NTNU – Trondheim
Norwegian University of
Science and Technology

A Plugin for Visualizing and Refactoring Structurally Complex Software Architecture

Christian Thurmann-
Nielsen

Master of Science in Informatics
Submission date: June 2014
Supervisor: Reidar Conradi, IDI
Co-supervisor: Tosin Daniel Oyetoyan, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Background: Modifiability and Maintainability are important software qualities for existing software systems. However, structural complexities such as dependency cycles, are known to inhibit these software qualities. These architectural hotspots are common in many software systems, and the larger they get, the higher the chance that such flaws exists. Several studies have also shown that these structural complexities are difficult to refactor and can be difficult to detect without using code analysis tools.

Research Goals: If we could improve the overall architecture by refactoring a select few components, this could translate into a better, more maintainable software system. The main goal of this research is therefore to create a plugin that could help developers to reduce these structural complexitites and finding suitable candidates to refactor. The plugin will be a simulation tool which can simulate refactoring and visualize dependencies between components in a graph that can be manipulated.

Approach: My main researh methods will be Desgin Science and I will use several software engineering frameworks to create the architecture of the plugin. I will perform tests to verify that the plugin meets the requirements that I have set, and I will use case studies to verify how useful the plugin is and how it can help developers to refactor source code.

Expected Results: A plugin for Visual Studio, for analysing C# source code. The plugin should be able to simulate refactorings based on the analysis of the source code. The user should be able to simulate breaking cycles, moving edges and adding intermediate nodes. This tool should help developers to restructure and decouple the architecture at the class granularity level in order to make the system more maintainable.

Preface

This Master thesis is the result of the course *IT3901 - Informatics Postgraduate Thesis: Software* at the Department of Computer and Information Science under the Faculty of Information Technology, Mathematics and Electrical Engineering(IME) at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway.

I would like to thank Reidar Conradi and Tosin Daniel Oyetoyan for giving me valuable feedback and for their continued support and guidance throughout my work with this thesis.

I would also like to thank Powel for letting me test my plugin on some of their software and for their cooperation.

Trondheim, June 23rd, 2014

Christian Thurmann-Nielsen

Table of Contents

List of Tables

List of Figures

Part I

Introduction

Chapter 1

Introduction

Refactoring is an important part of any software system. To ensure that the source code meets certain quality requirements, refactoring is almost always necessary. The problem is how to know In this section I will give a short introduction of my thesis. The study focuses on detecting structural complexities in software programs, and the main contribution is a plugin that can detect and perform refactoring simulations. In this thesis I will present some of the theory behind the refactoring strategies that the plugin uses, and present my results on some case studies where I have used this plugin. In the following subsections I will give a short introduction of the motivation, the context and the structure of my thesis.

1.1 Motivation

When designing and implementing new software, developers should identify overall software qualities that the architecture should be based on. These software qualities describe general terms that every architecture should strive for, such as usability, modifiability and maintainability. However, structural complexities work against these software qualities and complicates the inclusion of such wanted behaviour. These flaws occur in almost every software architecture, and they are not easily discernible. In light of these findings, a tool that could discover and solve these unwanted architectural defects, should be useful in ensuring the greatest possible software quality. Other less complex code smells like god class, feature envy etc. already have quite a large pool of tools that can refactor and solve these problems. However, the software community seems to lack tools that can help to refactor architectural flaws, as I will discuss more detailed in section 3.4. I believe that developers generally has a skeptical view on automated refactoring tools. Tools that automatically refactor source code, could potentially make several changes in the source code. There may be a good reason behind these refactoring decisions, but how can the developer be certain that the changes are correct. Most developers are also developing in an already established environment, making the potential influence of these changes enormous. They have to trust that the changes made are completely sound, and most developers are not

ready to hand such responsibility to an automated tool.

1.2 Context

This thesis is a part of the Smart Grid Research initiative by IME, NTNU. The main goal is defined as: **Improved Management of Software Evolution for Smart Grid Applications**. More specifically, I am collaborating closely with Tosin Daniel Oyetoyan, a PhD student at IME, NTNU. He is looking into: **Software Design Complexities and Impact on the Reliability of Large Systems**. I will mostly be concentrating on developing a plugin that will discover these structural complexities inside a Visual Studio environment

The entire study has been conducted in collaboration with Powel AS. Powel AS is a major software company for Smart Grid solutions in Norway and have their main office in Trondheim. They have kindly let us perform simulations on one of their software systems, and the results of the consequent refactorings, using this plugin, is included in this study.

1.3 Contribution

Our main contribution will be a plugin for C# to analyze dependency cycles and simulate them in a graph. A developer should, with our tool, be able to analyze source code for dependency cycles. After the tool have analyzed the source code the developer should be presented with a visual representation of the structure, as a graph. After receiving the optimal graph, the developer should be able to move edges, rename nodes etc. and getting feedback regarding which classes are still in a dependency cycle. This tool should serve, not as an automated tool for refactoring dependency cycles, but rather as a tool to propose different refactoring suggestions which the developer freely can choose whether to implement or not.

1.4 Architecture

Seeing that The main contribution of my master thesis is a plugin for Visual Studio, therefore, in Chapter 4, ”Architecture”, I provide an explanation about how the plugin works, and what the architecture looks like. Here I include several different diagrams, to describe the architecture and to explain how the plugin interacts with third party libraries.

1.5 Result

We have performed several simulations on both open source projects and software at Powel. I have recorded fitness values for the projects before and after refactoring, to compare the results. These results will be discussed in chapter 7

1.6 Structure

This thesis is divided into pour parts; Introduction, prestudy, contribution and evaluation. The introduction is a short part with only one chapter dedicated to give a brief introduction to the goal of this thesis and what it should contain. The prestudy contains my findings on similar refactoring tools and some theory regarding refactoring. In the contribution I will present the plugin, it's software architecture and the results I have gotten from using the plugin. Finally, in the evaluation, I will evaluate the architecture of the software and discuss the results.

Part II

Research Design

Chapter 2

Research

2.1 Research Questions

The research questions should provide a general overview of what we want to find out and what can be concluded about the research that have been conducted.

RQ1: *Can we help developers in making architectural changes at the class granularity level, by using this plugin?*

The first thing that I want to achieve with this thesis is to make a tool that can help developers in making refactoring choices to improve the overall architecture of software. The tool should hopefully be easy to use, as the requirements in Chapter 5 reflects, and it should propose reasonable candidates to refactor.

RQ2: *How much improvement can the proposed refactorings provide?*

Another important question that I will research in my thesis, revolves around the qualitative improvement of software when using this tool. To what extent can a refactoring, that is proposed by this tool, affect the overall structure of software? And how much effort is needed to perform these refactorings? These are some of the aspects that I will investigate when analyzing the results from the plugin.

2.2 Research Method

In this chapter I will present the research method that I have used while working on my thesis.

2.2.1 Design Science

While working with my thesis I have used Design science as my research methodology. Design science is mostly used in information technology, and it specifies guidlines for evaluation and iteration for research projects. Design science focuses on an artifact throughout the development, that the research should revolve around. This artifact could be an algorithm, a solution to a specific problem or, as in this case, a tool. There exists several different frameworks for design science, but the overall structure is mostly the same. In my thesis I have used the framework designed by John Venable[31].

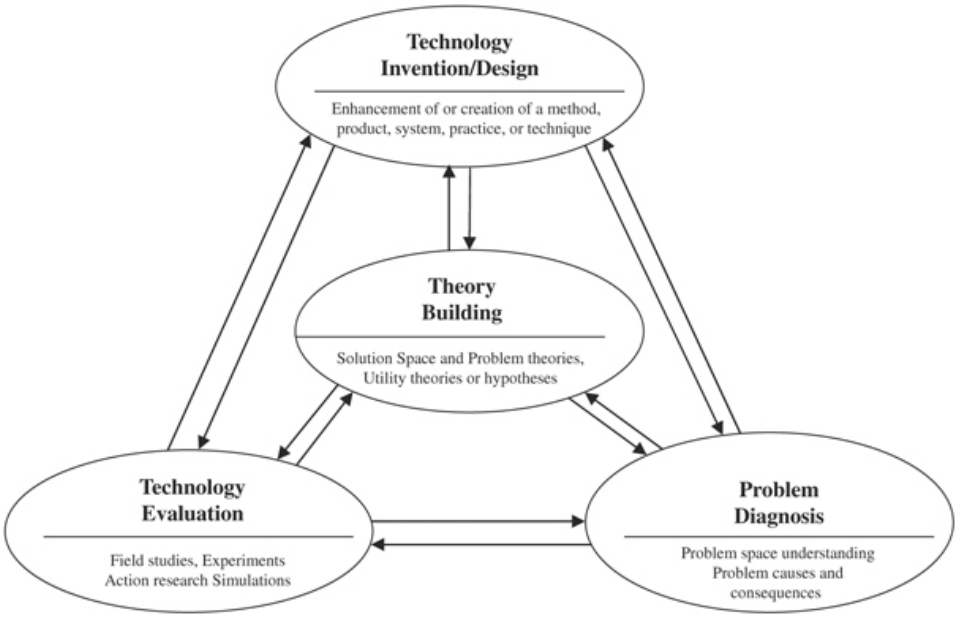


Figure 2.1: An activity framework for design science research. Venable, (2006b)[31]

In figure 2.1 you can see a diagram that shows the iterative process of design science. There are four different parts of this process, which I will describe according to the context of my thesis.

Problem Diagnosis: The context of this tool is the refactoring of structural complexities in software. In this regard, it is important to analyze why this is important and why it happens. This is covered in some extent, in state of the art, in chapter 3, where I present the problem and give some motivation as to why we should perform these types of refac-

torings. Furthermore, the problems are also mentioned in the discussion part of my thesis, in chapter 10.

Theory Building: This part of the process is where the problem that should be solved are thoroughly examined, and discussed. The theory of this thesis have been presented in the state of the art, and the solution to these problems are presented in chapter 4. Generally speaking, the main challenges in my thesis is parsing a solution and the underlying models for simulation. This, however, is only briefly discussed as it's not directly part of my thesis, but rather a part of the analysis tool developed by Tosin Daniel Oyetoyan, which is described in his paper[26].

Technology Invention: In this context the invention artifact is the plugin in itself. I have created a new tool for visualizing class specific dependencies and simulating refactoring on an architectural scale. The technological invention should be described and compared to existing tools. In state of the art I present some existing tools and compare them to the plugin I am developing.

Technology Evaluation: To verify the validity of the technological invention, it's important to evaluate it. When working with my thesis I have tested the tool in an industrial setting, at Powel and with an open source application. I have also presented the plugin and the results that I have gotten, at Powel. An interview with the developer that we worked closely with at Powel, has also been conducted.

Part III

Prestudy

Chapter 3

State of the Art

There exists many software tools today that strive for better refactoring methods and algorithms. In this section I write about some of the tools that I have found that has similar goals or include similar features that this plugin also will include. This is important for ascertaining whether or not this plugin is something that could contribute to the field of software refactoring or if it's redundant. First, I will shortly introduce some of the more common code smells, what they are, how they affect the software and how we can refactor them. Then I will go into more detail on the activity of refactoring, what methods that are most common and they affect the structure of the software. Then I will summarize my findings on some refactoring tools, what features they include and how they compare to the plugin that I am creating.

3.1 Code Smell

This work concerns restructuring of software program dependencies at the class level. It thus complements other studies and tools for removing code smells in software programs. However, its focus is refactoring architectural anti-pattern such as dependency cycles among software classes. Since this tool is only simulates the refactoring of components in a solution, the refactoring must be manually performed by developers. This plugin will also only report refactoring strategies that concerns the overall structure of the source code. That does not mean however, that these structural complexities, which this plugin tries to solve, are the only code flaws that has any impact on the software quality. Developers should also take a note when they encounter code smells.

Code smells is an indication of deeper problems in the software. Most code smells have certain characteristics, which why some call them anti-patterns. These include problems like God class, Feature envy, Long method etc. Most software systems, especially systems that are large and/or have many contributors, usually consists of many different code smells. These code smells can of course range from being insignificant, like bad comments, or could be very significant, like feature envy. Most of these smells are trivial

to refactor, like naming conventions, and tools for refactoring these exist in most IDEs. Other non-trivial code smells are more tedious to refactor, like duplicated code. However, there already exists several tools for refactoring all these code smells, and compared to the structural complexities that this plugin tries to solve, these code smells are relatively easy to refactor. In the several subsections following, I will describe some of the most well known code smells, why we should avoid them and how we can get rid of them. When describing how to refactor these code smells I will mention certain refactoring techniques, which will be described in section 3.3, Refactoring.

3.1.1 Duplicated Code

If there are a large portions of code which are very similar, we have duplicated code. This is a very common code smell and especially prevalent for inexperienced developers who are not that concerned with the design. There are several apparent flaws with this. First of all, reusability is something every developer should strive after. So the fact that similar code exists several places in the code clearly works against the notion of reusability. Performance is another quality attribute that suffers from duplicated code. Imagine having some subclasses of a parser class. These parser subclasses all contains their own parse method. The ImageParser class parses an image, the PersonParser parses an object of type Person etc. Now, when using these methods, every method must be put into the memory. Even though these different parse methods will clearly contain similar code, the compiler won't understand this, and the memory will contain duplicate code. If these methods are accessed at certain times we could also get a scenario where one method is removed from memory to make space for the other, which clearly should be avoided. As this affects both memory usage and processing speed it's easy to justify getting rid of duplicate code.

```
public void SomeMethod(ArrayList l1, ArrayList l2)
{
    l1.Sort();
    foreach (object o in l1) { PerformActionOnObject(o); }
    l2.Sort;
    foreach (object o in l2) { PerformActionOnObject(o); }
}
public void RefactoredMethod(params ArrayList[] lists)
{
    foreach (ArrayList l in lists) { PerformActionOnList(l); }
}
public void PerformActionOnList(ArrayList l)
{
    l.Sort();
    foreach (object o in l) { PerformActionOnObject(o); }
}
```

Figure 3.1: Example of duplicated code

There are several ways to get rid of duplicated code, and it depends on where the duplicated code is placed relative to each other. In figure 3.1, you can see perhaps the easiest example, where the duplicated code is in the same method. The method has been refactored into two methods. In the first we loop through the lists and in the other we loop the objects in a list. Notice also that this method has been further generalized by the use of the params keyword, which now enables this method to take an arbitrary amount of ArrayLists as parameters when calling the method. Another example of duplicated code could be when the duplicated code is placed in two different methods within the same class we could simply extract the two methods into three methods; one which includes all the similarities and one for each of the original two methods which would contain the differences for each of them. Hopefully the method which contains the similarities will be bigger than the other two methods. The observant reader will note that this actually introduces one extra method; where we originally had two, we now have three. However, the overall number of code lines should be decreased(it definitely is if you disregard the method definition of the newly introduced method). The next scenario is when the duplicated code is in two different sibling classes. In the superclass of these sibling classes you can extract the similarities from the two methods and create a new method. If it's necessary you could also pull some of the fields from these subclasses up to the superclass.

3.1.2 Feature Envy

One example of code smell is Feature envy. Feature envy is when a component uses methods or functions from many other components. Which is why we call it Feature "envy", it's envious of the other components because they have functionality that it need. This problem is naturally more prevalent with inexperienced programmers and is a very common error when designing GUI components, which uses many different components and calls many different methods on these components. A solution to this problem can be to move the method, since it clearly doesn't want to be where it currently is.

The refactoring method that we call "Move Method" is applicable here. What it does is simply that; it moves the method from one place to another. As you can see in figure 3.2, the class GUIClass is "envious" of the class GUIObject because it calls three different methods on an object of this type. The solution here is to take all the methods that is called in GUIClass on GUIObject and move them to the GUIObject class. But what if the method contains only a small part that is envious of other components? Should we still move the whole method? The answer could be that we should just move a subsection of the method to another, more suitable, place. This refactoring procedure is called "Extract Method", and extracts a certain part of the method and moves it elsewhere. A problem that is prevalent in this circumstances is where we should move the whole, or part of the method. In most cases, these components uses methods from several other components. So, which component should be the new home of this method? A solution to this would simply be to examine all the components, and move the method to the component that is called upon most times in the method. Another solution could be to perform "Extract Method" several times until all sections of the method has been moved to the software component responsible for the methods that is called from the method.

```
class GUIClass{
        private GUIObject guiObject;
        private void InitializeComponent(){
                guiObject.Enabled = true;
                guiObject.SetTitle("Title");
                guiObject.Parent = this;
        }
        private void RefactoredInitializeComponent(){
                guiObject.InitializeComponent(this, "Title");
        }
}
class GUIObject{
        public void InitializeComponent(GUIClass parent, string title){
                this.Enabled = true;
                this.Title = title;
                this.Parent = parent;
        }
}
```

Figure 3.2: Example of feature envy

3.1.3 God Class

A God Class, or Large Class which it's also called, describes a class, or software component that has too much responsibility. These classes often include a host of member fields from other classes. All these member fields are often uncritically placed in one class, where it perhaps it not used as much as to justify it being placed there. A solution to this problem would be to delegate different member fields to several sub classes of the original god class. These sub classes should include the member fields of the god class that have something in common to increase the cohesion of these sub classes. If necessary, the original god class, now stripped of all it's functionality, could act as a controller or delegator class which could include a member field of all the sub classes. Other classes will then only need one reference to the delegator class instead of several references to the sub classes. According to a study by Olbrich et. al.(2010)[24] the God Class is actually not especially harmful as long as the size is not of extreme proportions. As other code smells, one would expect a god class to include more defects and a higher change frequency. However, the paper concludes that, as long as it's within a reasonable size for a god class, the results are the exact opposite.

3.1.4 Lazy Class

Contrary to the previously discussed God Class, a Lazy Class is a class that does almost nothing. To define how much responsibility or code a class should include to not be labeled a lazy class is not straightforward. There are several different definitions and some

developers doesn't regard a lazy class as a code smell. A Data Class for instance, which is a class that only includes member fields and getters and setters for these fields, could easily be described as a lazy class in certain scenarios. However, these type of classes can also be just as viable as any other type of class in another scenario. Even though there exists several different definitions of a lazy class, we could still be able to identify certain qualities, or rather lack of qualities, that could describe a lazy class. For example if a class only contains a single member field and a getter and setter method, we could ask ourselves: Do we really need a class to contain a single field? The answer, in most cases, should be no.

