoring, however, most of the tools are created for the second strategy.

4.3. Impact of refactoring on code quality

Weißgerber and Diehl [18] found that refactorings might increase bugs in the code. They observed phases of the project, during some of which increased number of refactoring led to a higher ratio of bugs while during other phases no increase of bugs was detected. In contrast, Moser et al. [19] found indicators that refactoring leads to less complex code, simpler design and increased maintainability of the software.

Ferreira et al. [21] observed that refactorings often introduce new bugs. However, changes made during root-canal refactoring are less prone to bugs than those made during floss refactoring when the main concern of the developers is other than improving the program structure.

Shatnawi and Li [20] measured the effects of refactoring by four software quality metrics; flexibility, reusability, effectiveness and extendibility. They observed that certain refactorings such as *Extract Interface*, *Extract Subclass*, *Move Method*, etc. improved software quality while others e.g. *Pull Up Method*, *Extract Method*, *Remove Middle Man*, etc. deteriorated it. Overall, about half of the refactorings improved, around 19% deteriorated, and one third of them did not change the software quality.

Cedrim et al. [4] observed that 80% of the refactorings were performed on code smells. However, only 9.7% removed code smells, while 57% did not remove them and 33% even introduced new ones that were not removed afterwards. The most common refactorings were *Extract Method*, *Move Field*, *Inline Method* and *Move Method*. Among them, the most occurrences of neutral changes were in the cases of *Move Field* and *Move Method*, while the

most new code smells were introduced during *Extract Method*. Both *Move Method* and *Pull Up Method* increased the occurrence of *God Class* (35% and 28% respectively). The most successful one was *Inline Method*. 57% of the refactorings was neutral, meaning they did not remove the code smell but did not introduce new ones either.

According to another study [6], only 42% of the refactorings are performed on code smells and as low as 7% of them remove code smells. The most common code smells, around 16% of the cases were either *Large Class*, *Long Method*, *Spaghetti Code* or *Feature Envy*. Long classes and methods were much more likely to be involved in refactoring. On the other hand, *Long Parameter List* and *Refused Bequest* code smells were rarely removed.

As the studies above show, refactoring might increase software quality, however, developers are wary of performing such changes as they fear it would introduce new bugs. Even though multiple automatic refactoring tools are available, they are underused and most of the developers and testers are not even concerned about improving quality or removing code smells.

4.4. Feature Envy and Move Method

As the main subject of this thesis is detecting *Feature Envy* code smells in TTCN-3 codes and removing them, it is worth getting into more details of the different approaches and solutions published for this problem. However, the literature is still limited on this subject and most of the ideas proposed are for programs written in Java.

Fokaefs et al. [16] created an Eclipse plugin for Java source codes, called JDeodorant, that identifies *Feature Envy* code smells and solves them by moving the method to the right class. They calculate the distance between a method and the rest of the methods of a given class and between the method and the methods from other classes. The distance is the difference between the entities used by the method and the entities the class contains. The target class for the method is the one to which the distance of the method is minimal while its distance from the other classes is maximal.

The algorithm proposed by Napoli et al. [5] is based on modularity metrics. First, they compute Lack of Cohesion in Methods (LCOM) and Coupling Between Objects (CBO) metrics for the whole system, calculate for each method the number of methods they call and

the number of methods they are called by and measure the similarity between entities using the Jaccard similarity coefficient. Based on these metrics, they choose the target class for the method with the intention of lowering both LCOM and CBO. The disadvantage of this algorithm is that computing metrics on large software systems takes long.

Sales et al. [15] propose an approach using dependency sets. They assume that in the case of a good design, methods in classes usually create dependencies with similar types. They compute the static dependency set of a method from a class and the rest of the methods from the same class and for the other classes too. If the difference between the method's dependency set and another class' one that is smaller than from its own, the method should be moved over there. The dependency set is created by taking into consideration the method calls, fields accesses, object instantiations, local declarations, return types, exceptions and annotations.

Oliveto et al. [17] use method friendships to identify *Feature Envy* code smells. Two methods are considered friends if there is semantic similarity between them, they are in the same class, or their structure is similar or if they share the same responsibilities, e.g. they are related to the same feature or they use the same data structures. The assumption is that friend methods should be in the same class. They identify the envied class by analyzing the friendships of the methods. The envied class of a method is the one that contains the most of its friends, this is where the method should be moved.

5. Implemented solutions

The proposed implementation detects *Feature Envy* code smells in code written in TTCN-3 and recommends possible solutions through *Move Method* refactoring. In the relocation of the functions, it was important to respect the design and the architecture of the project, aspects that can not be easily measured through metrics. Therefore, before the execution of the refactoring, the users can revise the suggested locations and decide whether they find them appropriate.

Let's define *Feature Envy* in the case of TTCN-3 codes. We distinguish two different situations; *Feature Envy* based on accessed data and based on components. In the former case, a function in a module does not use any features from its own module but contains references to features from other modules, so it should be moved to one of them. In the latter case, we assume that there are several functions that run on the same component but they are not in the same module. In the case of *Feature Envy*, a function does not use any data from its current module and runs on a component on which multiple other functions run too but they are in a different module. In this situation, the function is to be moved to the module where there are the most methods that run on the same component as the moved one.

The algorithm can be divided into three main parts. First, *Feature Envy* code smells are detected in the modules. Then, all possible target modules are computed, and the optimal one is chosen based on different metrics or criteria. Finally, the methods are moved into the selected module and the missing imports are inserted.

Each of the algorithms will be demonstrated through an example. Let's assume that the selected modules are *m1*, *m2*, *m3* which look the following:

```
module m1 {

    type record R1 {
        integer field1
    }

    type component C1 {
    }

    type component D1 {
    }

    type component C2 extends C1 {
    }

    function f6() runs on C1 {
      var R1 v_1 := {field1 := 1};
    }

    function f5() runs on D1 {
      var R1 v_1 := {field1 := 1};
    }
}
```

1. Figure - Module m1

```
module m3 {
    import from m1 all;

    type record R2 {
        integer field1
    }

    function f3() runs on C1 {
      var R1 v_1 := {field1 := 1};
      var R2 v_2 := {field1 := 1};
    }

    function f4() runs on C2 {
      var R2 v_1 := {field1 := 1};
    }
}
```

2. Figure - Module m3

```
module m2 {
    import from m1 all;
    import from m3 all;

    function f1() runs on D1 {
      var R1 v_1 := {field1 := 1};
    }

    function f2() runs on C2 {
      var R1 v_1 := {field1 := 1};
      var R2 v_2 := {field1 := 1};
    }
}
```

3. Figure - Module m2

Although it is not obvious at first sight, not all of the functions are at the best location at the moment. Neither function *f1* nor *f2* contain any references to features defined in module *m2*, but they use features from other modules. Function *f1* uses a type from module *m1* and function *f2* contains references to features both from module *m1* and *m3*. Thus, a better new

location can be found for them so that they would be in a module from which they use features.