3.1.5 Long Method

The next code smell represent a method that consists of too much functionality, and is simply too long. A long method is characterized that it contains functionality that should be divided into many smaller methods. Negative impacts of a Long Method could be performance. If a developer wants to use just one out of the many features that the method offers, time is spent on performing unwanted operations. Another concern is reuseabilty. The long method may contain functionality that we want to use elsewhere. This can of course be done, but we have to import the component first into the other components who wants to use the method. However, the long method contains other unwanted functionality, which perhaps shouldn't even be in the given software component. So, every other component that wants to use the long method may have to import unwanted functionality which could be avoided. The question is then; when does a method become too long? This is not easily discernible, but according to Fowler et. al.(1999)[10]; "The key is the semantic distance between the method name and the method body". If the method name says that a method is a simple get or set method, then that method should not contain several lines of code calculating something else just because that functionality is also needed. This should also be a reminder to software developers out there that naming a method properly is just as important as the functionality that the method contains.

3.1.6 Contrived Complexity

Sometimes, people choose overly complicated solutions for a specific problem. This is also true for software developers. By designing a too complicated architecture, there could easily be more drawbacks than advantages. Some developers, for instance, tend to emphasize too much on aesthetically pleasing design that could compromise performance, usability or other attributes. Designing the architecture overly complicated, could also have other negative impacts. The reuseablility could be worsened, as only the designer would know how to properly reuse the code. To rely on a few experts, in deciphering the code, is not a good idea. People are, as any other resource in a business environment, limited. People change jobs, get ill, retires or dies. Imagine a software developer, that has designed an overly complex architecture for a part of a program that he is responsible of. If another developer had to get to know this architecture it would require several days. Then, the developer responsible for the architecture suddenly dies. Apart from the apparent psychological trauma that the staff of the company will endure, the company will also

be confronted with unnecessary work costs. By having ignored this threat of contrived complexity, the company finds themselves in a lose-lose situation.

3.1.7 Refused Bequest

Refused Bequest is when a class inherits from another class even though it doesn't need most of the methods that it inherits. This is typically a sign of bad design as a developer, instead, the developer should probably introduce a sibling class which it can delegate some of the methods from the superclass to. By doing this, the methods from the superclass is extracted and the subclass gets rid of this unwanted functionality.

```
class ChildClass : SomeInterface{
        public void MethodA(){ }
        public void MethodB(){ }
        public void MethodC(){
            Console.Write("This␣method␣is␣implemented");
        }
        public void MethodD(){ }
```

Figure 3.3: Example of refused bequest

In figure 3.3 the class ChildClass implements the interface SomeInterface which contains the definition of four different methods. However, the only method that actually is implemented inside ChildClass is MethodC(). The three other methods are not implemented. In this example the ChildClass should probably not implement SomeInterface, but another interface that defines only the single method that it actually needs.

3.2 Dependency Cycles

Dependency cycles occur when you have several different classes that depends on each other in a circular manner. This problem, unlike most code smells, are not easily solved. In some cases all that is needed is to introduce an abstraction or a generalized class (interface/abstract class or super class) between the two components involved in the cycle, other times you may have to rearrange the entire architecture.

Dependency cycles between classes or packages can be a sign of bad design decisions, and should be analyzed. In order to properly identify where our main contribution to the study of refactoring code smells will be, we will have to look at previous literature. I have looked into several existing tools that detects dependency cycles. As of yet, I have not been able to find a tool that also can automatically refactor these cycles.

3.2.1 Detecting Dependency Cycles

In order to detect dependency cycles we must analyze the software components. Obvious dependencies like hierarchical dependencies, inheritance and implementations are easy to analyze. By using reflection we can analyze assemblies and gather metadata about the binaries. Looking at the metadata that says anything about their base class and interface types will reveal what they hierarchically depend on. If there are several layers of inheritance we just run a simple recursive method. Most software components also doesn't have many layers, which makes the complexity of the recursive method manageable. However, Classes can depend on each other in many other ways as well. A class will also depend on all the different member variable types that it contains. This is also information which is easily acquired through reflection. A third possible way that a class can depend on others is through methods. Local variables in methods and functions. If the references are all stated in the method head, either as one of the parameter types or the return type, we can identify them via reflection. However if there exists references to other types within the method body, that are only used locally in this method, then we would not be able to identify them by reflection. Reflection can not reveal the contents of a method, just the method head. Because of this, I am using a parser to parse the source code, which is explained in section 3.5.

3.2.2 Breaking Dependency Cycles

Breaking dependency cycles can often be tricky. To break a dependency between two components you have to remove the corresponding fields from one of the components. These fields can either be member fields, method parameters or the interface the component implements. The tricky part is then how to remove these fields or parameters while still preserving the behaviour of the component. One solution would be to introduce an intermediate component, that can serve as a controller class. Another solution, and a question that developers should ask themselves often, is whether fields and methods are placed in the correct component. A member field, and its corresponding usage in a component, could be moved to another component and the dependency cycles could be broken this way. It is a win-win situation, the method and fields are placed in a more suited component and the dependency cycle is broken. However, more often than not, this is not the case. In order to fully break a cycle several fields and methods would be required to move, and some components may have to be deleted altogether. When i researched other existing tools regarding dependency cycles, I found none that could automatically refactor these cycles. This is not a coincidence. Not only is it difficult to come up with an algorithm that can break a cycle, it is also very likely that it will lead to errors or unwanted changes in the architecture.

Lets look at an example of a dependency cycle. Say we have four classes, A, B, C and D. B extends A so it depends on A. C has a private variable a, of type A, so C also depends on A. D on the other hand extends C, and thereby depends on C. If we now say that class A has a variable of type D, then A depends on D. Now, since D depends on C, which in turn depends on A, which again has a dependency to D, then we have a cycle. A $\rightarrow$ D $\rightarrow$ C $\rightarrow$ A. There is of course many ways to break this small cycle. We could introduce an

intermediate class between A and D which both depends on, which does not depend on neither A or D.

3.3 Refactoring

As discussed in earlier sections, refactoring is an essential tool in software programming. Basically, refactoring is the process of changing the structure of software without changing the functionality. Incentives for refactoring can be anything from reusability to performance, it depends on the structure that should be refactored.

How then, can we solve these code smells? By refactoring of course! There have been a lot of research into refactoring and several books have been written about refactoring. A popular book which I also have consulted when writing this paper is *Refactoring: Improving the Design of Existing Code*, by Fowler et. al.[10]. Thanks to similar books and research, we now have a handful of refactoring tools that can help us in solving these software problems. In the next subsections I will be discussing the most common refactoring methods and how we can apply them. The last section will consist of benefits and challenges when refactoring.

3.3.1 Naming Conventions

The most basic and one of most important refactoring techniques is simply the matter of giving methods, variables etc. a correct name. Easily forgotten and neglected by software developers everywhere, this is extremely important when writing software in large software systems. A good practice is to read through each method when it's finished and verify that it actually does what it's method name implies. Additionally, developers should remember that a method name should describe **what** the method does, not **how** it does it. With a very describing name for a method you could also eliminate the need to comment the method. However, if the method name doesn't follow these rules it should be renamed. If there already exists a method with a very similar name, then you most likely have a case of duplicated code, which I have explained in a previous section. On the other hand, if the method can't be given a sufficiently describing name, then it probably is too long.

3.3.2 Extract Method

A very common refactoring technique is to extract a method. As stated in the previous sections where I described several code smells, this technique could be used for many different things. Most obvious is the case of a long method. A long method, most likely, has functionality that can, and should be, divided into smaller methods. Why should we do this? First of all, a long method could be very difficult to understand for other developers. Two important terms in software architecture is coupling and cohesion. Every software developer should know the sentence: ”Reduce coupling, increase cohesion” by heart.

3.3.3 Move Method

Unlike extract method, move method, as the name implies, moves the method in its entirety, rather than extracting parts of it. This would be needed when a method is clearly placed in the wrong class, when classes are highly coupled or when confronted with a god class. A more interesting perspective is when to move a method rather than extracting it and vice versa. The simple answer is to look at the methods references. If the method only uses one external reference, then it would presumably be better to just move it to the referenced class. On the other hand, if the method references several other methods, you could choose to preserve the main functionality in a central place, like a controller class, and extract the referenced parts to their respective classes.

3.3.4 Generalize Type

The idea behind generalizing types, is to draw on the benefits of object oriented code. When creating a new reference or adding a class to the list of parameters for a method, you should consider generalizing the type. BY generalizing we can ensure better reusability and most probably you will have to write less code by doing so.

3.3.5 Benefits and Challenges Regarding Refactoring

Refactoring, as previously stated, is defined as changing the structure while preserving the semantics of a given software system. Several studies have also shown that refactoring decreases the total number of defects. However, according to Zimmermann et. al.(2012)[17] this is not necessarily a view shared by everyone. In one survey they found that: ”participants perceived that refactoring involves substantial cost and risks...”. Obviously, there are also challenges when refactoring. A study by Bavota et al.(2012)[3] shows that while refactorings in general are unlikely to cause bugs(15%), certain refactoring techniques are very likely to induce bugs. Specifically, the *Pull up Method* and *Extract Subclass* refactoring techniques had a likelihood of around 40% of inducing bugs in the source code.

Another study, by Emerson et. al.(2008)[22], discovered that nearly all their test subjects experienced errors when given the task of extracting certain methods in a select few open source projects. This clearly indicates, as their study also showed, that using refactoring tools is not a trivial task. They found that 9 out of 11 test subjects, experienced at least one error while extracting code from selected open source projects. The most prevalent error messages the developers ran into was regarding syntactical selection. They then developed three different tools to help developers with refactoring decisions. One of the tools, which is called Refactoring Annotations, visualizes extract method operations, by assigning each variable a color and highlighting occurrences. This tool improved the diagnosis of precondition violations with about 85 percent. Box view and Selection Assist, which are the two other tools, reduced code selection errors with 95 and 84 percent respectively. This study shows that with a few simple improvements to refactoring techniques, errors are much less likely to occur, which would increase productivity and user satisfaction.

3.4 Existing Tools

In the following subsections I will present my findings on existing tools, especially regarding structural complexities, like dependency cycles. I have read through the documentation of every tool and compiled a list of features for every tool. At the end of this section, I will present a table summarizing their functionalities, supported programming language etc. This section will serve as a justification for creating a tool instead of using a preexisting tool. What we are looking for in these tools are in general terms: A visual representation of dependencies. The tool should show dependencies on both file- and class-level. The tool should be able to simulate the refactoring of structural complexities. This prestudy was performed between august 2013 and November 2013, so the tools could have been updated in the mean time, which of course means that the results in this section could be outdated.

3.4.1 JooJ

A plugin for eclipse that gives real-time feedback concerning design choices[20]. The feedbacks are calculated by examining dependencies between packages. JooJ offers Real-time analyzing of source code and gives you immediate feedback if something is amiss. JooJ also supports visual representation by dependency graphs. This enables JooJ to discover cyclic dependencies and report this to the user. However, JooJ does not support any additional refactoring choices and does not concern itself with other strategies such as static field inlining or singleton registers.

3.4.2 STAN

STAN is a program for Java developers to analyze the structure of their source code[13]. STAN can discover several code smells, dependency cycles in particular. After STAN has analyzed the source code, it can perform queries on one of the many metrics that it calculates. Metrics like simple counting of classes, lines, etc. to more advanced metrics like average component dependency and cyclomatic complexity. STAN can also generate a report based on its findings. STAN, however, does not provide refactoring simulations based on the structure analysis.

3.4.3 JDepend

JDepend traverses java class files and directories to find dependencies[7]. JDepend is free to use but analyzes source code in batch-style, which is time consuming and inconvenient. There also does not exist a plugin or add-on to any IDE which would have been useful(Jdepend4Eclipse exists, but is not developed by the same people). JDepend does not feature any graphical representation of its findings, like dependency graphs or matrices. When analyzing, JDepend stops once any cycle is found, so several other cycles could be present, but JDepend doesn't report them before you have fixed the first one and run the

tool one more time. Additionally, JDepend does not feature any kind of refactoring or refactoring simulations based on the source code.

3.4.4 NDepend

NDepend is a dependency tool for developers using .NET[29]. Similar software for java and C++ exists; javadepend and cppdepend respectively. NDepend features code-query which gives the user a more powerful and customized tool. NDepend also supports several different ways to graphically represent the code architecture.There is also no specialized tool to detect dependency cycles, you will have to do this via code query. Code query however, will demand greater knowledge from the user (commands, syntax), and NDepend costs 400 USD for one license.

3.4.5 JDeodorant

JDeodorant is a plugin for Eclipse[30]. It analyzes Java source code to find many different types of code smells. JDeodorant does not specialize in finding dependency cycles. JDeodorant can discover and resolve the following code smells: Feature envy, type checking, long methods and god class. JDeodorant offers IDE implementation, with tooltips and a user-friendly interface. JDeodorant also supports refactoring of similar code(clones). On the other hand, JDeodorant can't discover dependency cycles, so it is only a tool that can fix common code smells.

3.4.6 Classycle

Classycle is a tool that analyses java .class files to find dependency cycles[8]. Based on it's findings, it will generate a report with a list of the dependencies and metrics in XML format. However, Classycle does not support any kind of graphical representation of cycles.

3.4.7 Nitriq

Nitriq is a code review tool for .NET that analyzes assembly code[14]. Through customized code query almost any sort of code smell can be detected. As with, NDepend, code query gives the user a more powerful and customized tool. However, it also demands greater knowledge from the user in terms of commands and syntax. As opposed to NDepend though, Nitriq is free to use. Nitriq does not support any automatic refactorisation of code smell, it just reports them. There is also no specialized tool to detect dependency cycles, you will have to do this via code query.

3.4.8 Structure101

Structure101 is a powerful tool to visualize and resolve problems regarding the architecture of software[4]. A software architect can design the software with the Architecture

Development Environment(ADE). The developers can then view the architecture and action lists from the ADE via a plugin to their IDE. Structure101 supports several different programming languages, like Java, .NET, and C#. It also supports IDE implementation with many of the most popular IDEs for the supported languages. Structure101 can only resolve certain code smells and is not free to use.

3.4.9 CodePro AnalytiX

A tool developed by Google for Java developers compatible with Eclipse[15]. It is separated into several different tools, so it includes many features, such as dependency analysis. CodePro Analytix can discover and refactor several code smells, and bad code convention like naming, empty catch clause. It supports visual representation by dependency graphs. IDE-implementation with Eclipse makes it easy to use. Generates customized reports, with a host of different metrics to analyze your code. CodePro Analytix also includes a tool that can automatically create JUnit test cases. As of now, CodePro Analytix only works together with Eclipse. It does not support visual representation by dependency matrices, only graphs. It contains no Real-Time analyzing, and you have to manually select the packages you want to analyze.

3.4.10 Summary

Amongst all the tools that I have studied many can detect structural complexities like dependence cycles. However, none of these tools, as you can see in table 3.3, can refactor structural defects. Most of these tools only contains simple refactoring possibilities like move method or extract interface, and none of them can simulate refactorings based on the architecture of the software that it analyses. As of now, there does not seem to exists any tool that can offer the same refactoring functionality as the plugin which I have developed.

3.5 Lexing, Parsing and Compilers

This tool analyses a solution written in C# and generates a dependency graph from all the different classes in the solution. But how should we analyze the solution? Trying to create a new approach to such a well known, and described problem is redundant, especially considering all third party libraries that exists which are easily implemented. These well known tools are all based on the same principle; compiling. Now, there are certain phases of the compiler that are unnecessary for this tool, and hence are outside the scope of this thesis. However, there are two important phases of the compiler that I will describe; lexing and parsing. In this section I have consulted the ”dragon book” by Aho et al.(2007)[1] for the theory which I describe.

3.5.1 Programming Languages

In order to understand the process of how a computer compiles a program, we should have some understanding of what a programming language is and how it is defined. A

Table 3.1: Dependency Cycle Tools

No.	Tool	Goal	Functionalities	Granularity			Implementation	Language
				Method	Class	Package		
1	JooJ	Prevent cycle, Break cycle	Detects Dependency cycle		x	x	Plugin and Standalone	Java
2	JDepend	Find dependency cycles	1.Find dependency cycles			x	Plugin only	Java
3	NDepend	Find dependency cycles and offer refactoring opportunities to resolve them.	1.Find dependency cycles 2.Code query finds the most suited code to refactor		x	x	Plugin only	.NET
4	Classycle	Find dependency cycles.	1.Find dependency cycles 2.Generate report		x	x	Plugin and Standalone	Java
5	Nitriq	Code review, refactoring method name, attributes, etc.	1.Code query 2.Visualization by treemaps. 3.Code statistics	x	x	x	Standalone only	.NET

Table 3.2: Code Smell Tools

No.	Tool	Goal	Functionalities	Granularity			Implementation	Language
				Method	Class	Package		
1	STAN	Structure analysis and refactoring of several code smells.	1.Structure analysis to discover design flaws 2.Generate metrics to rate the problems. 3.Generate report	x	x	x	Plugin and Standalone	Java
2	Structure 101	Improve software architecture.	1.Build architecture 2.Inform of existing or potential flaws 3.Generate action lists	x	x	x	Plugin and Standalone	Several
3	CodePro AnalytiX	Improve software quality in general.	1.Code analysis 2.List		x	x	Plugin only	Java

programming language is a formal language. Similarly to a natural language, which you are reading right now, a formal language consists of tokens and grammars which defines the structure or the syntax of the language. These tokens can be anything that is defined by the grammar, a number, a class, an operator, etc. The grammar then consists of rules that defines all the different sequences the tokens can appear in, to represent a syntactically correct sentence in the language. Despite these similarities there are striking differences between a natural language and a formal language. A natural language is defined as a language that is not created for a particular purpose, as opposed to a formal language,

Table 3.3: Overview of Tools

Functionalities	Tool								
	Jooj	JDeodorant	STAN	JDepend	NDepend	Classycle	Nitriq	Structure 101	CodePro Analytix
Visualization		x	x		x		x	x	x
Reporting			x			x		x	
Metrics		x	x			x	x	x	x
Detect Dependency cycle	x			x	x	x	x	x	x
Refactor cycle									
Detect other smells		x	x		x		x	x	x
Refactor smells		x	x					x	x
Code query					x		x		
IDE implementation	x			x		x		x	x
Real-time analyzing of code	x	x			x				x

which is a constructed language. Because of this ad hoc behaviour of a natural language, certain unwanted properties could appear. For instance, in English, and almost every other natural language, we have ambiguity. When two completely similar words, or sentences, can mean something different. The meaning of the word or sentence is then derived from the context, the semantic, and not the syntax. In a formal language, a sentence, or a word(token), must have only one possible meaning. When dealing with a programming language, it is easy to see why this is an important feature. If every word or sentence could have several different meanings, then the process of compiling a program would become much more difficult and would require a lot more processing resources. A formal language, on the other hand is easy to analyze because it lacks ambiguity. According to Noah Chomsky's hierarchy of grammars, a programming language can be placed into Type 2 languages or context-free languages, as described by Jiang et al.(2010)[16]. A context-free language is defined as when one sentence, can only have one meaning.