5.1. Detect methods to move

Detecting which methods should be moved from its module is based on the fields and methods the given function uses. Each assignment is examined in the function's body to see if those refer to another method, field or other feature from the module. Only features that are defined outside of the method but defined within the module are considered. If the function does not use anything from its own module, it is added to a list that contains the methods to be moved. Such analysis is performed on each method in the selected modules. So that this process would be as fast as possible, nothing is computed at this phase that is not necessary for identifying movable functions but will be needed later for the relocation.

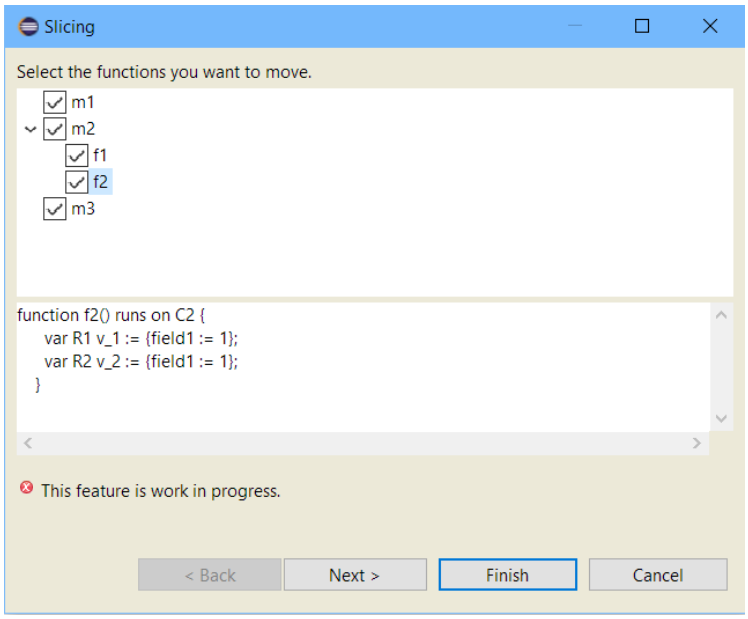
Only not private methods will be selected as movable because private ones would not be accessible from another module. It would not be a problem if the function is not used in its own module. However, if it is not used and it does not use anything, as it was considered movable, then there is no point in having that function and should be deleted instead of relocated.

```
public List<FunctionData> selectMovableFunctions(TTCN3Module ttcn3module, SubMonitor progress) {
    if (!functions.get(ttcn3module).isEmpty()) {
        return functions.get(ttcn3module);
    }
    for (ILocateableNode node : ttcn3module.getDefinitions()) {
        if (node instanceof Def_Function) {
            Def_Function fun = (Def_Function)node;
            if (fun.getVisibilityModifier().equals(VisibilityModifier.Private)) {
                continue;
            }
            ReferenceVisitor refVis = new ReferenceVisitor();
            fun.accept(refVis);
            boolean dependent = false;
            for (ILocateableNode node2 : refVis.getLocations()) {
                if (node2 instanceof Reference) {
                    final Assignment assignment = ((Reference) node2)
                    .getRefdAssignment(CompilationTimeStamp.getBaseTimestamp(), false, null);
                    if (assignment != null && !assignment.isLocal()
                        && assignment.getMyScope().getModuleScope().equals(fun.getMyScope().getModuleScope())) {
                        dependent = true;
                        break;
                    }
                }
            }
            if (!dependent) {
                FunctionData fd = new FunctionData(fun, createFunctionBody(fun, ttcn3module));
                fd.setModule(ttcn3module);
                functions.get(ttcn3module).add(fd);
            }
        }

        progress.worked(1);
    }
    return functions.get(ttcn3module);
}
```

4. Figure - Selecting movable functions

To make selecting movable functions faster, it is executed in parallel. For each selected module, a new job is started that finds the movable functions in the given module based on the algorithm explained above. As the modules are distinct, the different threads will not interfere with each other.



5. Figure - Select movable functions

In the example, as all the functions module *m1* and *m3* contain use a type from that module, no movable methods are listed below them. By selecting a function, its body is displayed so that the users can make sure this is the one they want to move.

5.2. Find destinations for Move Method

After having collected the methods to be moved, the possible target modules have to be identified. This implementation provides three main approaches; selecting the shortest module that is used by the method, selecting that of the used ones that needs the fewest new imports after inserting the function and finding the one which contains the most methods running on the same component as the one to be moved. Even though selecting the destination for the method could have been done as soon as the method was found to be independent from its own module, postponing this step results in less computational time. As the users choose some of the proposed methods to be moved, the destination selection has to be done for only on a subset of the originally listed methods. Even if all the methods are

selected, it will not take more time than if it had been computed earlier, but finding the destination for methods that are not chosen to be moved after all, would be a waste of time.

Although, at first, choosing destination based on the most features used by the method seems an obvious solution, it would not respect the design concept. For example, imagine a situation where functions are placed into different modules based on their role and the work they perform but all or some of them accesses the same data. The result of refactoring based on the coupling of the modules would be that all or almost all of the functions are moved to the modules containing the types. However, it would not reflect the desired architecture and it would increase the occurrences of *God Class* code smell. As Cedrim et al. [4] observed, *Move Method* refactoring increased the occurrence of *God Class* code smell by 35%. During the development of the refactoring tool this phenomenon was noticed at one point and one of the approaches, namely choosing the module which needs the lowest number of new imports, still might create larger modules but it can be prevented by setting the filtering options for the destination. Choosing the shortest module for destination, however, tries to balance the lengths of the modules so that none of them would become too large.

At first, only the optimal target module was given as a result of the destination selection, however, due to the request of the users, now all possible targets are listed. They are rated between 0 and 100 to show at what degree the given suggestion is recommended, to see if it is the optimal one or if there is any point at all in moving the method to that module. The users are free to choose from the listed targets, the ratings serve only as a guide.