3.5.2 Compilers

When analysing source code we need some way to identify the code and turn it into meaningful data that the computer can understand and manipulate. In order to do this, each programming lanugage needs a compiler. A compiler will convert the code written by developers into zeroz and ones that will represent the program. A compiler consists of several different phases. Among those are lexing and parsing.

3.5.3 Lexing

The first step in the compiling process is lexing. Basically what a lexer does is analyse all the characters written in a source file, and turn them into tokens. Hence why lexers also are called tokenizers. However, a lexer should have some additional features as well which I will discuss later in this section. A tokenizer takes a stream, for instance a text file, and converts it into tokens. The tokenizer does this by following a set of rules for breaking

up tokens. A normal rule would be to break when encountering a space, tab or new line. Given this example:

```
int number = 9989;
```

This example would be broken up into four different tokens: 'string', 's', '=' and '9989;' After the tokenizer has split the stream up into many different tokens, the lexer analyses them and creates lexemes. A lexem is an identification of the type of the token that we found and it's value. So the lexer can take the token '9989;' and identify it as two different lexemes; a number with value of 9989 and an operator identified as a semicolon.

3.5.4 Parsing

Where the lexer can identify different tokens and their values, the parser takes it one step further and analyses the syntactical meaning of these tokens. The syntax of a sentence is defined by the structure of the grammar. If a set of tokens that are parsed does not comply with any of the rules in the grammar, then the sentence is syntactically incorrect. The parser takes the lexemes from the lexer as input and creates an abstract syntax tree. Based on a grammar, the parser looks at all the different lexemes, and more importantly, in which order they come in, to construct the abstract syntax tree.

Part IV

Own Contribution

Chapter 4

Dependency Simulation Plugin

This plugin can be divided into four parts; The plugin itself, the parser, the controller and the model. The first part is the Visual Studio plugin. This part is responsible for implementing the tool into the visual studio environment.

The parser, which takes all the source files in the solution that it is analyzing and parses this into an abstract syntax tree. The tool then recursively finds the nodes of interest concerning dependencies and builds up a graph from this data.

The controller takes care of everything that has to do with data storage, common data structures used within the program and logic concerning different functionality. The model holds all the important data which will be used when visualizing the graph and simulating the dependency refactoring.

4.1 User Interfaces

In this section I will present some of the graphical user interface that is presented to the user, and explain the general view of the plugin.

The main screen of the plugin can be seen in figure 4.1. As you can see, there are several menu items on the top of the tool window, which holds the plugin. The ”Click Action” drop down menu is used when manipulating the graph, and will be further discussed in section 4.5.4. The next item, ”Load Solution” starts the process of parsing through the solution, calculating the dependencies and generating the graph. The first thing that will be shown is loading bar that shows the progress of the plugin. When the loading bar reaches it end, the nodes in the graph will be shown, and the graph will be relayout. The layout of the graph is set according to a predefined algorithm which places the nodes according to which components they are connected to. The two next items ”Undo” and ”Redo” are also connected to graph manipulation, basically they undo and redo a manipulation that has been performed on the graph. ”Generate Mapfiles” generates thirteen different text files in the solution folder under a subfolder called ”SimulationData”. These text files contains

the different maps that this plugin uses in order to simulate a refactoring. These maps are further discussed in section 4.4.1.

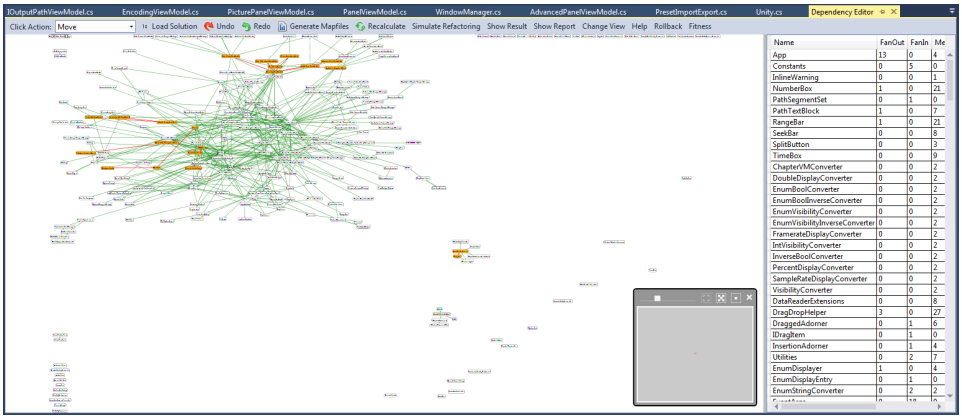
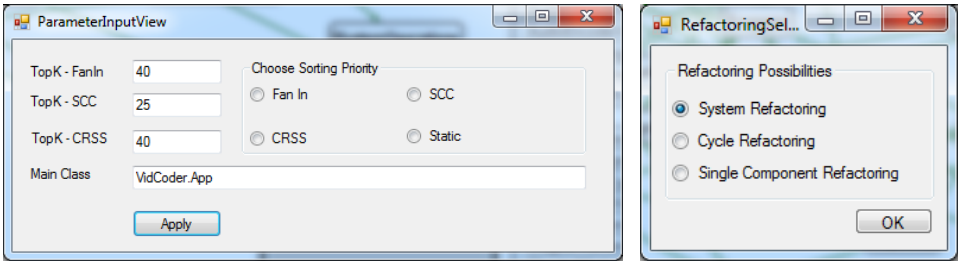


Figure 4.1: The main screen of the plugin

The two figures in figure 4.2b shows two different parts of the simulation process. Figure 4.2a shows the window that will be presented to the user right after clicking on ”Simulate Refactoring”. This view contains several parameters that must be set, for the simulation to work. The simulation are explained more thoroughly in section 4.5.5. After inputing all the needed the parameters a new window will appear. This window is shown in figure 4.2b. This view basically lets the user choose what type of simulation that should be performed.



(a) The parameter input view **(b)** The simulation selection view

Figure 4.2: Two of the simulation views

After selecting a simulation the user will be presented with a list of candidates to choose from. In figure 4.3, you can see the candidates of a system refactoring. This view presents the user a list of classes that should be refactored. In this case, with a system refactoring, the user only needs to choose the number of refactorings that should be performed. With the two other types of refactoring the user must choose which classes to refactor.

When a simulation has been performed the window shown in figure 4.4 will appear.

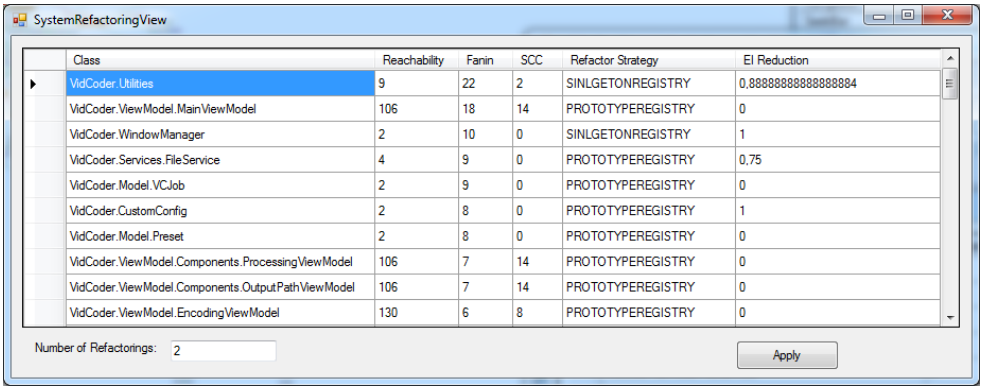


Figure 4.3: The candidate view for a system refactoring

This view includes several metrics that say something about the general quality of the solution being refactored, before and after the simulation. These metrics are explained in section 4.5.5.

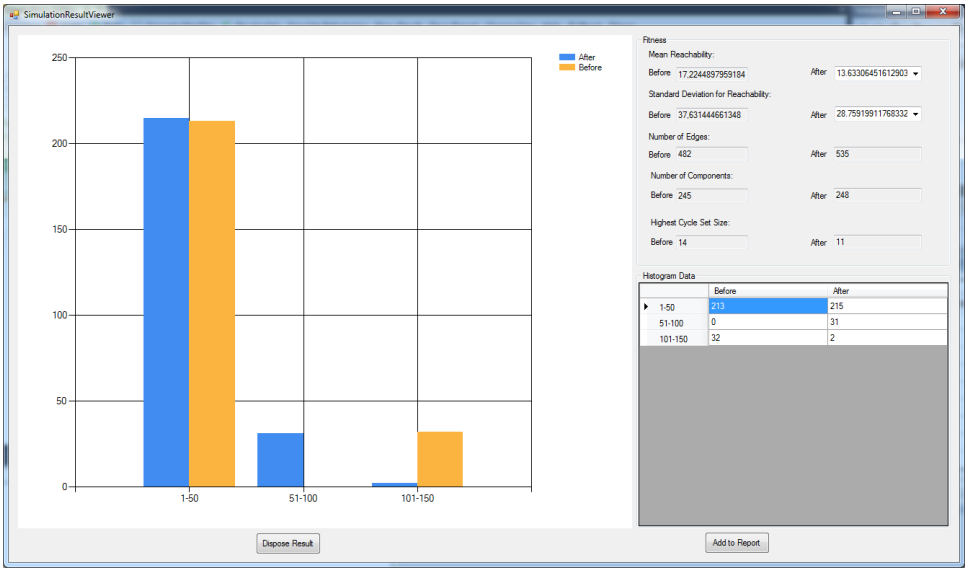


Figure 4.4: The simulation result view

Another important part of the plugin is the list view shown on the right side of the main screen. This list contains every class in the solution and several metrics are displayed for each of them. The metrics are explained in section 4.5.3. A view of the list is shown in figure 4.5.

Name	Namespace	Project	FanOut	FanIn	MethodCount	Fields	Reachability	DependsOnSCC	SccCount
MainViewModel	VidCoder.ViewModel	VidCoder	50	2	32	58	90	2	0
ProcessingViewModel	VidCoder.ViewModel.Components	VidCoder	35	1	26	41	58	3	0
OptionsDialogViewModel	VidCoder.ViewModel	VidCoder	23	1	4	54	48	1	0
PreviewViewModel	VidCoder.ViewModel	VidCoder	20	1	14	37	42	1	0
OutputPathViewModel	VidCoder.ViewModel.Components	VidCoder	17	0	17	8	35	1	0
EncodingViewModel	VidCoder.ViewModel	VidCoder	16	0	4	13	115	6	0
MainWindow	VidCoder.View.MainWindow	VidCoder	14	0	32	6	62	1	0
App	VidCoder.App	VidCoder	13	0	4	2	98	2	0
IEncodingViewModel	VidCoder.ViewModel	VidCoder	12	7	2	0	113	0	8
EncodeJobViewModel	VidCoder.ViewModel	VidCoder	11	0	6	12	24	1	0
PresetsViewModel	VidCoder.ViewModel.Components	VidCoder	11	0	10	5	30	1	0
VideoPanelViewModel	VidCoder.ViewModel	VidCoder	11	2	13	14	113	0	8
Updater	VidCoder.Services	VidCoder	10	1	10	6	27	2	0
QueueTitlesDialogViewModel	VidCoder.ViewModel	VidCoder	10	3	3	12	23	0	2
PresetImportExport	VidCoder.Services	VidCoder	8	0	2	3	23	2	0
PreviewWindow	VidCoder.View.PreviewWindow	VidCoder	8	0	12	3	45	1	0
AudioEncodingViewModel	VidCoder.ViewModel	VidCoder	8	1	10	23	113	0	8
SubtitleDialogViewModel	VidCoder.ViewModel	VidCoder	7	3	12	9	20	0	3
Logger	VidCoder.Services	VidCoder	7	1	10	6	15	1	0
EncodeResultViewModel	VidCoder.ViewModel	VidCoder	7	1	0	9	16	0	0
Database	VidCoder.Model	VidCoder	6	4	5	4	18	0	4
Utilities	VidCoder	VidCoder	6	22	30	9	9	0	2
LogWindow	VidCoder.View.LogWindow	VidCoder	6	0	5	3	7	0	0
ChapterMarkersDialogViewModel	VidCoder.ViewModel	VidCoder	5	1	0	4	17	1	0
RemoteEncodeProxy	VidCoder	VidCoder	5	1	19	19	9	0	2
SubtitleDialog	VidCoder.View.SubtitleDialog	VidCoder	5	0	9	2	23	2	0
EncodeJobsPersist	VidCoder.Model	VidCoder	5	2	3	2	18	0	4
DriveService	VidCoder.Services	VidCoder	5	0	4	2	9	0	0
ProcessAutoPause	VidCoder.Services	VidCoder	5	0	8	9	10	0	0
HandBrakeEncoder	VidCoderWorker	VidCoderWorker	5	1	9	4	6	0	2
Presets	VidCoder.Model	VidCoder	5	4	16	7	18	0	4
AudioPanelViewModel	VidCoder.ViewModel	VidCoder	5	3	9	5	113	0	8

Figure 4.5: The list view containing all the classes

4.2 The Parser

This tool uses a parser in order to find the actual dependencies within a C# solution. More specifically, this tool uses a third party library called NRefactory to parse the source files. NRefactory can parse C# solutions, and generates an abstract syntax tree with pattern matching support.

A parser has, of course, some limitations. A parser will not necessarily correctly identify class names, namespaces, return types etc. inside a class. A compiler on the other hand would know this, as the compiler must build the program and load symbols and classes where it is needed. A compiler does indeed include a Parser among other components. Where the compiler contains a lexer, a parser, semantic analysis, an optimizer and a code generator, the parser only contains the lexer in addition to the parser itself.

```
using System.Collections.Generic;

public class Namespace
{
        private List<Class> _classes;
}
```

Given the scenario above, where does the List<T> class comes from? We know that it comes from the namespace System.Collections.Generic as stated in the using statements. A compiler would also gather as much. But a parser would have no idea where this symbol comes from. It would simply state that it found a variable declaration statement where the identifier is ”List” with some type arguments. So in order to fully identify all symbols when using a parser, we must identify what scope we are in. A .cs file may have hundreds of using statements, which in turn can include hundreds of symbols. A compiler resolves this by using a symbol table[Reference]. A symbol table contains a record for each identifier what type it is and other attribute fields. In my tool I have solved this by creating my own symbol table. When I find a new class I store it’s full name, the namespace and the source file it was found in. Then when I find a reference to this class I first simply try to find it based on the identifier given by the parser. In most cases, however, this is not sufficient. Consider the Following scenario with two different source files:

common.cs:

```
namespace Common.Data
{
        public class Class
        {
        }
}
```

namespace.cs:

```
using System.Collections.Generic;
using Common.Data;

public class Namespace
{
        private List<Class> _classes;
}
```

When parsing namespace.cs we find the symbol ”Class”. The compiler knows that this symbol is loaded from the namespace Common.Data, but the Parser can’t know this. Therefore we have to manually find it ourselves. Because this tool parses C# we are faced with an additional challenge in the form of the logical structuring that is used in C#. In C# namespaces are used in a logical way as opposed to Java and it’s physical structuring based on packages. A namespace in C# can exist in several class files regardless of it’s location. My solution to this problem is to create data structures to hold information about each class file, it’s using statements, the namespaces and the classes declared in the class file.

We need this information because there are several ways that a symbol could be loaded. It could be located in the same namespace in the same class file, which is the most straight forward example. Another more complex example could be to reference a symbol inside a different source file via a using statement.

Although a compiler would solve all the issues concerned with just using a parser, there are benefits of using a parser. Given the requirements and purpose of this tool we can make some simplifications concerning code analysis. First off, this tool should only analyze the components within a single solution file. If it finds symbols that does not exist in this scope, it should disregard these symbols. The only thing that is interesting to look at is the internal structure of the current solution being analyzed. Components that do not have any impact on the structure of the solution being analyzed, should be analyzed separately. So, to use a compiler in this tool would actually be a little overkill. This tool does not analyze third party libraries or other external symbols. A compiler would try to load all external symbols in order to generate a complete symbol table, a parser would not try to load any external libraries. A parser is context free, and does not know where the symbols that it parses comes from, it can only identify they are symbols, operators, numbers etc.

4.3 The Visual Studio Plugin

This program is a plugin for Visual Studio or more specifically, a VSPackage. VS Packages was introduced with Visual Studio 2010 and is slightly different than the old implementation of a plugin, a Visual Studio Add-In. In general, a VS Package was built to further integrate software within the Visual Studio environment. VS Packages are called ”first class citizens” in the Visual Studio environment, and a lot of functionality is added into Visual Studio by these packages. Add-Ins are more limited in their functionality, and are restricted to certain object models. A more detailed explanation between a VSPackage and an Add-In can be found on [21]. This particular plugin is integrated into the Visual Studio environment as a tool window. This means that the plugin is contained inside the same type of window that will contain source code when writing code in Visual Studio. This enables the user to drag and drop the window, wherever the user wants it to be, within the Visual Studio environment.