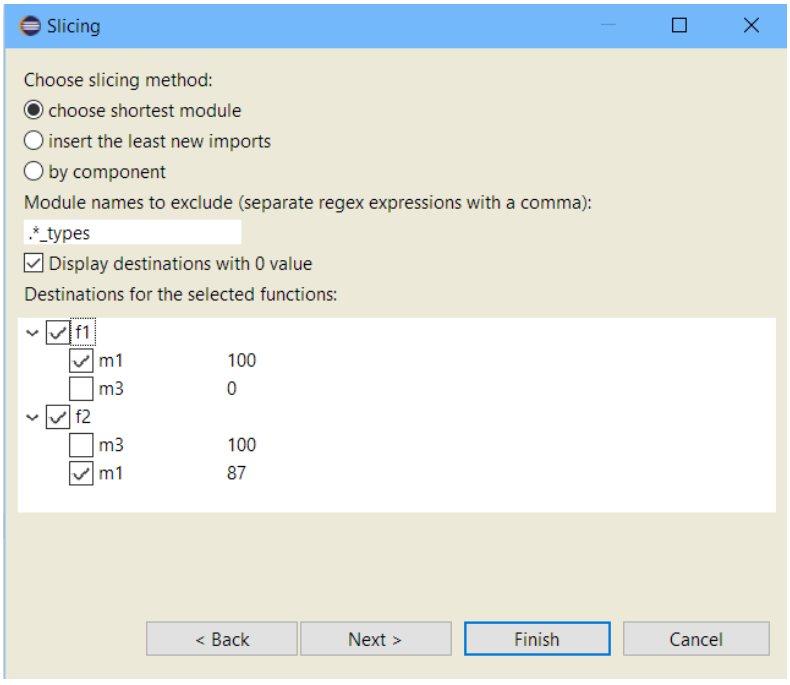
For both moving the method to the shortest module or the one with the fewest new necessary imports, the first step is identifying those modules that are used by the method. This is done in a similar way as the algorithm described in the previous chapter. Only those modules can be used by the function that are imported in their own module. For each of such modules, we have to check if the method uses a field or method or other feature from that module. As imports are not transitive in TTCN-3, it is enough to inspect those modules that are imported, as the method can not use features from any other module. If the method has references to entities from that module, it is added to the list of possible destinations.

The possible destination module names can be limited by the user by giving regular expression patterns. For example, if the files that contain the types are usually named name_types, the users can exclude such modules from the set of destinations.

5.2.1. Choosing the shortest module

In the case of choosing the shortest module, the selected possible destinations are sorted based on their length. Their degree of recommendation is calculated the following way. The shortest modules are rated 100 indicating that they are the best destinations. For the rest of the suggestions, their rate is the length of the shortest one divided by their length, displayed as percentage. This way, as the shortest ones have been chosen for optimal solution, the rest of the rates will fall between [0,100) as their length will be longer than the best one's. However, a value of 0 is highly unlikely, it would only happen as a result of precision loss during division because the module's length was so big compared to the shortest one.

All modules that are not used by the function but are among the originally selected ones, are added to the list with a value of 0. Thus, even if the users do not like any of the better suggestions but still want to move the function, they can perform the relocation within the tool. A recommendation rate of 0 would be a proper indicator of an unused module as, in all probability, a used module will get a value larger than 0.



6. Figure - Shortest module destinations

In the example, after the refactoring, both function *f1* and *f2* is moved to module *m1* from module *m2* and a new import is inserted, as *f2* contains a reference to a type from *m3*.

```
module m1 {
    import from m3 all;

    type record R1 {
        integer field1
    }

    type component C1 {
    }

    type component D1 {
    }

    type component C2 extends C1 {
    }

    function f6() runs on C1 {
      var R1 v_1 := {field1 := 1};
    }

    function f5() runs on D1 {
      var R1 v_1 := {field1 := 1};
    }

    function f2() runs on C2 {
      var R1 v_1 := {field1 := 1};
      var R2 v_2 := {field1 := 1};
    }

    function f1() runs on D1 {
      var R1 v_1 := {field1 := 1};
    }
}
```

7. Figure - Module m1 after refactoring

```
module m2 {
    import from m1 all;
    import from m3 all;

}
```

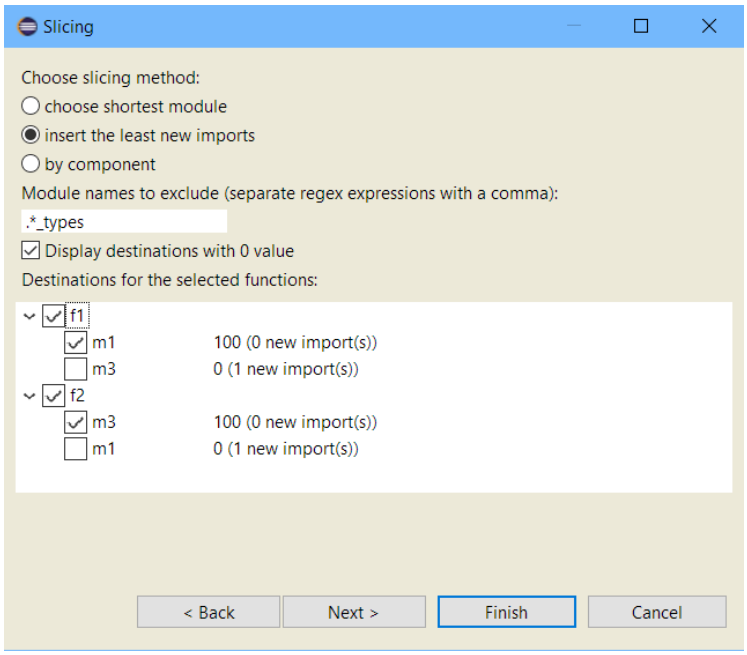
8. Figure - Module m2 after refactoring

5.2.2. Minimal new imports

In the case of choosing the module with the fewest new imports, for each of the possible destinations it is counted how many of the necessary imports they contain. The necessary modules are the same ones as the possible destinations as these are the ones from which the method uses features. Because only those functions are analyzed at this point of the algorithm that were found not to use anything of their original module, that will not be among the needed ones.

After having counted the imports in the modules, the suggested destinations are sorted by that number. The way the rate of recommendation is calculated is similar to the one in the previous algorithm, however, a few changes are made. Those with the fewest necessary new imports will get a value of 100. For the rest, their value will be the minimal new imports divided by the number of imports needed in the given module as a percentage. In this case, values of 0 might appear. If the best destination contains all the imports necessary, the number of new needed ones are 0, so no matter how many are needed in another module, zero divided by any number stays zero, so all those that need more than 0 new imports, will get a value of 0. Nevertheless, the number of needed new imports is displayed to the user who can choose the destination based on this information.

As in the previous case, not used but selected modules are listed among the possible destinations with a value of 0, but the necessary imports are calculated for them as well.



9. Figure - Minimal new imports destinations

In our example, in the case of ordering the destinations by the number of new imports, we can see that for function *f1* the best option is module *m1*, while for function *f2* it is module *m3*. By selecting the checkbox to show destinations with a value of 0, it becomes visible that they could be moved into other modules as well, but those require a higher number of imports to be inserted.

```
module m1 {

    type record R1 {
        integer field1
    }

    type component C1 {
    }

    type component D1 {
    }

    type component C2 extends C1 {
    }

    function f6() runs on C1 {
      var R1 v_1 := {field1 := 1};
    }

    function f5() runs on D1 {
      var R1 v_1 := {field1 := 1};
    }

    function f1() runs on D1 {
      var R1 v_1 := {field1 := 1};
    }
}
```

10. Figure - Module m1 after refactoring

```
module m3 {
    import from m1 all;

    type record R2 {
        integer field1
    }

    function f3() runs on C1 {
      var R1 v_1 := {field1 := 1};
      var R2 v_2 := {field1 := 1};
    }

    function f4() runs on C2 {
      var R2 v_1 := {field1 := 1};
    }

    function f2() runs on C2 {
      var R1 v_1 := {field1 := 1};
      var R2 v_2 := {field1 := 1};
    }
}
```

11. Figure - Module m3 after refactoring

```
module m2 {
    import from m1 all;
    import from m3 all;


}
```

12. Figure - Module m2 after refactoring

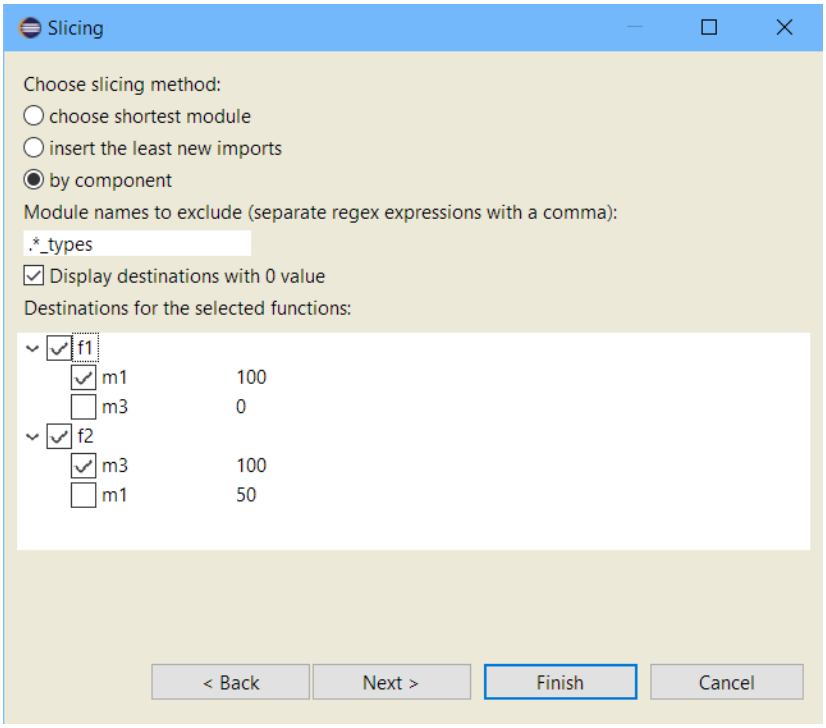
5.2.3. Moving based on component

The third way of finding a new destination is based on the component the function runs on. This can be imagined in a way as if components were classes turned inside out. The functions are the members of the class but the syntax allows them to be placed elsewhere. The goal is to group together the methods running on the same component, just like in the case of classes where the methods belonging to the same class are in the same file.

First, the component types have to be identified to find the base component for the method. If the type of the component is defined for the method, all of those components have to be collected that are extended by this type, up to the root of the type hierarchy. This way, we will have a list of all the types that extend the base type.

Afterwards, all the modules in the project have to be checked to see if they contain functions that run on any of the collected components and if they do, how many of them are in the module. The goal is to move the function to the module which contains the most of the methods running on the same component.

After the possible destinations are rated and ranked, they are presented to the user who will select from them the new destination for the function. In the case of minimal new imports, the number of necessary imports is displayed too.



13. Figure - Destinations based on components

In the example, the best new location for function *f1* would be module *m1*, as that one contains the most functions running on the same component. For *f2* the best would be *m3*, because as we can see from the source code, it contains two functions that run on the same component, while module *m1* contains only one.

```
module m3 {
    import from m1 all;

    type record R2 {
        integer field1
    }

    function f3() runs on C1 {
      var R1 v_1 := {field1 := 1};
      var R2 v_2 := {field1 := 1};
    }

    function f4() runs on C2 {
      var R2 v_1 := {field1 := 1};
    }

    function f2() runs on C2 {
      var R1 v_1 := {field1 := 1};
      var R2 v_2 := {field1 := 1};
    }
}
```

14. Figure - Module m3 after refactoring

```
module m2 {
    import from m1 all;
    import from m3 all;


}
```

15. Figure - Module m2 after refactoring

```
module m1 {

    type record R1 {
        integer field1
    }

    type component C1 {
    }

    type component D1 {
    }

    type component C2 extends C1 {
    }

    function f6() runs on C1 {
      var R1 v_1 := {field1 := 1};
    }

    function f5() runs on D1 {
      var R1 v_1 := {field1 := 1};
    }

    function f1() runs on D1 {
      var R1 v_1 := {field1 := 1};
    }
}
```

16. Figure - Module m1 after refactoring

5.2.4. Comparing the different approaches

By comparing the three methods, it is visible that only the first one, moving to the shortest module inserted a new import. As it was a small example, no conclusion can be drawn from this regarding the number of new imports or whether the destination modules would be the same for all three options in larger projects.