4.4 Analysis tool

One of the most important parts of this plugin is of course the ability to generate suitable candidates and simulate a refactoring on these candidates. This is where the analysis tool comes in. This analysis tool is written in Java and is developed by Tosin Daniel Oyetoyan[[26]]. The tool consists of several different refactoring strategies that can be simulated based on the dependencies found in the source code that is analyzed.

4.4.1 Data Structures

In order to simulate refactorings, the program needs several maps that is needed for gathering information about dependencies between the components. These maps are sent to the analysis tool as parameters when the program needs to generate candidates for the simulation, and when a simulation should be performed. These maps are also displayed in the class diagram in figure 6.3. Most of these maps are a mapping from string to a Set of string (Map<string, Set<string>>). Where the string is the textual representation of a given class and the Set contains all the classes, within the same solution, that occurs as the type of dependency that the map represents. the data structures that is used are Map and Set from the java.utility package. These classes are, of course, not accessible from a default C# project. You will need a reference to the IKVM.OpenJDK.Core and IKVM.OpenJDK.Util libraries. The IKVM library are discussed in section 4.4.2.

Maps

The following maps are generated by the plugin and then used by the analysis tool to simulate refactorings. All these maps represent different types of dependencies that can occur between two classes in the same solution.

All Types: Simply a map that includes all the classes that a given class depends on.
Field Usage: This map contains a few different things. Naturally every variable declaration will be present, as well as variable initializers of a given class. Additionally, if a member inside a referenced class is being used, the class containing the member must be included in this map, as seen in the code example below. In the example, ClassB is not declared as a variable, nor initialized, but is defined as a method parameter. However, it will still be referenced in the Field Usage Map because a member inside ClassB is referenced inside ClassA.

```
public void MethodInsideClassA(ClassB b)
{
        b._member;
}
```

Static Field Usage: This map includes every class that has a static member which is accessed inside the given class
Method Usage: Every member inside a class which has a method called from its object instance, is added to this map. Also every method parameter and method return type is added.
Static Method Invocation: For all methods that are called statically, the class containing the method is added to the map.
Constructor Invocation: Straight forward, every Class that is initialized by a constructor invocation is added to this map.
Published Types: If a method return type or parameter is found in a public method, it is added to this map. This is needed for interface extraction.
Class Types: This map include, for every class in the solution, what type the class is. Either ”I” for interface, ”A” for abstract or ”C” for a regular class.

Method Types: Every method inside a class is either static or nonstatic. This map states whether the class includes only static, only nonstatic or both static and nonstatic methods.
Method Size: A simple count of how many methods that exists within a class.
Base Types: Every class that a given class inherits from, that is not an interface, is declared in this map.
Interface Types: Every interface that the class implements are listed in this map.
Super Types: Combines the base and interface maps.

4.4.2 Refactoring Strategies

The refactoring strategies used by the the tool can all be used in the three different types of refactoring simulations that can be performed. The strategies range from simple movement of fields from one class to another, to more advanced registries with entries for object instances that should be accessed.

Extract Interface: The first refactoring strategy that is used is extract interface. When refactoring a class using this strategy you extract all the public methods and properties into a new interface. After extracting these methods, you first need to implement the interface in the original class. If the original class implemented another interface, the new interface must inherit from this interface. However, in some cases the class that should be refactored inherits from another class. If we have that scenario, we must introduce an abstract class instead of an interface. An example of this is explained in section 7.1. In figure 4.6 you can see an example of an interface extraction performed on the class shown in figure 4.7

```
public interface IA
{
        public static readonly string ID = "Namespace.A";
        public abstract void MA1(C c);
        public abstract List<D> MA2();
        public abstract int MA3();
}
```

Figure 4.6: The interface of class A

Static Method Inlining: When a class depends on another class via a static method invocation, we can use this refactoring strategy to remove the dependency. This strategy basically moves the static method from the target class to the source class. Then reverses the dependency from the source to the target. This is visualized in figure 4.8. However, this strategy does not work under certain circumstances because it doesn't remove the dependency between the two components, but reverts it. Additionally, the decision to move a static method should not be made easily, and it should always be the developers that makes the final decision, independent of the simulation. For example, it would be a bad

```
public class A : IA
{
        public static int _number;
        public override void MA1(C c){ }
        public override List<D> MA2(){ return new List<D>(); }
        public override int MA3(){ return _number; }
}
```

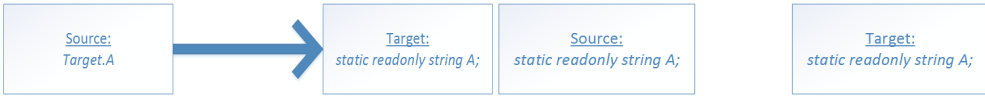
Figure 4.7: Class A

idea to move a static method inside a utility class, to a class that depends on it. In such a scenario, you would also have to move all the other dependencies, that exists through this static method invocation, in other classes. Because of this, the most appropriate scenarios to perform this refactoring strategy on, would be where just a single, or a select few classes uses the static method that should be moved.

(a) Before performing a static method inlining **(b)** After performing a static method inlining

Figure 4.8: Static Method Inlining

Static Readonly Field Inlining: As with static method inlining, this refactoring strategy revolves around moving the object of dependency. In this case though, the object that should be moved is a static readonly field. Because of this, we can simply copy the field because it cannot be changed. In figure 4.9 there is an example of this strategy. Observe that the dependency between the two classes has been broken, contrary to the case as in static method inlining, where the dependency is reversed.



(a) Before performing a static field inlining **(b)** After performing a static field inlining

Figure 4.9: Static Readonly Field Inlining

Singleton Registry: This refactoring strategy should be used when a dependency to a singleton class should be removed. This strategy combines the creation of a registry class with the extract interface strategy already discussed. First, you should create a new class to hold the singleton instances of all the singleton classes. This class in itself will be a singleton, and will contain a registry, for instance a Dictionary<string, object>. However, when interacting and getting instances from this singleton registry, we use the interface

of the singleton class. By doing this we move the dependency towards the singleton class from the classes that use it, to the interface of the class. Note that the interface should contain a static readonly string that should be unique that is used to fetch the class instance of the implementation of that interface. There should be only one class that implements this newly created interface, as you can only store one class instance for each interface that is registered. In figure 4.10 a visual explanation of this strategy can be seen, and in figure 4.11 an example of this strategy is presented. In the diagram in figure 4.10 the observant reader will note that a Main component is present. This represents the entry point of the solution that should be analyzed. This is a limitation with the tool and will be further discussed in section 4.7.

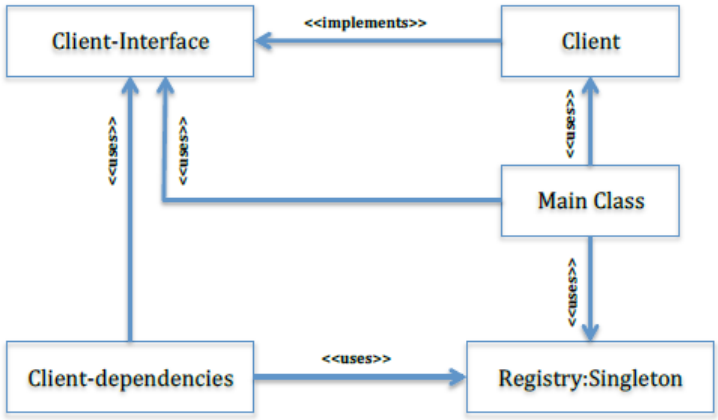


Figure 4.10: Singleton Registry, Oyetoyan et al. (2014)[26]

```
public class ADepend
{
        private SingletonRegistry _registry;
        public int M(){
                registry = SingletonRegistry.GetInstance();
                IA a = registry.GetClassInstance(IA.ID);
                return a.MA3();
        }
}
```

Figure 4.11: Usage of a singleton registry (continuation of the code found in figures 4.6 and 4.7).

Prototype Registry: Similarly to the singleton registry, this is strategy consists of creating a new class to hold class instances, and interface extraction of the class that should be refactored. As with the singleton registry this prototype registry class will only contain a single instance of each class, and the registry itself will be a singleton class. The catch

however, is that the interfaces of the different classes that should be contained in the registry, must define a cloning method. This method should basically return a new instance if the same class that implements the interface. This is easily achieved in C# by implementing the ICloneable interface. In figure 4.12 a diagram of the prototype registry is shown, and in figure 4.13 there is an example that shows how to use this strategy.

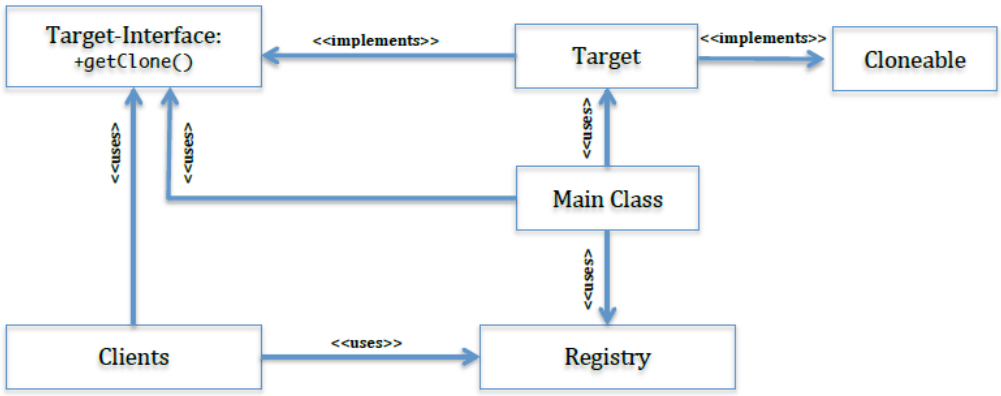


Figure 4.12: Prototype Registry, Oyetoyan et al. (2014)[26]

```
public class ADepend
{
        public void M2(){
                IA a = registry.GetClassInstance(IA.ID);
                //instead of using -> A newA = new A(); we use:
                IA newA = a.Clone();
        }
}
```

Figure 4.13: Usage of a prototype registry.

Integration with .NET

The analysis tool, is as mentioned earlier, developed in Java, which introduces some difficulties considering that this plugin is written in C# with Microsoft Visual Studio integration. In order to integrate a Java library in .NET environment the assemblies needs to be converted into a dynamic link library or an executable. There are a few libraries that offers this feature of converting .jar files into .dll. For this tool I have chosen the IKVM.NET library. IKVM.NET is an implementation of Java for Mono and the Microsoft .NET framework. In order to run compiled Java code on Microsoft .NET there are certain components

that needs to be in place. IKVM.NET includes four different components. A virtual machine for Java implemented in .NET, a .NET implementation of the class libraries that Java contains, a tool to translate bytecode in order to convert .dll's to .jar's and vice versa, and finally a tool that can support interoperability between Java and .NET. In IKVM.NET the Java bytecode is translated to Common Intermediate Language. A more detailed look at IKVM.NET is available at their website[11].

4.5 Features of the plugin

The plugin has several features concerning source code analysis. From visualizing the dependencies in a graph and manipulating these nodes, to simulating refactorings, the most important features will be discussed in the following sections.

4.5.1 Visualizing a C# solution

One of the primary and most important features of this tool is of course the ability to visualize the source code of a C# solution in a graph. The visualizing of a dependency graph is made possible through a third party library called GraphSharp. This is an open source project, licensed under the Apache v.2 license. The source code and binaries are available at [5]. The nodes in the graph each represents a single class from the solution. A special case to note here is a partial class which can be declared several places, with different dependencies. All the different declarations and dependencies of a partial class has been combined into one node. For every dependency that is found inside a class an edge will be drawn between the corresponding nodes of the given class and the class it depends on. All these edges can contain several different dependencies. These different dependencies are defined by an enum(see figure 6.3). The plugin also features a method to determine whether nodes are in a cycle. The nodes that are inside a cycle will be colored orange in the graph, to differentiate between these nodes and the nodes that are not in a cycle. Edges can also be cyclic. If two classes both depends on one another, the edge between them is cyclic. This will be shown in the graph as edges that are colored red.

4.5.2 Miscallenous features

There are some additional features in the tool that I also would like to present. First, there is a possibility of right clicking on a node, or an item in the list view in the plugin and choose "View Code". This feature is basically the same as choosing "go to definition" inside the Visual Studio environment. Since the nodes in the graph and the items in the list view all represents classes, this feature will open up a window in Visual Studio with the source file that contains the definition of the class that the user chose.

There are also some features linked to the list view that contains all the classes. This list view and the metrics that it contains are described next, in section 4.5.3. This list view contains 21 different metrics. To show all of them could be overwhelming and confusing, so I have introduced a feature to show and hide these metrics. This can be done by right

clicking on one of the headers in the list view, displaying one of the metrics. When right clicking here, a context menu will appear with all the different metrics listed in the menu. Besides each metric in the menu there is a check box that shows whether this metric is shown or hidden in the list view. By clicking on one of the metrics in this menu, the user reverts it's status. If it was hidden, it will now be shown, and vice versa. Another feature is the ability to sort by these different metrics. By simply double clicking on one the metrics the user will sort the list by either ascending or descending order. This is an easy way to identify which classes has the highest reachability, number of methods etc.

4.5.3 Metrics

When analyzing a C# solution the tool gather and stores several class specific metrics from the source code. These metrics are useful to quickly identify certain characteristics of a class. For instance, if a class has only static methods, the class is most likely a utility class and will most likely depend on none or few other classes, but many classes will depend on it. This and many other useful metrics are datamined from the source code and is presented in the list view shown in figure 4.5. Following is a list of all the metrics for each class.

Name: A self explanatory metric that describes the name of the Class. The name is only the identifier declared in the class declaration.
Namespace: Every class in C# should have a namespace which should describe the logical structuring of the code. The namespace and the name of each class is combined into the unique name identifier of a class.
Project: Every class must also exist inside a project. There can be several projects in a C# solution, and knowing which project each class belong to can make it easier for developers to move them depending on which classes they depend on.
Fan in: A metric that reflects how many other classes that depends on this particular class.
Fan out: As opposed to Fan in, this metric reflects how many other classes the given class depends on.
Reachability(CRSS): Reachability or Component Reachability Set Size, represents the transitive relationship between classes and the classes that it depends on. If a class directly can reach one other class it could still have a reachability of hundred other classes, depending on the single class that it directly depens on.
SCC count: Strongly Connected Components are classes that depends on each other in a circular manner. This metric shows how big this cycle is. Please note that this size will of course be the same for all components in the same cycle.
Method count: A straight forward metric describing how many methods that is contained in a class.
Static method count: This shows the number of static methods that's inside the class. Will always be a subset of the number of total methods.
Constructors: The number of different constructors found in a class.
Fields: Describing the amount of globally declared fields inside a class. Different from member properties.
Static fields: The amount of static fields in a class.
Private fields: The number of private fields found in a class.

Public static fields: All the fields inside a class that are both public and static will be represented by this metric.
Properties: In C# there is a particular way of encapsulating private fields inside a class; member properties. At first glance these encapsulation methods looks like normal methods with defined access modifiers and return type. However, inside you can only place a get or set bracket followed by some logic. Additionally, you can not define parameters, as there can only be one(for the set method), of the same type as the return type. These properties should not be regarded as normal fields as they usually expose one of these fields, and should therefore be represented in another different metric.
IsInterface: There are several boolean metrics calculated by this tool, IsInterface is one of them. Returns either true or false, on whether the component is an interface or not.
IsEnum: Shows wether or not a component is an enum.
AreAllMethodsStatic: True if all methods inside a class are static.
AreAllFieldsStatic: True only if all member fields located inside a class are static. Useful in locating so called utility classes that only hold static fields that can be accessed from anywhere in the project.
ProjectReferences: A boolean value displaying wether or not the component either depends on or is depended upon by other components in the same project. If this value is false, then the developers should consider moving this particular component to another project.
Depends on SCC: Given a specific component, this metric shows how many of its connected components that are inside a cycle.

4.5.4 Graph manipulation

This plugin has several ways of manipulating the graph. In a large solution, the graph will be huge, minimizing the potential usefulness of this feature, as it's difficult to discern from all the different components and the edges that is connected to it. However, on smaller projects this could be quite a helpful tool to quickly add a node, add some edges and see if the results of the simulation will be any different.

The first feature I would like to discuss is the ability to add a node. This will also introduce a new class in the underlying data structure of the plugin, so the simulation would be updated according to the nodes and edges that are added to GUI. You can remove a node by right clicking on the node and choosing "Remove Node" in the context menu that appears. Another feature is the possibility of adding new edges. To do this the user would have to select "Add Edge" in the drop down list called "Click Action" in the menu. After choosing this action, a user can proceed to click on a node, which will mark this node as the currently selected node, and then click on a second node. This should draw a new edge between the two node. To remove an edge, simply right click and choose "Remove Edge". The model will of course also be updated according to all these different changes in the graph.

4.5.5 Simulating Refactorings

This plugin consists of three different types of refactorings, each modeled after specific problems. These types of refactorings are all performed in the analysis tool presented in section 4.4, and can use all the different refactoring strategies described in that section.

System Refactoring: The overall goal of this type of refactoring is to decouple the whole system. This refactoring needs the user to input the number of refactorings that should be performed. This is because this refactoring is based on iterations, where for each iteration, a new class is selected based on the parameters and the sorting priority that is given, and the previous refactoring that was simulated in the model.
Cycle Refactoring: The whole idea of this refactoring revolves around breaking specific cycles. Cycles, or strongly connected components, are visualized in the graph as orange nodes. These can be refactored by selecting a set of edges between these components and removing them. The algorithm in the analysis tool computes the minimum set of edges that is needed to break the cycle, and report these edges back. In most cases you would also have to introduce some new edges, because of transitive dependencies between the components.
Single Component Refactoring: Given a class with a very high reachability compared to the other classes in the solution, a developer might want to refactor only one class. This is supported in the plugin. The procedure is the same; input some parameters and the plugin will present a list of candidates. In the list, the user must choose which classes that should be refactored.

4.6 Algorithms and data structures

As described in section 4.2, there are several drawbacks when using a parser compared to a compiler, most importantly, that you are not able to identify every symbol that is referenced in the source code. In order to simulate the same behaviour as in a compiler I had to introduce several data structures and algorithms to properly identify every symbol inside a class. In this chapter I will describe and explain the most important data structures and algorithms that I had to introduce in order to overcome these challenges. The first section will introduce the data structures in detail before I explain how these data structures are manipulated and utilized in the algorithms in the following section.

4.6.1 Data structures

In figure 6.3 you can view the structure of the classes that I am going to describe. Each class that is parsed from the source code is represented as either a Class or a PartialClass, both inherits from BaseClass. It's important to note the difference between these two; a partial class can de divided into several different type declarations, in several different source files. So while one declaration of a partial class may be included in a source file which consists of several hundred using statements, another declaration can perhaps have

```
private string value;

public void ParseString(string data,string type){
        switch(type){
                case "double":{
                        double value = double.Parse(data);
                        break;
                }
                case "char":{
                        char value = char.Parse(data);
                        break;
                }
        }
}
```

Figure 4.14: Example of an overriden identifier

zero using statements in it's corresponding source file. Therefore, we need to keep track of every dependency, and the source file that it occurred in. For a normal class this is not necessary as it can only have one source file. When encountering upon a using statement, the string representation of the namespace is added to a list inside the current SourceFile object. Then inside the DependencyCalculator class there is a dictionary of Namespaces which are mapped to the string representation of each namespace.

Another challenge comes in the form of identifiers and their scope. In the example in figure C.1 the identifier ”value” is declared three different places, with different scopes, which is allowed in C#. First, it is declared as a global identifier of type string. Then, inside the method ParseString, it is first defined as a double in the first case, then as a char in the next case. From this example it is easy to see that we need to keep track of these identifiers and what it's currently declared as. If, for instance the parser finds a method invocation on the identifier ”value”, we must know which type that is currently assigned to the identifier and add this type to the references in the class that is currently being parsed. Therefore, in the parser class, DependencyFinder, there are three different dictionaries, which maps an identifier to the type it is associated with. One for the current identifiers, one for the identifiers found in the current method and the last dictionary holds the identifiers that has been overridden. For each new variable declaration that is found, it is added to the current identifiers, and to the method identifiers if it was found inside a method. Then, when a new method declaration will be parsed, each identifier from the method identifiers are removed from the current identifiers and replaced if the overridden identifiers contains a value which is mapped to the same key.

4.6.2 Algorithms

Now that we have the data structures that is necessary we can manipulate the data and figure out the dependencies between the components. First I will describe the algorithm for preserving the correct identifiers and then I will present the algorithm that calculates the dependencies between the components.

Variable identifiers

When encountering a new variable of a certain type, I need to store this identifier to know if a method invocation, variable reference etc. has occurred on this identifier. This algorithm adds the identifier to a Dictionary where the name of the identifier is the key, while the name of the variable type is the value. First, I simply check if the map already contains a definition for the identifier name. If not, then I add the identifier to the dictionary. If I find a new variable inside a method, that overrides one of the global identifiers, I have to save the previous value of that identifier, so that I can retrieve and set the identifier to its old value when I have parsed through the method. When I have parsed through a method, I remove all the key pairs that exists in the method identifiers and remove them from the current identifiers. This is to ensure that a global variable will be added correctly if it's defined after a method has declared a variable with the same identifier name. The algorithm is presented in figure C.1 in Appendix C.

Calculation of dependencies

The entire algorithm is presented in Appendix C. The algorithm is divided into four methods. The first Method, CalculateDependencies(), loops through every dependency in a class and calls the method FindDependency() for all the dependencies that was found by the parser. This FindDependency() method then performs a number of checks to find the full class name. First, we check whether the class is in the same namespace as the class that were analyzing. If not we check if the class is contained in one of the namespaces defined by the using statements in the sourcefile. If the class is a partial class, then we must choose the source file that this dependency was found in. If we still can't find the class, we check if the class is found in the global namespace. If none of these checks are true, then the class does not exist in the solution that we're parsing, and we can discard it. The two last methods are simply helper methods that are reused inside the FindDependency() method. These are only defined because of reusability.

4.7 Limitations

This tool is by no means perfect, and this section will explain and acknowledge the known limitations to the plugin.

As mentioned in section 4.4.2 this plugin needs and entry point to be able to correctly refactor some of the dependencies in a solution. Because of this, the plugin will not be able to correctly generate refactorings that can be used in a .dll or a class library for instance.

The reason for this is because the instances of the classes that should be refactored using either the singleton registry or prototype registry strategy, must be added when the application starts. This is to make sure that these instances actually exists when we need them from the registry. Obviously, a class library is completely independent of the other code in the library and can't be executed, so these solution projects can not be refactored this way. However, Microsoft has a similar registry called Unity, already implemented in its libraries. Unity include features to resolve instances that are stored in the registry, which may solve this issue. However, I have not been able to thoroughly experiment with Unity, so I can't say for certain that it will solve this problem.

Another limitation to this plugin is that it can't analyze dependencies through an implicitly typed variable. An implicitly typed variable is a relatively new feature in C# as it was introduced in C# 3.0. These variables are only defined as *var* and will be typed depending on what the right hand side of the expression that initializes the variable. The right hand side can of course be anything that could be defined as a type, whether that be a simple number, or the return type of a method inside another class. In figure 4.15 there is an example of an implicitly typed variable. In this example, SomeClass should contain, given the maps described in section 4.4.1, a field usage and a method invocation reference to ClassA after analyzing the source code. However, given that I have not introduced sufficient data structures and algorithms in the plugin, this is not recognized as of now.

```
public class SomeClass
{
        public void Method()
        {
                var implicitVariable = new ClassA();
                implicitVariable.SomeMethodInsideClassA();
        }
}
```

Figure 4.15: An example of an implicitly typed variable.

Chapter 5

Requirements

In this chapter I will present the different requirements and scenarios that I have identified for the plugin. These will act as guidelines when developing the plugin and should also reflect the most important features that should be present in the plugin. I have divided the functional requirements into two different parts. The first will present the functional requirements of the plugin, while the other contains the functional requirements for the user interaction.

5.1 Functional Requirements

The functional requirements in a software system should describe the functionality of the software system. The functional requirements are presented in table 5.1 and 5.2

Table 5.1: Functional requirements for the plugin

The Plugin		
ID	Description	Priority
FR1	The parser should be able to analyze C# solutions	High
FR2	The plugin should give the user feedback on the loading process	Medium
FR3	The plugin should be able to visualize a solution.	High
FR4	The plugin should be able to detect and visualize dependency cycles	High
FR5	The plugin should be able to show all the classes in a solution, in a list view that contains important metrics for a class	Medium
FR6	The plugin should be able to simulate refactorings	High
FR7	The plugin should show the result of a refactoring simulation	High
FR8	The plugin should be able to generate a report based on the simulation	High
FR9	The plugin should be able to show a class-view and a file-view	Medium

Table 5.2: Functional requirements for the view

User Interaction		
ID	Description	Priority
FR10	The user should be able to add a new class to the graph	Medium
FR11	The user should be able to add an edge between two nodes in the graph	Medium
FR12	The user should be able to remove a node from the graph	Medium
FR13	The user should be able to remove an edge from the graph	Medium
FR14	The user should be able to sort the classes in the list view, based on each metric	Medium
FR15	The user should be able to show and hide different metrics in the list view	Medium
FR16	The user should be able to set a class as the main class in the graph	High
FR17	The user should be able to input parameters for a refactoring simulation	High
FR18	The user should be able to choose between three diferent types of simulations: System refactoring, cycle refactoring and single component refactoring.	High

5.2 Quality Requirements

Quality requirements are non-functional requirements in a software system, and can be described by quality attribute scenarios. There are several quality attributes that has been identified as important for software quality. These can range from how well the software performs, to how well the source code can be maintained. I will also describe certain scenarios that will explain how to test and verify the quality of the attributes I have selected. The quality attribute scenarios are divided into six different portions that all describes a certain aspects of the scenario. Source of stimulus, stimulus, artifact, environment, response and response measure. Quality attributes scenarios are further explained by Len Bass et al[2]. In the following sections I will present the quality attributes I have identified as the most important for this plugin, and give some scenarios that should describe how to verify the quality of these attributes.

5.2.1 Correctness

It is of the utmost importance that the simulations are correct depending on the solution that is analyzed. Even if one dependency is missing or wrong, the whole integrity of the simulation may be affected. Because of this concern, the primary quality attribute for this tool is correctness. The software quality attribute correctness basically concerns itself with how correct the program behaves, which is important when dealing with simulations. For this, I have created some scenarios that should help in understanding how the correctness should be and might be affected when using this tool.

C1: Parsing a Solution

The parsing process must find all the dependencies in a class towards other classes in the same solution.

Table 5.3: C1: Parsing a solution

Portion of Scenario	Value
Source of Stimulus	User
Stimulus	Loading a Solution
Artifact	The map model
Environment	At runtime after a solution has been opened in Visual Studio
Response	The map model should contain every dependency that exists in the source code
Response Measure	Manually check the source code and verify in the map model

C2: Graph Manipulation

When a user performs a graph manipulation, the plugin must ensure that the model is updated according to the changes done in the view.

Table 5.4: C1: Graph Manipulation

Portion of Scenario	Value
Source of Stimulus	User
Stimulus	Add/Remove Edge/Node
Artifact	The map model
Environment	When the solution has been loaded and the graph has been drawn
Response	The map model should be updated and the view should be updated
Response Measure	Generate mapfiles and manually check the changes

C3: Recalculating after Simulating Refactoring

This plugin uses a third party tool to simulate refactorings. When this is done, the model is changed. Therefore, after a simulation, the view needs to be recalculated based on the changes performed on the model.

Table 5.5: C2: Recalculating after Simulating Refactoring

Portion of Scenario	Value
Source of Stimulus	User
Stimulus	Recalculate
Artifact	The map model and the generated report
Environment	When the user have performed a simulation and then recalculates
Response	The view should be updated, according to the simulation
Response Measure	Manually check the changes in the report

5.2.2 Usability

For every software that is based upon user interaction, usability is important. There are certain refactoring principles in this plugin that can be difficult to fully understand, and it's therefore important that the users learn how to use the plugin appropriately.

U1: Learning To Use the Simulation Tool

The process of learning to use the plugin should be painless. This can be verified by the following scenario.

Table 5.6: U1: Learning To Use the Plugin

Portion of Scenario	Value
Source of Stimulus	User
Stimulus	Using the tool
Artifact	The plugin and the help view
Environment	At run-time
Response	User learning the tool
Response Measure	How easily the user can learn how to use the plugin. This can be checked through an interview

5.2.3 Performance

This plugin will analyze a solution by parsing all the class files inside. If the solution contains thousand, or more classes, this process can take a long time. Performance is therefore an important concern for this plugin.

P1: Loading a solution

The process of loading a solution will of course be completely related to the size of the solution. However, this process should not take longer than a minute for a solution consisting of approximately thousand classes.

Table 5.7: P1: Loading a solution

Source of Stimulus	User
Stimulus	Loading a Solution
Artifact	The plugin
Environment	The plugin at run-time
Response	The load bar
Response Measure	The load bar should represent how far in the loading process the plugin is currently at. This should take no longer than a minute

Chapter 6

Software Architecture

In this section I will present the architecture of the plugin that I have developed in Visual Studio. First I will present my reasons for choosing the architecture that I did, and why I chose this specific architecture. After that I will present the architecture. In addition to a textual explanation, I will present several different views that gives a visual representation of the architecture.

6.1 Architectural Drivers

When designing software architecture, a developer are always limited or forced to take some actions depending on the situation of development. All these aspects that can affect the architecture, positively or negatively, are called architectural drivers. In this section I will briefly describe the arechitectural drivers that has affected the process of developing this plugin.

6.1.1 Inexperience with .NET

Before I started to work on my master thesis, I had no prior experience with coding in C#, and only a limited experience with coding in C++. This meant that I had to invest some time to learn the syntax and the features of the C# library and the .NET platform. Thankfully, as an object oriented programming language, C# is very similar to Java, which I have most of my coding experience from. This made it easier for me to understand C#. The .NET platform is also very well documented, and the Microsoft Developer Network contains forums where people can post all sorts of questions regarding the .NET platform.

6.1.2 Integrating Third Party Libraries

This plugin relies on several different third party libraries. All these libraries should be implemented in a simple way and should interact well with each other. When implement-

ing these libraries I needed to learn the most important features of the library in order to fully utilize its features. The licenses of all the libraries must also comply with each other. This forced me to choose open source libraries to ensure that the licenses complied.

6.1.3 Consistent Feedback

I have cooperated closely with another developer at Powel while developing this plugin. During this time I have received a large amount of feedback regarding features and usability of the plugin. This has provided an easy way of getting external review on the plugin, which has proven invaluable as I have discovered certain bugs and faulty features which could have compromised the plugin. This also made me commit changes that I most likely wouldn't have made, and I had to make some adjustments to my original architecture.

6.2 Data-View consistency

Data manipulation is an important part of this tool, so when a change is done from the view we must ensure that the changes that are displayed is consistent with the underlying data structure. For this reason an important architectural driver for this project is Data-view consistency. If the data and the view is not consistent then the behaviour of the program and also the correctness of the simulation, as discussed in section 5.2, will be compromised.

6.3 Model View Controller

As discussed previously, this tool must have consistency between what the user sees, the view, and the underlying data structure, the model. For this particular scenario, the model-view-controller pattern fits perfectly. The MVC pattern breaks the architecture into three parts; the model, the controller and the view. The main idea behind the MVC pattern is to offer a consistency between what is shown in the view, and what the data actually contains. The view should never be updated based on what is updated in the view, but rather what is updated in the model. When a change is made from the view, which contains the user interaction, the model should be updated. After the model is updated, it should report back to the controller that a change has happened. The view should not contain any logic, so the controller has the responsibility of manipulating the model and updating the view. The MVC pattern is described by Schmidt, Douglas C. et al[27]

6.4 View Model

I have used the "4+1" view model designed by Philippe Kruchten[18] to present the architecture of the software. This model is based on several different views, that should all describe the software architecture through various diagrams. The incentive behind the use of these different views is that the different stakeholders in a software project should all

have their concerns described. Stakeholders can be developers, end-users, project managers etc. The four main views are the physical view, development view, logic view and process view.
I have used UML to describe my architecture. UML is a general purpose modeling language for software engineering[19], which is used to visualize the design of a software system. UML contains several graphic notation techniques to describe visual models of object-oriented software systems. In the view model of the software architecture I have decided to not include the physical view. The physical view shows the communication between different parts of the program and comes to it's right when it describes communication between different physical parts, like server and client communication. As the only communication I do between the different components in the software is through method calls, I think this view is redundant as I describe the inter-relational communication in other views.

6.4.1 Development View

The development view should present an overview of the developmental structure and should also show how the different parts is connected to each other. This view is designed for developers.

In figure 6.1, you can see the overall structure of the architecture. The architecture is divided into four main parts; the controller, the model, the view and the parser. The view is directly connected to the controller, which in turn is connected to the model. The parser parses the solution and reports back to the controller which in turn adds data to the model. When a change in the model occurs, the controller handles the event and updates the view accordingly.

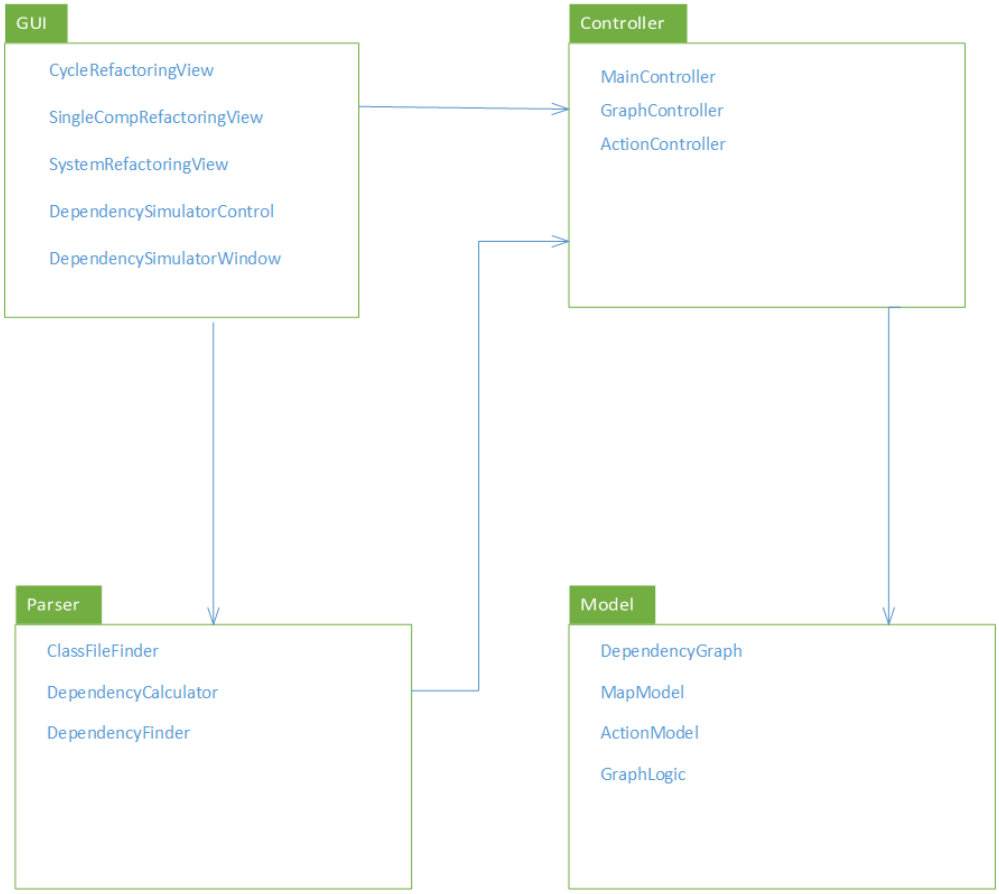


Figure 6.1: Overview of the Architecture

6.4.2 Logical View

The logical view is mainly used by the developers to give a more detailed description of all the classes in the different parts of the software.