Approach	Functions	Original module	Module after refactoring	Number of inserted new imports
Shortest module	f1	m2	m1	0
	f2		m1	1
Minimal new imports	f1		m1	0
	f2		m3	0
Based on component	f1		m1	0
	f2		m3	0

5.3. Imports

After the final destination for the method has been selected, the missing imports have to be inserted. The missing imports are those modules, that the method uses but are not among the imported modules of the destination. In the case of moving multiple methods into the same module at the same time, it is important to check that one module should be imported only once. In case both methods want to insert the same new import, only one should be added.

```
private InsertEdit insertMissingImports(Module destinationModule, List<Module> usedModules) {
    List<Module> importedModules = destinationModule.getImportedModules();
    String importText = "";

    for (Module m : usedModules) {
        if (!importedModules.contains(m)
                && !m.equals(destinationModule)
                && !moduleImports.get(destinationModule).contains(m)) {
            importText += "import from "+m.getIdentifier().getTtcnName()+" all;\n  ";
            moduleImports.get(destinationModule).add(m);
        }
    }
    final TextFileChange insertImports = new TextFileChange(destinationModule.getName(),
                                        (IFile)destinationModule.getLocation().getFile());
    final MultiTextEdit rootEdit = new MultiTextEdit();
    insertImports.setEdit(rootEdit);
    int offset = destinationModule.getLocation().getEndOffset();
    Assignments assignments = destinationModule.getAssignments();
    int nOfAssignments = assignments.getNofAssignments();
    for (int i=0; i<nOfAssignments; i++) {
        int assignmentOffset = assignments.getAssignmentByIndex(i).getLocation().getOffset();
        if (offset > assignmentOffset) {
            offset = assignmentOffset;
        }
    }
    rootEdit.addChild(new InsertEdit(offset, importText));
    if (importText.equals("")) {
        return null;
    }
    return new InsertEdit(offset, importText);
}
```

17. Figure – Find missing imports from destination module

Those modules that import the original location of the method have to be checked to see if they contain references to the function. If they do, then their imports have to be examined to decide if they import the new destination. If they do not, then after the moving of the function, a new bug would be introduced, as the module and the feature that used the method would no longer have access to it. Thus, a new import to the destination module has to be inserted.

5.4. Parallelization of method selection

Let's discuss in more detail the parallelization of the movable function selection. First, the algorithm was implemented sequentially. As a result, in the case of large modules, selecting the methods to be moved took a long time and meanwhile the user interface was not responsive. As only one module could have been processed at a time, when the users wanted to select multiple modules, they had to wait with the selection of the next one until the previous one had been processed. In the cases of large projects this took several minutes which would make the use of the tool uncomfortable.

After the parallelization, when a module is selected, the computation starts on a separate thread; therefore, the user interface stays responsive and the user can select further modules without having to wait for the previous thread to finish. This way multiple modules can be processed at the same time, which leads to less computational time in total. As each thread processes exactly one module and the modules are distinct, the threads will not lock the same resources, it will not lead to a deadlock. The only occasion when synchronization is necessary is during refreshing the user interface, when after having found the movable functions for that module, the results are displayed.

For more convenient use, a progress bar shows the degree of completion for each selected module. Its value is calculated from the number of processed definitions and the total number of definitions in the module.

```
WorkspaceJob wj = new WorkspaceJob("Find functions in module: "+event.getElement()) {
    public IStatus runInWorkspace(final IProgressMonitor monitor) {
        try {
            SubMonitor progress = SubMonitor.convert(monitor, ((TTCN3Module)event.getElement())
                                                .getDefinitions()
                                                .getNofAssignments());
            progress.setTaskName("Analysis of module: "+event.getElement());

            final Object[] children = refactoring.selectMovableFunctions((TTCN3Module)event.getElement(), progress)
                                            .toArray();
            progress.done();

            Display.getDefault().syncExec(new Runnable() {
                @Override
                public void run() {
                    if (children != null) {
                        tree.add(event.getElement(), children);
                        tree.setSubtreeChecked(event.getElement(), event.getChecked());
                    }
                    tree.expandToLevel(event.getElement(), 2);
                    setCheckedFunctions();
                }
            });
            return Status.OK_STATUS;
        } finally {
        }
    }
};
wj.setSystem(false);
wj.setUser(true);
wj.schedule();
```

18. Figure - Parallelized movable function selection

5.5. Visitor pattern

The TTCN-3 projects are represented in a graph, where the nodes are the different types and language elements such as modules, types, parameters, etc. Whenever the members of a module or a function body have to be analyzed, its subgraph has to be iterated over. This is done by the visitor pattern, meaning that new operations can be defined for a type without changing its class.

For example, it was used to decide if a given function body contains references to features from the same module. So each node from the graph that represents the function was iterated over, and the references were collected which was used later to decide if any of them refer to a feature from the same module. Practically the same was done when those functions

were searched that run on the same components or when those modules had to be identified that contain references to the moved function.

```
private static class ReferenceVisitor extends ASTVisitor {

    private final NavigableSet<ILocateableNode> locations;

    ReferenceVisitor() {
        locations = new TreeSet<ILocateableNode>(new LocationComparator());
    }

    private NavigableSet<ILocateableNode> getLocations() {
        return locations;
    }

    @Override
    public int visit(final IVisitableNode node) {
        if (node instanceof Reference) {
            locations.add((Reference)node);
        }
        return V_CONTINUE;
    }


    private static class LocationComparator implements Comparator<ILocateableNode> {

        @Override
        public int compare(final ILocateableNode arg0, final ILocateableNode arg1) {
            final IResource f0 = arg0.getLocation().getFile();
            final IResource f1 = arg1.getLocation().getFile();
            if (!f0.equals(f1)) {
                return f0.getFullPath().toString().compareTo(f1.getFullPath().toString());
            }

            final int o0 = arg0.getLocation().getOffset();
            final int o1 = arg1.getLocation().getOffset();
            return (o0 < o1) ? -1 : ((o0 == o1) ? 0 : 1);
        }
    }
}
```

19. Figure - Reference visitor

6. User feedbacks

The software was regularly tried and tested by the future users who then gave feedbacks, such as the need for further features, presence of bugs or changes needed to be made. This chapter describes how the software changed due to the users' feedbacks and requests to create a useful tool.

Originally, the plan was to implement two approaches, finding the shortest module or the one with the minimal new needed imports. In the first version, only the best destination was displayed, the potential destination modules could not have been filtered by their name and the whole selected project was processed, not only the selected modules. The users found that although the algorithm worked well, it was hard to respect the design, as some modules were created by another group of developers than others or for different products and the refactoring should not be made across these different fields. Thus, they requested that the subjects of the refactoring could be limited in a way that only those modules would be refactored that are selected, not the whole project. This way, both the modules from which methods are moved and the destinations would be among the selected ones. Another suggested solution for the problem was that the tool would provide the option to exclude the given destination module names. One of the reasons for this request was that the code was often organized in a way that the types were in modules with a postfix of "_types", while the modules containing the functions had a "_functions" postfix. The tool would have recommended to move the functions to the modules with the types which would not respect the desired architecture. By being able to filter the possible destination names, this problem can be solved.