Class Diagrams

I have created class diagrams for the most important classes in my project. These class diagrams does not define every field or method contained in each class, but rather the methods or fields that I feel is the most important in order to understand how they function.

Model

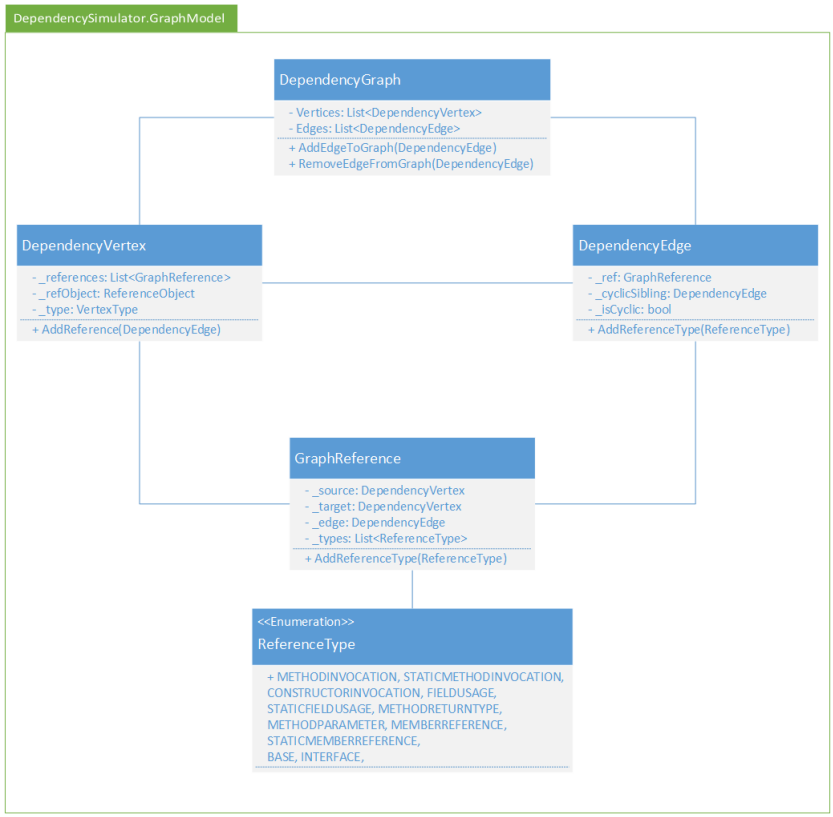


Figure 6.2: Central GraphModel classes

The class diagram presented in figure 6.2 shows the architecture of the most important classes of the graph model. This part of the program is derived from the open source project GraphSharp, found on [5]. The class DependencyGraph is the custom graph class that is shown in the tool window. This graph class consists of vertices and edges. The vertices are represented by the class ClassVertex, and the edges by the class DependencyEdge. The ClassVertex consists of a list of references for each vertex, a ReferenceObject, which can be seen in figure 6.3, and a VertexState. The VertexState implies whether the vertex is in a cycle or not. Each DependencyEdge consist of a source and a target, both of type ClassVertex. Furthermore, each edge also has a GraphReference which defines what kinds of dependencies that exists between the two vertices. The different kinds of dependencies is represented by the enum ReferenceType.

The dependency graph consists of classes and their dependencies. For this we need a model to represent these data structures. These components are shown in figure 6.3. The

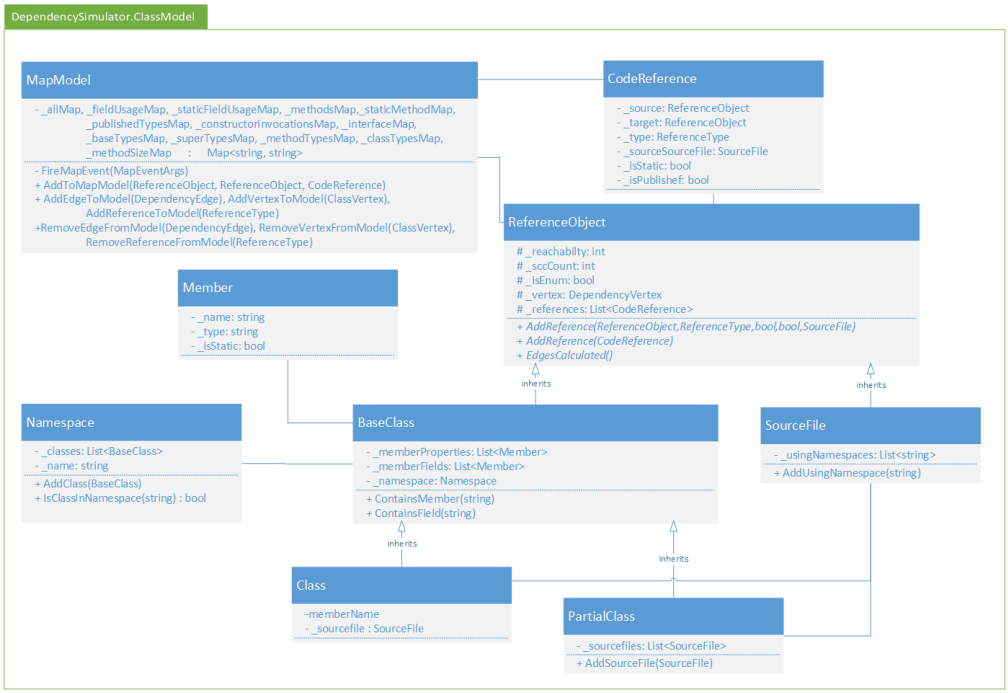


Figure 6.3: Central classes in the model

MapModel class consists if all the maps that are needed to perform simulations on the solution. There are several different maps, all described more detailed in section 4.4.1. The ReferenceObject is the underlying object that holds all the data that are relevant for source code analysis. Metrics like reachability, fan-in, fan-out, number of methods etc. are all stored in this class. Note that this class is abstract since it should not be initialized itself but only acts as a parent to other classes. There are two classes that inherits from ReferenceObject; BaseClass and SourceFile. In order to show a physical view of the solution we need a class to represent each source file.

Each SourceFile consists of a list of using statements, which simply includes the string representation of all the namespaces that are defined in the SourceFile. This is also necessary to correctly identify all the symbols that is used in each class, which I describe in more detail in section 4.6. The Namespace class includes the list of classes that exists inside the namespace and the name of the namespace. A class can only exist in one single namespace, so BaseClass has a reference to its namespace via this class. Additionally, the class BaseClass contains a list of member fields and properties, which is needed when calculating member references from another class. There are two classes that inherits from BaseClass; Class and PartialClass. The only difference between these two cases is that a partial class can be found inside several different source files, and thereby needs a list of source files, while a class simply needs a single instance of a source file.

View

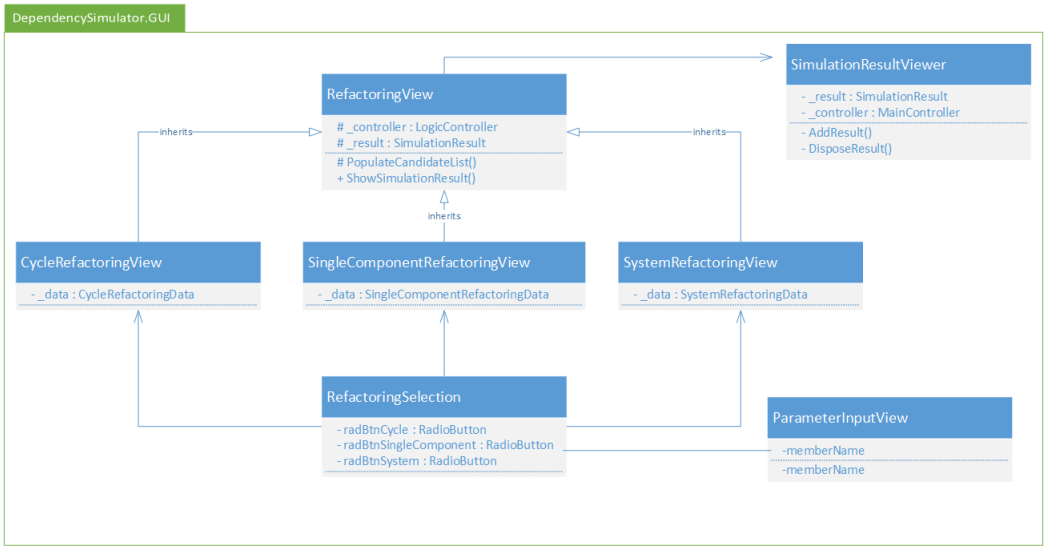


Figure 6.4: Central GUI classes

The classes shown in figure 6.4 presents the most important GUI classes. All these classes either directly or indirectly inherits from the Form class in Visual Studio. Apart from the tool window which contains the graph and list of components, these are the only other GUI classes in the project. All these classes are also related to when the user wants to simulate a refactoring by clicking on ”Simulate Refactoring” in the menu. The first view that wil e presented after clicking on ”Simulate Refactoring” is the ParameterInputView. This view contains textboxes for inputing parameters needed for the simulation, namely TopKFanin, TopKSCC and TopKCRSS. Additionally, the user have to input a main class for the simulation. This can also be set by right clicking on a vertex in the graph and selecting ”Set as main class”. The last parameter that needs to be set in this view is the sorting parameter for the candidates that will be returned to the user.

After setting all the parameters the next view is the RefactoringSelection. This class includes three different radio buttons, one each for System, Cycle and Single Component refactoring respectively. Only one can be selected, and based on the radio button selected, the user will be presented with one of three different views. The first possible is the SingleComponentRefactoringView, which contains a datagrid for viewing candidates. This candidate view is generated based on a list of SingleComponentRefactoringData, from the LogicController. In the datagridview, the user can check a checkbox to select which components should be refactored from the list of candidates. The second view is the CycleRefactoringView, which also contains datagridview to present the candidates and a checkbox for selecting components. The only real difference is that the datagridview is

populated based on a list of CycleRefactoringData. The last view is the SystemRefactoringView. Like all the other refactoring views, this also contains a datagridview for the candidates of type SystemRefactorngData. However, in this view there is no possibility of selecting components. Instead there is a textbox where the user should input the number of refactorings that should be performed on the system.

All these refactoring views inherits from the class RefactoringView. This view contains a pointer to the LogicController that generates the candidates and simulates the refactoring. Also, when a refactoring is done in one of the child views, the method ShowSimulationResult is called. This method generates a SimulationResult based on the simulation performed in the LogicController, and then initializes a SimulationResultViewer with this SimulationResult. The SimulationResultViewer displays the situation before and after the refactoring simulation. It does this by showing several different metrics. A histogram shows how many components that has a reachability within a specific range, where the ranges are 1-50, 51-100, 101-150 etc. The number of components and edges, the standard deviation and mean reachability before and after is also shown. Additionally, if system refactoring is chosen the standard deviation and mean reachability after every refactoring can be shown from a drop down list.

Parser

The classes in the Parser can be seen in figure 6.5. First, the ClassFileFinder is the class that, obviously, locates all the class files in the solution that should be parsed. Therefore, this class needs a list of projects. This list is provided by the GraphController in the constructor of ClassFileFinder. In order to properly locate all the class files inside a solution we need the three private methods stated in the class diagram. There can for instance exist solution folders inside a project, inside a solution. Then we need to loop through all the environments in these folder, and this must be done for all projects of course. Additionally, a project item in C# which is what defines a class file, can also contain several class files. For instance, a form class in C# usually consists of two class files, in its project item. So we need to loop through the children of each project item as well.

After we have found all the class files in a solution, we can send these class files to the parser. The parser is defined as the class DependencyFinder. It contains a reference to both the GraphController, which it reports back to, and the DependencyCalculator when it's done parsing. The parsing process is started by the GraphController by calling ParseClassFiles(List<ClassFile>) in this class. For each class file in the solution, the DependencyFinder class loops through the abstract syntax tree provided by the NRefactory library. All nodes in the abstract syntax tree, that are of interest, are analyzed and all dependencies that are found are reported back to the GraphController.

The DependencyCalculator class then takes all the dependencies and the classes that the DependencyFinder found and calculates the correct nodes to place the dependency between. As I discuss in section 4.2, there are certain limitations to a parser, and this is solved in this class. Several data structures with information like using statements, names-

paces, classes are manipulated in this class in order to identify all the dependencies in each class. When a new dependency is found, this class also reports back to the GraphController which updates the model. The dependencies between source files are also calculated. After all the dependencies have been calculated, the GraphController is notified, and updates the view.

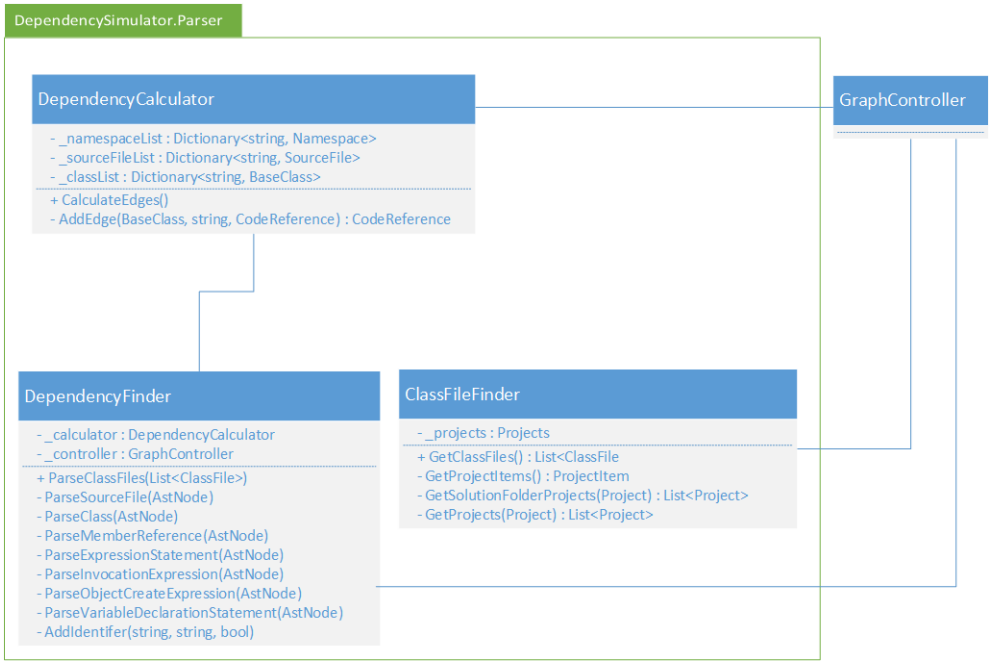


Figure 6.5: Parser classes

Controller

The controller classes in the plugin are responsible for making changes in the model, and then updates the view based on these changes. In figure 6.6 the controller classes are presented. The MainController serves as a singleton class that contains all the global instances of each of the other controller classes. When another component in the solution needs to access one of the controller classes, they call this class first.

The GraphController contains the GraphModel which represents the graph with all the nodes and edges. When a class is found in a solution this class adds a new node to the MapModel. The MapModel then contains an EventHandler which reports back to the GraphController with the information that it needs in order to update the GraphModel.

The ActionController is used whenever a graph manipulation occurs. When, for instance, a new node is added to the graph via the graphical user interface, a new GraphAction is added. This action is then added to a list in the ActionModel of currently active actions. If the user reverts the action by choosing ”Undo” in the menu, the previously added GraphAction is added to another list of currently removed actions simultaneously as it’s removed from the list of currently active actions in the ActionModel. Vice versa will happen if the user should redo the action later.

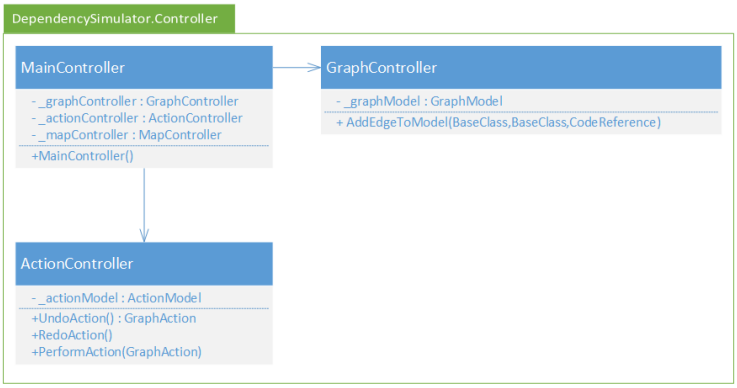


Figure 6.6: Controller classes

Sequence Diagrams

Loading a Solution

In figure 6.7 you can see a sequence diagram of how the program loads a solution. The entry class is the DependencySimulatorWindow, which inherits from the Visual Studio environment class ToolWindowPane, which defines a window pane in Visual Studio with plugin capabilities. This tool window The DependencySimulatorControl is the class that is directly connected to the components in the Tool Window. Because of this, this class needs to have access to the GraphModel, via the GraphController. Regarding the MVC pattern, it's important to note that the GraphController listens to the MapModel and adds an Edge in the GraphModel according to the event which is handled from the MapModel. After the graph is loaded, the DependencySimulatorWindow, which holds all the bindings directly connected to the Visual Studio environment, updates it view based on the newly generated graph in the GraphModel.

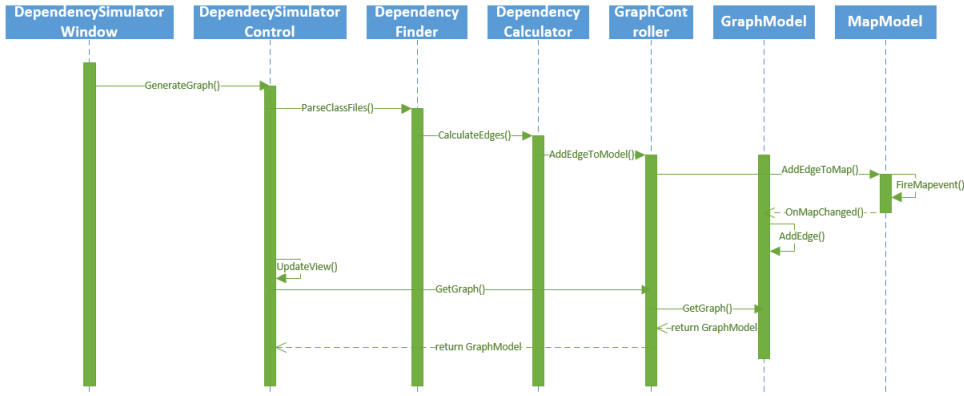


Figure 6.7: Sequence diagram of loading a solution

Simulating a System Refactoring

One of the three types of refactoring, as described in section 4.5.5, is system refactoring. The sequence diagram in figure 6.8 presents this process. The DependencySimulatorWindow is the class that contains the button ”Simulate Refactoring” shown in figure 4.1. This button starts the whole process of simulating a refactoring. When the button is pressed the DependencyController class is called, which calls all the other GUI classes and holds the data bindings to the graph and the list view in the main screen. First, the ParameterInputView view is called so the user can input the parameters that are needed for the simulation. Then the RefactoringSelection class is called to let the user choose between the three different types of refactoring. In this scenario, the system refactoring is chosen so the next class that is called is SelectSystemRefactoringView. In this view the user must select a reduction rate in order to choose candidates. This class will then call the LogicController to get the list of candidates, which it will use to initialize the next class, the SystemRefactoringView. The SystemRefactoringView will also call on the LogicController in order to simulate a system refactoring, which again calls the GraphLogic to calculate the simulation. Lastly the SimulationResultViewer is called to present the result of the simulation.

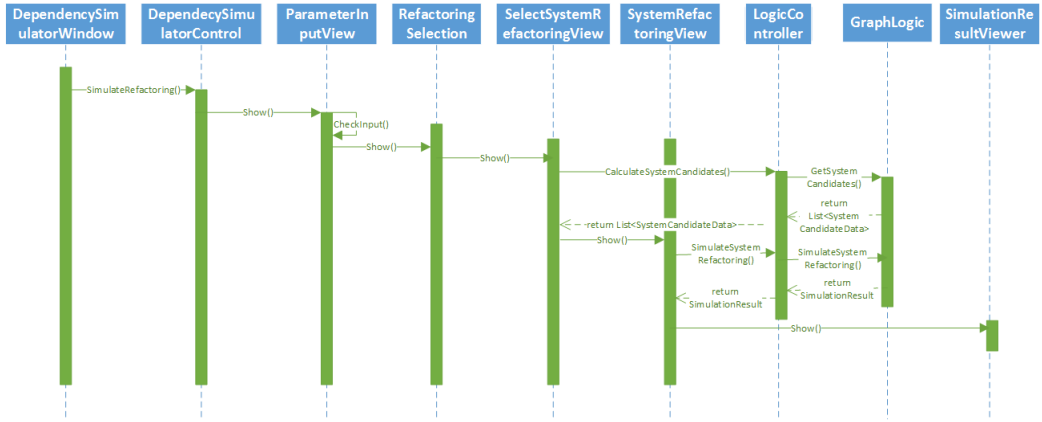


Figure 6.8: Sequence diagram of simulating a system refactoring

6.4.3 Process View

The process view focuses on the runtime environment and explains the system processes and captures the concurrency and synchronization aspects of the design. The process view should be used by developers and integrators.

Simulating Refactoring

The activity diagram in figure 6.9 shows the process of simulating a refactoring. The process is started when the user choose to simulate a refactoring. After inputing the parameters that is needed, the user must select what type of refactoring that should be simulated. After a refactoring, the result is shown and the process is finished.

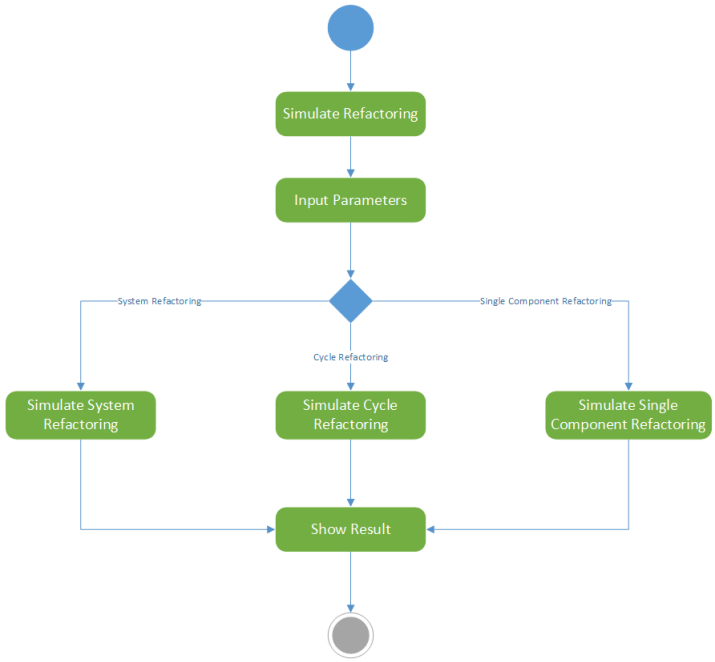


Figure 6.9: Activity diagram of simulating a refactoring

6.4.4 Scenarios

The scenarios, or the use-case view as it's also called, are used to illustrate interactions between actors systems through several different use cases. The use-cases in this section presents some of the most important features of the plugin, both as use-case diagrams and descriptions of the scenarios. The scenarios are useful for end-users in visualizing possible features of the tool

Use case 1: Loading a Solution

Identifier: UC 1
Description: Loading a solution from the Dependency Simulation Window

Preconditions: The user has loaded the solution into Visual Studio

Postconditions: The solution will be visualized in the Dependency Simulator Window, based on the source code.

Basic Course of Action:
1. The user clicks the ”Load Solution” button on the menu line in the Dependency Simulation Window
2. The program analyzes the source code for components, this can take several minutes, based on the size of the solution
3. The program generates a dependency graph

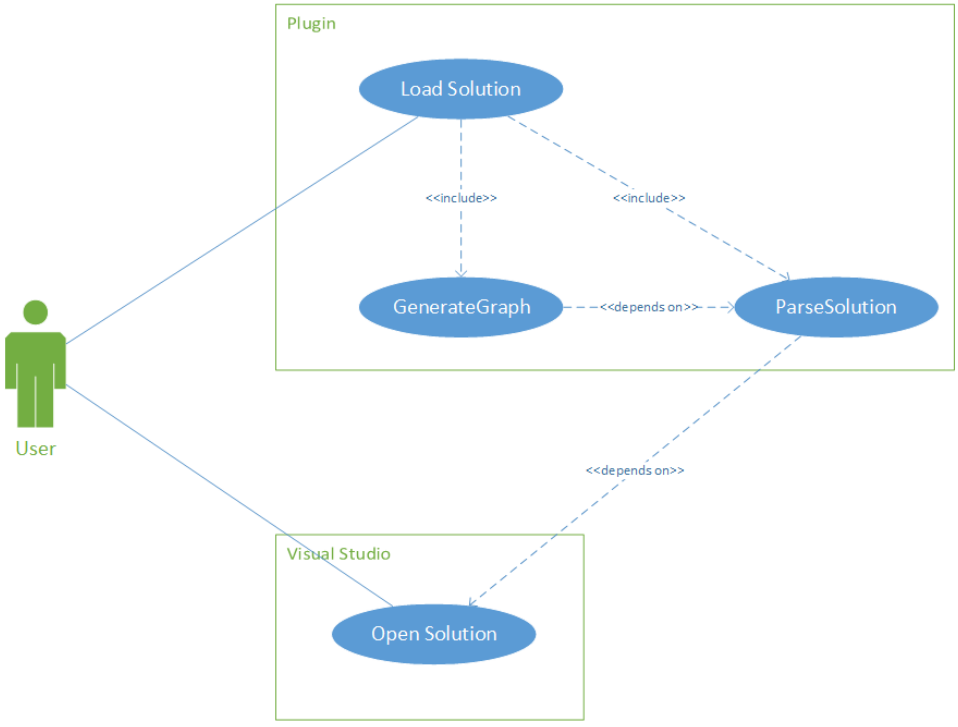


Figure 6.10: UC1: Loading a Solution

Use case 2: Graph Manipulation

Identifier: UC 2
Description: Manipulate the graph

Preconditions: The user has loaded the solution and the graph has been generated.

Postconditions: The graph will be updated based on the manipulation.

Basic Course of Action:
1. The user manipulates the graph, by either adding, removing or editing edges and nodes
2. The manipulation updates the model, which in turn updates the view via the GraphController

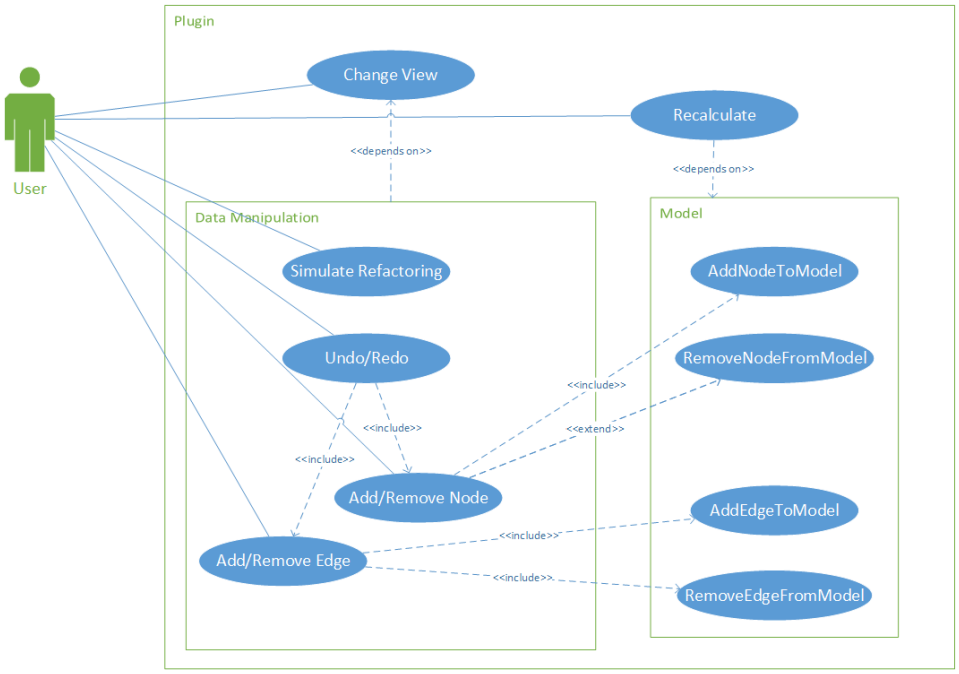


Figure 6.11: UC2: Graph Manipulation

Use case 3: Simulate Refactoring

Identifier: UC 3
Description: Simulate Refactoring

Preconditions: The user has loaded the solution and the graph has been generated.

Postconditions: The plugin will simulate a refactoring and generate a result, with pre and post values of important metrics

Basic Course of Action:
1. The user chooses ”Simulate Refactoring” from the menu.
2. The user inputs parameters for the simulation and chooses what kind of simulation that should be performed.
3. After choosing simulation type, the user will be presented with a set of candidates he can choose from.
4. The plugin will simulate a refactoring based on the candidates.
5. After simulating, the plugin will generate a result, which can be accepted or not.

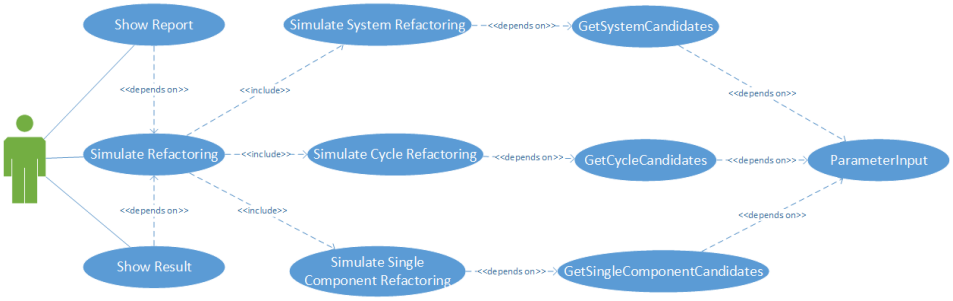


Figure 6.12: UC3: Simulate Refactoring

Part V

Evaluation

Chapter 7

Results

To analyze the tool I have performed simulations on two different kinds of projects, before manually committing the changes reported by the simulation and comparing the results. We want to test the plugin in both an industrial setting and in an open source environment. We wanted to do this to see if there is any significant changes from these two different scenarios. Open Source projects can have several hundreds different contributors and can become hard to maintain because no one has a complete comprehension of the whole structure. Projects from an industrial setting can also have many contributors, but other aspects, such as time to delivery can also affect the overall quality of the software.

7.1 Case study #1: VidCoder

The first project that I have analyzed is an Open Source project called VidCoder. VidCoder is a software for ripping and encoding video files in a Windows environment. Obviously, it's written in C#. The project can be found at CodePlex[6]. I have simulated a System Refactoring and recorded the fitness values before and after the manual refactoring and compared them to the fitness values that the simulation showed.

The simulation suggested to extract the interface of six different classes in the solution. All those classes, however, inherited, either directly or indirectly, from an external public abstract class, ViewModelBase, in a third party library. So there was no possibility of refactoring away this dependency. The solution to this was to introduce an intermediary class between ViewModelBase and the six classes that inherited from this class. There are some concerns with this approach, compared to simply extracting an interface. As with interface extraction we start by extracting all the public methods. C# also has member properties, which works just like getter and setter methods in Java, and can contain any implementation. As they behave just like how a method does, these member properties must also be extracted. Now, since all the extracted elements from the class is inserted into an abstract class rather than an interface we must make some adjustments. Where an

interface can simply declare a method by its name, return type and parameter, a method in an abstract class must declare an access modifier and some sort of implementation. Obviously, a virtual or abstract method cannot be declared as private, but apart from that the access modifier can be set to anything, as long as its set. It can contain a concrete implementation, or empty implementation, and be marked as virtual if the inheriting classes can give its own implementation. Or it can be marked as abstract and not contain an implementation, then the inheriting class must have its own implementation. Summarizing; a virtual method must declare a body, while an abstract method cannot declare a body. See figure 7.1 for example methods.

```
public virtual void VirtualMethod() { }

public abstract void AbstractMethod();
```

Figure 7.1: Two different ways of declaring a method inside an abstract class

In VidCoder, I obviously didn't want to risk changing any implementation, so I chose to set every method and member property as abstract in the abstract class that I introduced for each of the six classes that I refactored. By setting it as abstract I ensure that the class that will inherit from it must implement all methods or properties inside the abstract class.

Original Fitness Values	
Number of Components	245
Number of Edges	482
Mean Reachability	17.224
Standard Deviation for Reachability	37.631
Highest Cycle Set Size	14
Reachability Data	
1-50	213
51-100	0
101-150	32

Fitness Values from Simulation	
Number of Components	252
Number of Edges	538
Mean Reachability	10.785
Standard Deviation for Reachability	21.064
Highest Cycle Set Size	8
Reachability Data	
1-50	234
51-100	17
101-150	1

Table 7.1: Fitness values from VidCoder before manual refactoring.

Number of Components	252
Number of Edges	521
Mean Reachability	10 .674
Standard Deviation for Reachability	24.262
Highest Cycle Set Size	8
Reachability Data	
1-50	237
51-100	6
101-150	9

Table 7.2: Fitness values from VidCoder after manual refactoring

As you can see by the fitness values in table 7.1 and 7.2, the number of components and edges have increased after the refactoring. This is of course as expected, as I have introduced seven new classes that each contains several edges to other components. What is much more interesting however, is the results from the reachability data. The mean reachability has decreased quite significantly, its almost halved, and the standard deviation is also much lower after the manual refactoring was performed. Another important metric, the highest cycle set size, also decreased from 14 to 8. Even though the results are very good, they are not as good as the values from the simulation results. There could be several reasons why this is. In the simulation, the tool suggested to create a singleton registry and register instances of all the six classes that was to be refactored in the solution. However, VidCoder already used some sort of registry in Unity. Unity is a registry for objects and types created by Microsoft and already contains several features that would be hard and would also require a lot of time to replicate. When I refactored these edges I therefore chose to keep the usage of Unity and didn't introduce a custom registry. This have certainly affected the results, but it would be difficult to get any precise data on how they were affected.

7.2 Case Study #2: Powel

I have had a close relationship with Powel during my thesis. Together with Tosin and a developer at Powel I have analyzed one project at Powel and have gotten some encouraging results from the simulation. However, due to work load and the severity of the refactoring reported in the simulation this developer at Powel hasn't been able to fully refactor the source code according to the simulation results. What we have gotten though, is some valuable feedback regarding the potential and the usability of the tool according to a third person developer. In an interview conducted by me and Tosin with the developer, he had some valuable insight on what he thought was good about the tool, and what he thought could be improved. This feedback has also helped me in identifying further features that could be implemented in the plugin and future work. A wanted feature was for instance the ability to view where in the source code a dependency on a class existed in a given class. For example, lets say that class A depends on class B through a method invocation inside MethodA() in class A. Then the developer wanted the ability to go to the exact line in the source code that contained this method invocation. At Powel I also had two presentations for some of the other developers and managers there. The most valuable feedback we got from these presentations was that none of the developers that were present said they would trust an automated refactoring tool of this scale. It would be too dangerous to blindly trust an automated refactoring tool that changes dependencies and the architectural structure of a software system.