After having implemented the requested features, the next version was tested by the users. A new idea they had was to enable the users to move the function to a different module than the proposed destination. And another destination finding approach was requested, that besides length and imports, there would be an option to find new location based on the component the method runs on. The way it was implemented was that instead of further automatizing the refactoring, we took a step back and gave more control to the user. The original refactoring - that automatically selected the methods from the modules, chose the destination and displayed only the final results, comparing the original locations to those after

the refactoring would be executed - had been divided into three steps. First, the selected modules are listed with the movable functions from which the users can choose the ones they want to move. Then on the next page the destinations are listed for the methods to be moved and this is where the users can set filtering aspects. The third page, like in the previous version, presents the differences before and after the refactoring.

As finding the movable methods for the selected modules was very slow in the case of large projects, instead of displaying all the movable methods for all of the originally selected modules, only the modules were displayed first. After selecting one, the movable functions from that module were listed. As a result the users did not have to wait a long time before the user interface appeared as finding the functions were postponed and maybe not even all of them had to be found in case not all of the originally selected modules were selected once again from the tool. As this solution was still slow, the movable method detection part has been parallelized as it is explained in a previous chapter.

After the changes made to decrease computation time, the users were able to test again the new version. They found that during moving the functions, their comments stay in their original module. It is due to a platform limitation, so no solution has been proposed for that. However, as they tend to revise the refactoring suggestions before executing it, they will know where they should copy the comments. A new request was that not only the optimal destination would be displayed but all of the possible destinations with a rating to show to what extent it is recommended. They also asked that those modules which were selected but there is no point in moving the function there based on the selected option would also be listed. This way the users could move the methods by using the tool even when no better location was found but the users have some background information on where the method should be. In the final version of the software these features have been implemented along with the refactoring option based on the component the method runs on.

7. Measuring changes

7.1. Tools and aspects

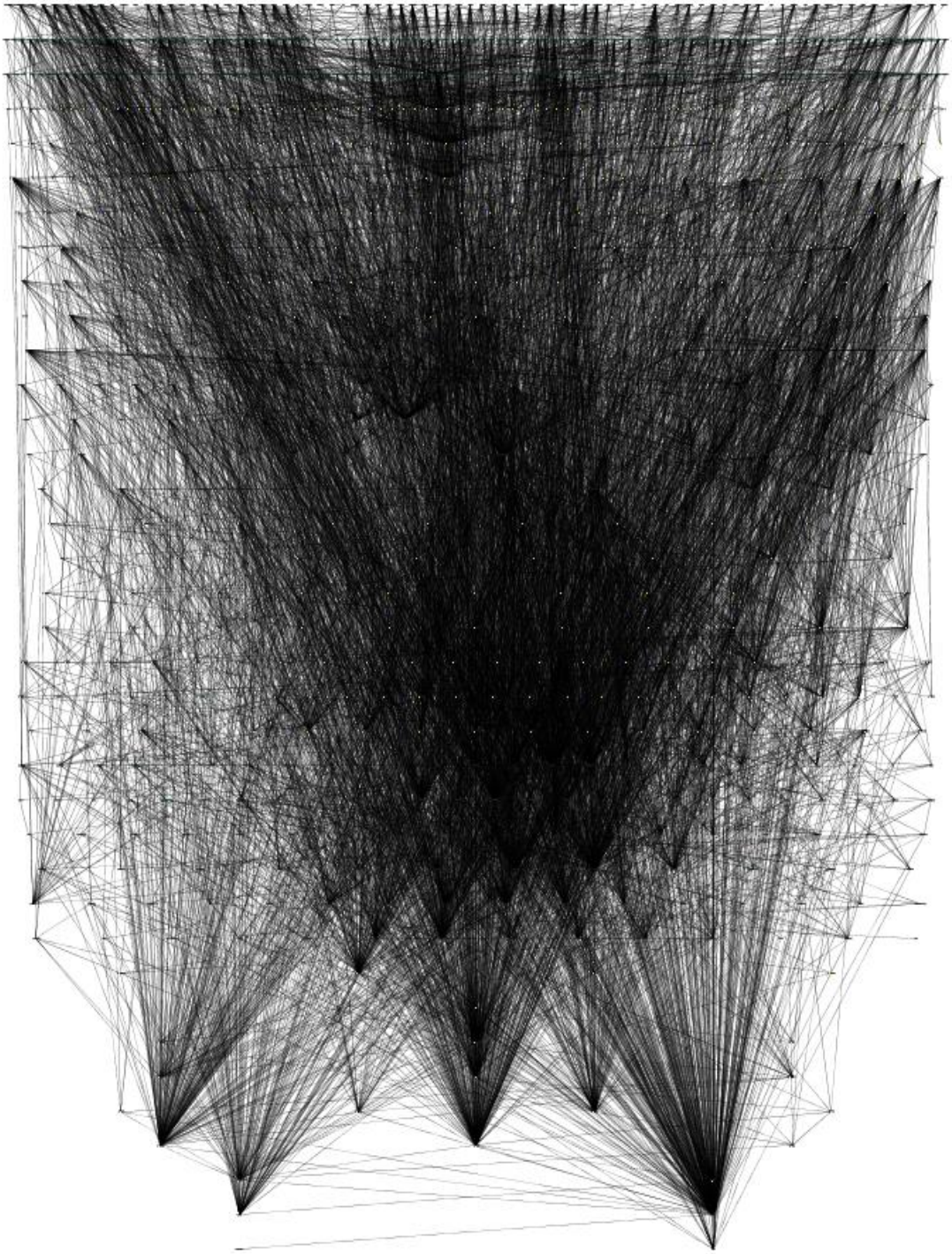
There are multiple software quality metrics available for TTCN-3 codes in Eclipse. Those that reflect the changes best caused by the implemented refactoring are lines of code, afferent and efferent coupling, number of imports and external feature envy. Afferent coupling is the number of modules that contain references to the given module, while efferent coupling means the opposite, the number of modules that are referenced from the analyzed module. External feature envy gives the number of references to features that are defined outside of the module.

During measuring the refactoring's effects, always the best recommendation was chosen except when the name of the module suggested that it contains only type definitions. In that case, if there was such a possible destination which had a rate bigger than 0 and its name did not suggest that it contained only types, this module was chosen. If there was no such destination, the function was left in its original module. In both projects, a large module was chosen to be refactored whose name did not suggest that it served to group together functions or types and contains movable functions.

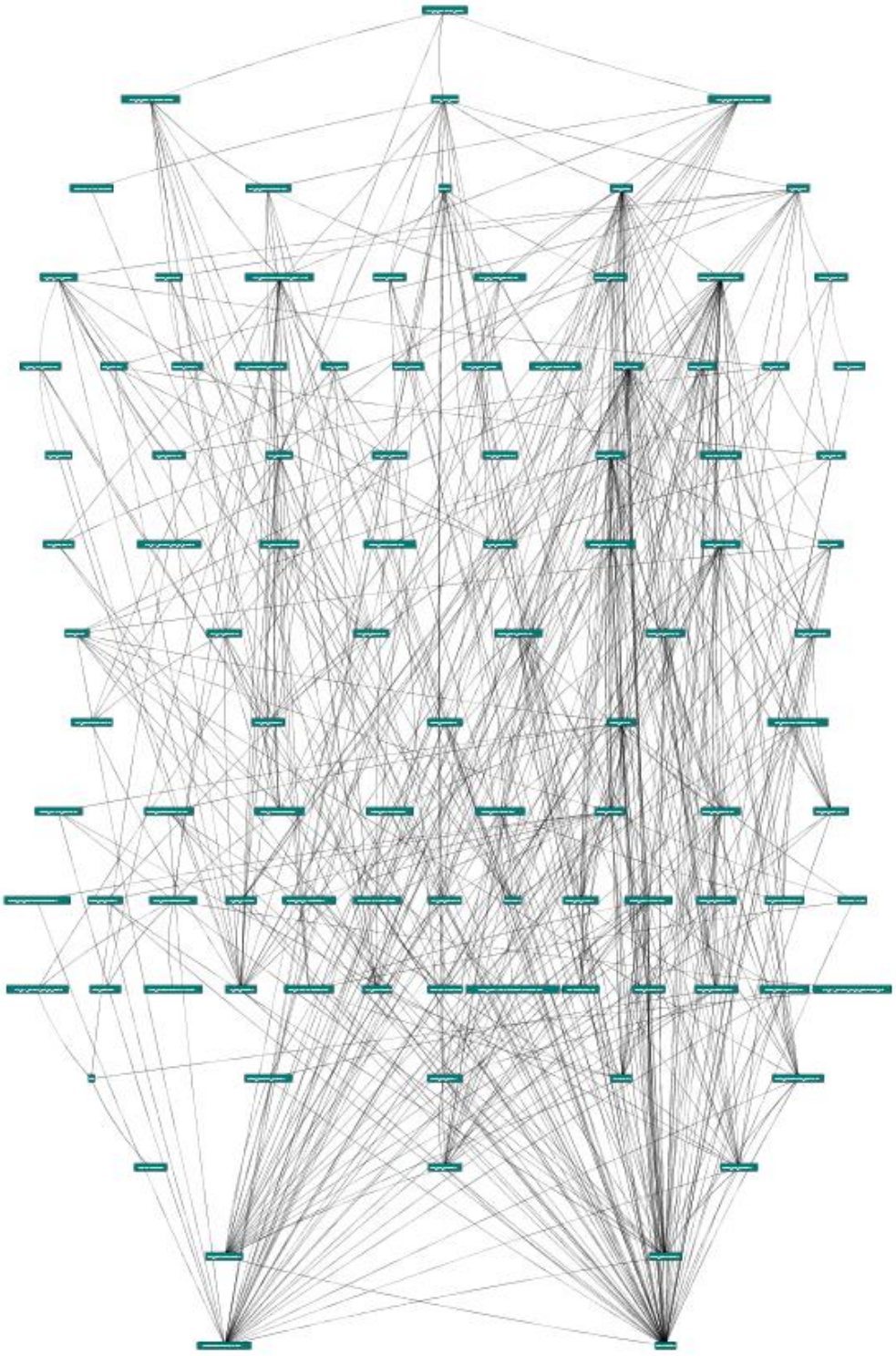
7.2. Measured projects

The results of the algorithms has been measured on two projects; on an industry project called TitanSim from Ericsson and on the 3GPP_IMS [23] test suite. The TitanSim project contains about 1 million lines of code, 942 modules and there are 7993 imports among them which means that the dependency between the modules is quite complex. The IMS test suite is smaller with 102 modules and 527 imports.

The next two pictures show the structure and the dependency between the modules.



20. Figure - TitanSim module structure



21. Figure - IMS test suite module structure

7.3. Shortest module results

In the case of choosing the shortest module, both the average length and the standard deviation of the module lengths decrease slightly. Thus, it looks like it is a good way of refactoring without risking that all functions would end up in the same module in the end. Indicators of coupling decreased significantly both for the refactored module and the whole project which shows that the functions were put into the right module after the relocation. On the other hand, the number of imports increased as those modules had to be imported in the destination module that the moved function uses and the destination had to be imported in all those modules that contained references to the function but did not have the new file imported.

Project	TitanSim	IMS
Refactored module	EPTF_GenApp_DIAMETER_LGen_Functions	EUTRA_CommonProcedures
External feature envy in module/project before	4628/255540	268/8074
External feature envy in module/project after	3816/255166	139/7742
Afferent coupling in module/project before	0/60305	5/3932
Afferent coupling in module/project after	0/60100	2/3774
Efferent coupling in module/project before	780/51933	155/3081
Efferent coupling in module/project after	678/51925	102/3037
Imports before	7993	527
Imports after	8183	552
Lines of code before: Mean Standard deviation	 1133.5 1985.3	 409.16 579
Lines of code after: Mean Standard deviation	 1133.16 1982	 409 578.36

7.4. Minimal new imports results

In the case of choosing the module which needed the fewest new imports, all the measured metrics decreased except for the number of imports. Although the goal with this approach is to insert the minimal new imports in order not to make the dependency between the modules more complex than they are, it is still necessary to add new imports.

Project	TitanSim	EUTRA
Refactored module	EPTF_GenApp_DIAMETER_LGen_Functions	EUTRA_CommonProcedures
External feature envy in module/project before	4628/255540	268/8074
External feature envy in module/project after	3786/254846	139/7960
Afferent coupling in module/project before	0/60305	5/3932
Afferent coupling in module/project after	0/60079	2/3885
Efferent coupling in module/project before	780/51933	155/3081
Efferent coupling in module/project after	670/51855	102/3033
Imports before	7993	527
Imports after	8137	536
Lines of code before: Mean Standard deviation	 1133.5 1985.3	 409.16 579
Lines of code after: Mean Standard deviation	 1133 1982.14	 407.78 579.5

7.5. Most functions on same component results

When the modules were refactored based on the component the functions run on, the coupling indicators decreased, even though the new destinations did not depend on the features the functions use.

Project	TitanSim	EUTRA
Refactored module	EPTF_GenApp_DIAMETER_LGen_Functions	EUTRA_CommonProcedures
External feature envy in module/project before	4628/255540	268/8074
External feature envy in module/project after	3699/254593	139/8070
Afferent coupling in module/project before	0/60305	5/3932
Afferent coupling in module/project after	0/59965	2/3931
Efferent coupling in module/project before	780/51933	155/3081
Efferent coupling in module/project after	651/51797	102/3100
Imports before	7993	527
Imports after	8169	552
Lines of code before: Mean Standard deviation	 1133.5 1985.3	 409.16 579
Lines of code after: Mean Standard deviation	 1132.45 1980.6	 407.78 579.5

8. Results

As it is seen in the previous chapter, all three refactoring approaches improve the software quality metrics. However, deciding which would be used in a given situation depends on the architecture of the system. In the tested projects there was a clear distinction between the modules that contained the types and those containing the functions.

As a result, the best destinations suggested by the refactoring based on the number of new imports could not give a significantly different outcome as the one based on the module length. The best destinations suggested in the former case were those that contained the types, which were not selected as a new location in order to respect the design. Therefore, the new location had to be chosen from among the modules that contained functions and the method to be moved used features from there. As there were only a small number of modules that fulfilled such criteria, these two approaches often ended up executing the same changes, which accounts for the lack of significant difference between the metrics.

Refactoring based on the component is quite different from the other ones, as it does not take into consideration the dependencies of the movable functions. As the motivation behind this was improving the structure or the design of the code, so that it would to some extent mimic the classes of object-oriented languages, it was unexpected to see that the coupling between the modules decrease, even if the change is small.

9. Conclusion

The goal of this thesis was to implement a refactoring tool for TTCN-3 code that moves functions which are not in the best module to improve software quality. This task has been accomplished as a refactoring tool is available now to detect movable functions and to move them to a new module based on the selected options and the features it uses. Three approaches have been implemented, moving the method to the shortest module that contains used features, moving to the one into which the lowest number of new imports have to be inserted and moving to that module which contains the most functions which run on the same component as the moved one.

9.1. Improvement possibilities

Even though the tool can be used to refactor modules, there are still several possibilities to increase its usability and effectiveness.

9.1.1. Scaling refactoring options

One change that would improve the tool is to make it possible for the users to fine tune the options. Namely, instead of the three options that are available now, each aspect could be weighed on a scale to show to what extent the given aspect is important. As a result, multiple combinations of the current options could be created.

9.1.2. Moving comments

At the moment, the comments that are within the moved function do not get moved to the new module. Moving the comments too would be another step in automatizing the refactoring.

9.1.3. Further refactoring approaches

Besides the three methods available, further approaches could be implemented. One possible addition is moving the function to the module where it is used the most, which contains the highest number of references to the function.

References

[1] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts: Refactoring: Improving the Design of Existing Code, 2002.

[2] Danilo Silva, Nikolaos Tsantalis, Marco Tulio Valente: Why We Refactor? Confessions of GitHub Contributors. In Proceedings of the 24th International Symposium on the Foundations of Software Engineering (FSE), 2016.

[3] https://medium.com/@aserg.ufmg/what-are-the-most-common-refactorings-performed-by-github-developers-896b0db96d9d (Last visited: April 2018)

[4] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, RohitGheyi, Leonardo Sousa, Rafaelde Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez: Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects. In Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2017.

[5] Christian Napoli, Giuseppe Pappalardo, Emiliano Tramontana: Using Modularity Metrics to assist Move Method Refactoring of Large Systems. 7th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS), 2013.

[6] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Fabio Palomba: An Experimental Investigation on the Innate Relationship between Quality and Refactoring. In Journal of Systems and Software, May 2015.

[7] Miryung Kim, Thomas Zimmermann, Nachiappan Nagappan: A Field Study of Refactoring Challenges and Benefits. In IEEE Transactions on Software Engineering, 2014.

[8] Attila Kovács, Kristóf Szabados: Knowledge and mindset in software development – how developers, testers, technical writers and managers differ – a survey. In: E. Vatai (ed.): Proceedings of the 11th Joint Conference on Mathematics and Computer Science, Eger, Hungary, 20th – 22nd of May, 2016.

[9] ETSI: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language.

[10] https://refactoring.guru (Last visited: April 2018)

[11] https://en.wikipedia.org/wiki/Code_smell (Last visited: April 2018)

[12] Andrew P. Black, Emerson Murphy-Hill: Why Don't People Use Refactoring Tools? Computer Science Faculty Publications and Presentations. Paper 115, 2007.

[13] Emerson Murphy-Hill, Chris Parnin, Andrew P. Black: How We Refactor, and How We Know It. In IEEE Transactions on Software Engineering, 2012.

[14] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia: Do they Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In Software Maintenance and Evolution (ICSME), 2014.

[15] Vitor Sales, Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente: Recommending Move Method Refactorings using Dependency Sets. In Reverse Engineering (WCRE), 2013.

[16] Marios Fokaefs, Nikolaos Tsantalis, Alexander Chatzigeorgiou: JDeodorant: Identification and Removal of Feature Envy Bad Smells. Conference Paper, 23rd IEEE International Conference on Software Maintenance and Evolution, ICSM 2007.

[17] Rocco Oliveto, Malcom Gethers, Gabriele Bavota, Denys Poshyvanyk, Andrea De Lucia: Identifying method friendships to remove the feature envy bad smell: NIER track. Conference Paper, 33rd International Conference on Software Engineering, 2011.

[18] Peter Weißgerber, Stephan Diehl: Are Refactorings Less Error-prone Than Other Changes? Conference Paper, Proceedings of the 2006 International Workshop on Mining Software Repositories, 2006.

[19] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, Giancarlo Succi: A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team. Conference Paper, 2007.

[20] Raed Shatnawi, Wei Li: An Empirical Assessment of Refactoring Impact on Software Quality Using a Hierarchical Quality Model. In International Journal of Software Engineering and its Applications, 2011.

[21] Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, Anderson Uchôa, Ana Carla Bibiano, Alessandro Garcia, João Lucas Correia, Filipe Santos, Gabriel Nunes, Caio Barbosa, Baldoino Fonseca, Rafael de Mello: Poster: The Buggy Side of Code Refactoring: Understanding the Relationship between Refactorings and Bugs. In ICSE ’18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018.

[22] Kristóf Szabados and Attila Kovács: Internal quality evolution of a large test system–an industrial study. In Acta Universitatis Sapientiae, Informatica, 2016.

[23] http://www.ttcn-3.org/index.php/downloads/publicts/publicts-3gpp/77-3gpp-ims-test-suite (Last visited: May 2018)