Chapter 8

Project Evaluation

In this chapter I will evaluate the software compared to the research method described in section II. I will reflect on whether the plugin meets the functional requirements and how the quality requirements are implemented in the software system.

8.1 Research Method

I have chosen to use design science as my research method. As described in chapter 2.2, design science consists of four different parts. When I started writing my thesis I started with examining existing tools for refactoring, and read different articles that described refactoring from several different points of view. This was the theory building process of my thesis. After that, I identified what my thesis should contain and what the contribution should be, as part of the problem diagnosis. The technology invention was the plugin that I developed. Finally the technology evaluation consisted of performing simulations and manually refactor the source code based on the simulations, in both an industrial setting at Powel and with an open source project. While working with my thesis, I have continually improved each of these parts iteratively. Design science is an iterative

8.2 Functional Requirements

This section will evaluate all the functional requirements presented in section 5.1, and verify if these requirements are met. Many of these functionalities has already been described more thoroughly in the Contribution part, so this will just serve as a quick summary of whether the functionalities are supported.

8.2.1 The Plugin

Following is a list of all the functional requirements concerning the plugin, and a quick description whether the requirement is met.

FR1: *The parser should be able to analyze C# solutions.*
The parser uses the NRefactory library and can successfully load a C# solution.

FR2: *The plugin should give the user feedback on the loading process.*
By using a loading bar, the plugin gives the user feedback about how far the loading process has proceeded.

FR3: *The plugin should be able to visualize a solution.*
The GraphSharp library enables the plugin to visualize a solution in a graph, where each node represents a class and each edge represents a dependency between two classes.

FR4: *The plugin should be able to detect and visualize dependency cycles.*
The analysis tool implemented in the plugin detects dependency cycles and the plugin colors the nodes in a cycle orange.

FR5: *The plugin should be able to show all the classes in a list view that contains important metrics for each class.*
The plugin contains a list view, as shown in figure 4.1, which contains several metrics for all the classes.

FR6: *The plugin should be able to simulate refactorings.*
By using the analysis tool, the plugin can simulate refactorings.

FR7: *The plugin should show the result of a refactoring simulation.*
After a simulation is done, the result is shown in the SimulationResultViewer.

FR8: *The plugin should be able to generate a report based on the simulation.*
If a simulation result is added to the report in the SimulationResultViewer, then the plugin will generate a report which the user can access.

FR9: *The plugin should be able to show a class-view and a file-view.*
The plugin generates both a graph based on all the classes, and a graph based on all the source files. This different graphs are available by changing views.

8.2.2 User Interaction

The functional requirements of the user interaction are described in this section.

FR10: *The user should be able to add a new class to the graph.*
The context menu of the graph view contains a menu item that allows a user to add

nodes to the graph.

FR11: *The user should be able to add an edge between two nodes in the graph.*
In the drop down list under the ”Click Action” menu item, the user can choose ”Add Edge” and click on two nodes to add an edge between them.

FR12: *The user should be able to remove a node from the graph.*
When right clicking on a node, the user can choose ”Remove Edge” to remove it.

FR13: *The user should be able to remove an edge from the graph.*
By right clicking on a node and choosing ”Remove Edge” in the context menu, the user can remove the edge.

FR14: *The user should be able to sort the classes in the list view, based on each metric.*
The plugin uses a DataGridView to contain the classes in a class view, which supports sorting by all columns.

FR15: *The user should be able to show and hide different metrics in the list view.*
The user can right click on any of the column headers in the list view to display a context menu that contains all the supported metrics and an option to show or hide the metric.

FR16: *The user should be able to set a class as the main class in the graph.*
The context menu that is displayed when right clicking on a node contains an option to set the corresponding class as the main class.

FR17: *The user should be able to input parameters for a refactoring simulation.*
When the user starts to simulate a refactoring the ParameterInputView is shown which contains several parameters that the user can change.

FR18: *The user should be able to choose between three different types of simulations: System refactoring, cycle refactoring and single component refactoring.*
After inputing parameters, the user can choose one of these refactorings in the RefactoringSelection.

8.3 Quality Requirements

The quality requirements that I defined in section 5.2 are the nonfunctional requirements of the plugin. These requirements should also be evaluated in order to verify that these requirements are met

8.3.1 Correctness

C1: *Parsing a Solution*
When the plugin parses a solution, we must ensure that every dependency is found.I have tested this by performing an accuracy test of the parser, which is explained in section 9.1. In this section I present the accuracy test that I performed on the NRefactory project. The accuracy test concludes that the plugin found all the dependencies that existed in the seven classes that I manually reviewed. Based on my tests I can therefore claim that the plugin meets quality requirement C1.

C2: *Graph Manipulation*
I have performed several test of the graph manipulation feature in the plugin. I have performed these tests by loading the VidCoder solution, generating the map files associated with this solution and manually noted the dependencies between ”VidCoder.Services.Processes” and ”VidCoder.Services.IProcesses”. The only dependency between these two classes were through an interface dependency from ”Processes” to ”IProcesses”, so the dependency between these two classes existed only in the interface map and the all types map. These map files are the ones that are described in section 4.4.1. Then I performed several graph manipulations. First I tried to remove the edge between those two classes. After the view was updated I generated the map files again and checked the dependencies between these two classes. I then confirmed that the dependency in the interface map, the all types map was removed.

The second test I performed was to remove the class ”VidCoder.Services.IProcesses”. This class is used by three other classes; ”VidCoder.Services.Processes”, ”VidCoder.Services.ProcessAutoPause” and ”VidCoder.Services.AddAutoPauseProcessDialogViewModel”. After removing the class from the view I confirmed that the class was not present in any of the maps, and that none of the three classes had any dependency towards it anymore. The plugin meets the requirement of C2.

C3: *Recalculating after Simulating Refactoring*
This quality requirement is concerned with the changes in the model when a simulation is performed. When performing one of the integration tests, Test 3, which is described in section 9.2, I generated the report and saved it. Then I recalculated the graph and generated the map files again. Then I verified that all the changes that were reported had been done in the map files. All the changes that were reported had been changed in the maps as well, so the plugin meets quality requirement C3.

8.3.2 Usability

U1: *Learning to use the Plugin*
The only other developer that has used the plugin other than myself is Tosin Daniel Oyetoyan and the developer who helped us to test the plugin at Powel. However, since Tosin already knew all the refactoring strategies used by the plugin, he had no problem in

learning how to use it. When the developer at Powel was trying the tool, the help view was not implemented, so Tosin and I had to explain him the basics of the plugin ourselves. Therefore it's hard to verify this quality requirement. In the interview that Tosin and I had with him, he said that the plugin could be difficult to learn but also stated that the implementation of a help view should improve how fast a developer could learn how to use the plugin.

8.3.3 Performance

P1: *Loading a Solution*
This quality requirement should reflect how long it takes to load a solution. Loading a solution consisting of approximately thousand classes should take no longer than a minute. When loading VidCoder, which consists of 245 different classes, I timed the time it took to load the solution ten different times. The average loading time based on this test was 4.8 seconds. A quick calculation shows that a similar project consisting of thousand classes, should finish loading in ca. 20 seconds. Which is well within the limit of a minute. However, the results that I got from loading the VidCoder solution is not definite proof that the quality requirement is met, but most probably. There are of course several factors when loading a solution, and most importantly processing power. If this plugin is used on an older computer the results would most likely be much worse.

Chapter 9

Testing and Validation

To ensure that the plugin is generating the most accurate results regarding the simulation, we must ensure that the tool is correctly calculating all the dependencies between the different components. In this section I will first test whether the plugin finds all the dependencies that exists within a class file. Then I will test the integration with the java tool which this plugin uses, to ensure that they report the same results.

9.1 Accuracy Test

In order to test the accuracy of the parser and dependency finder, I have manually reviewed seven different classes in the NRefactory solution[12]. In these classes I have logged all dependencies to other components in the same solution. The classes I have manually reviewed have been selected based on the differences between them, so that these classes will represent a stratified sample. Some have almost no outgoing dependencies and others have several. Some have only static methods while others have zero.

The following diagram shows the different kinds of dependencies found in each class based on the maps that this tool needs for analysing the solution. These maps are described in section 4.4.

9.2 Integration Test

In this section I will present my findings on some test that I have performed in the plugin compared with the Java tool developed by Tosin, which is used to simulate refactorings. It

[1] NS = Nonstatic, S = Static

Table 9.1: Accuracy of Parser on Test Project

Class	Field usage		Static field usage		Method Inv.		Static Method Inv.	
	Tool	Actual	Tool	Actual	Tool	Actual	Tool	Actual
JsonTextWriter	1	1	6	6	2	2	5	5
JValue	2	2	1	1	4	4	4	4
BsonBinaryWriter	6	6	1	1	1	1	2	2
DateTimeUtils	1	1	4	4	4	4	1	1
DataSetConverter	2	2	1	1	5	5	0	0
Product	0	0	0	0	0	0	0	0
DynamicWrapper	3	3	0	0	2	2	0	0

Table 9.2: Accuracy of Parser on Test Project

Class	Constructor Inv.		Published Types		Method Types [1]		All Dependencies	
	Tool	Actual	Tool	Actual	Tool	Actual	Tool	Actual
JsonTextWriter	1	1	1	1	NS	NS	11	11
JValue	2	2	1	1	NS & S	NS & S	10	10
BsonBinaryWriter	0	0	1	1	NS	NS	10	10
DateTimeUtils	1	1	0	0	S	S	6	6
DataSetConverter	1	1	3	3	NS	NS	7	7
Product	0	0	0	0	NS	NS	0	0
DynamicWrapper	2	2	0	0	S	S	3	3

is essential that the results from this tool is exactly the same as the results from Tosin's Java tool. If not, the whole integrity of this tool will be under scrutiny. These simulations are only used for testing the integrity of the tool and will not be discussed as part of the results of this tool. The simulations have been performed on the open source project VidCoder. For all the simulation tests that has been performed, the following parameters have been used:

TopK - Fanin: 20
TopK - SCC: 10
TopK - CRSS: 20
Sorting priority: SCC

For all the different tests that will be performed, we note the most important values, like mean reachability, highest cycle set size and reachability data. The following tests have been performed:

Test 1: System Refactoring
First we will select "VidCoder.App" as the main class. Then we will select "Simulate Refactoring" from the tools menu. We then choose "System Refactoring" from the refactoring selection view. In the view that shows all the candidates in a list, we input "4" as the number of refactorings.

Test 2: Cycle Refactoring
The second test will be to test the integrity of the cycle simulation features of the tool. First we select ”Simulate Refactoring” and then ”Cycle Refactoring” to start the test. In the cycle selection view we choose the cycle with the highest cycle set size. In this example, that would be the cycle that contains the class ”VidCoder.ViewModel.Components.PresetsViewModel” and the cycle set size is 14. When presented with the candidates we choose every candidate by double clicking on the ”Select” column in the list view.

Test 3: Single Component Refactoring
As with all the other simulations, we must first choose ”Simulate Refactoring”. Then we select ”Single Component Refactoring”. When confronted with the candidates from the tool we select the top 10 components based on the sorting priority.

In table 9.3, the columns named C# will present the numbers found by the plugin, while the Java column shows the numbers found by the Java tool which is implemented in the plugin. The results should of course be exactly the same.

Table 9.3: Integration Test on NRefactory

Test	Post Mean		Post Std. Dev.[2]		Post Nr. of Comp.		Post Nr. of Edges		Post SCC[3]	
	C#	Java	C#	Java	C#	Java	C#	Java	C#	Java
Test 1	10.30	10.30	20.13	20.13	250	250	538	538	8	8
Test 2	18.49	18.49	38.55	38.55	253	253	522	522	14	14
Test 3	19.80	19.80	41.58	41.58	256	256	519	519	15	15

[2]Post Standard Deviation of class Reachability
[3]Highest Cycle Set Size

Chapter 10

Discussion

The challenges to refactor existing software systems is not trivial, especially at the class granularity level. This work is an additional and significant contribution to existing work in resolving structural complexities of software systems.

Furthemore the developed tool is based on a sound software engineering theory and practice. Also, I have performed extensive review of existing tools and to the best of my knowledge, this is plugin offers new functionality from what existing tools are providing.

I have used the Visual Studio SDK and several other third party libraries in the plugin. Based on the feedback from the developer at Powel, the plugin is quite easy to use and is implemented into the Visual Studio environment, giving it several useful features. The developer at Powel stated in an interview: "*...with a proper help file, It shouldn't take long time to learn to use the tool by yourself...*" and agreed that the plugin was easy to learn quickly.

The results I have gotten from Powel and the open source project VidCoder, are encouraging. I manually refactored VidCoder and managed to reduce the complexity by quite a large margin, especially considering that I only refactored six classes. The mean reachability was reduced by almost 40 % and the standard deviation by approximately 35 %. The biggest cycle that contained 14 elements was broken and the highest reachability range, from 101 to 150 classes, contained only 9 classes compared to 32 classes before the refactoring.

The industrial validation of the tool that we did at Powel gave me several useful insights. First, It is very positive that the developer we interacted with accepted many propositions for refactoring from the tool. In addition, there are some inappropriate and surprising coupling decisions that the analysis tool revealed. His feedback of the tool in the interview that we had with him was also positive.

10.1 Research Questions

In section 2.1 I defined some research questions for my thesis. In this section I will answer the questions and discuss how it has been incorporated into my thesis.

RQ1: *Can we help developers in making architectural changes at the class granularity level, by using this plugin?*

In chapter 7, I presented my results from both the VidCoder project and from Powel. These results was quite conclusive in verifying that this plugin clearly can help developers in performing refactorings at the class level. The visualization of the source code gives developers means to visualize the inter-relational dependencies between the classes in a solution, which can help in identifying potential refactorings. Additionally, the list view contains several metrics that also can help developers in discovering classes that could be refactored. Lastly, and most importantly, the plugin has the ability to analyze the source code based on the reachability, fan-in and number of strongly connected components to calculate the best candidate classes to refactor.

RQ2: *How much improvement can the proposed refactorings provide?*

The results that I got from the VidCoder project clearly showed that the refactorings, which the simulation suggested, improved the overall structure of the architecture. The results from VidCoder indicated an improvement of approximately 40% in the mean reachability, which is a big improvement This manual refactoring took approximately two hours of actual coding, which indicates that even with a small amount of effort, big improvements can be made, using this plugin.

Chapter 11

Conclusion

In my master thesis I have developed a plugin for Visual Studio that can visualize the source code in a C# solution and then simulate refactorings, based on the structural complexities in the architecture. I have presented my findings on existing tools, and concluded that this plugin can help developers in refactoring several structural complexities that most other tools cannot do. The results from the plugin regarding the improvement of software structure are also encouraging, as I discussed in Chapter 10.

For future work, I have identified some improvements that can be made in the plugin. When the plugin presents all the different dependencies between two classes there is no way to automatically go to the source code where this dependency is defined. This is something that would be very useful when refactoring, as the developer could alter these dependencies more quickly. Another possibility would be to implement support for implicitly typed variables. As of now, these variables are not supported by the plugin. This feature is a relatively new feature in C#, and is not used as much as explicitly typed variables. However, this is still something that would affect the architecture of the software, and should therefore also be reflected in the analysis that the plugin performs on source code. In the future there would also be a possibility of adding functionality to perform automatic refactoring based on the simulations that the plugin proposes. As of now, it does not seem as though most developers would trust such a feature, but after performing several manual refactorings they might change their attitude.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compiler Principles, Techniques and Tools*. Pearson Education, 2007.

[2] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice(2nd Edition)*. Addison Wesley, 2003.

[3] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo. When does a refactoring induce bugs? an empirical study. In *IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 104–113. IEEE, 2012.

[4] Chris Chedgey, Paul Hickey, Paul O'Reilly, and Ross McNamara. Structure101, 2013. URL: `http://structure101.com`.

[5] CodePlex. Graphsharp, 2009. URL: `http://graphsharp.codeplex.com`.

[6] CodePlex. Vidcoder, 2013. URL: `http://vidcoder.codeplex.com`.

[7] Clarkware Consulting. Jdepend, 2009. URL: `http://clarkware.com/software/JDepend.html`.

[8] Franz-Josef Elmer. Classycle, 2012. URL: `http://classycle.sourceforge.net`.

[9] Donald Firesmith. Quality requirements checklist. 2005.

[10] Martin Fowler, Kent beck, John Brant, William Opdyke, and Don Roberts. Refactoring: Improving the desing of existing code. 1999.

[11] Jeroen Frijters. Ikvm.net, 2013. URL: `http://www.ikvm.net`.

[12] GitHub. Nrefactory, 2009. URL: `https://github.com/icsharpcode/NRefactory`.

[13] Odysseus Software GmbH. Stan, 2013. URL: `http://stan4j.com/introduction/introduction.html`.

[14] Goldfinch. Nitriq, 2013. URL: http://www.nitriq.com/features.

[15] Google. Codepro analytix, 2012. URL: https://developers.google.com/java-dev-tools/codepro/doc/features/features.

[16] Tao Jiang, Ming Li, Bala Ravikumar, and Kenneth W. Regan. Formal grammars and languages. pages 20–20. Chapman & Hall, 2010.

[17] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. 2012.

[18] Philippe Kruchten. Architectural blueprints - the "4+1" view model of software architecture. 1995.

[19] Craig Larman. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall, 2004.

[20] Hayden Melton and Ewan Tempero. Formal grammars and languages. pages 87–95. Australian Computer Society Inc., 2007.

[21] MSDNA Microsof Developer Network. Tutorial 1: Getting started with visual studio extensibility, 2014. URL: http://msdn.microsoft.com/en-us/library/bb330853.aspx.

[22] Emerson Murphy-Hill and Andrew P Black. Breaking the barriers to succesful refactoring: Observations and tools for extract method. In *ACM/IEEE 30th International Conference on Software Engineering*, pages 421–430, Leipzig, Germany, 2008. IEEE.

[23] Steffen M. Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. pages 390–400, 2010.

[24] Steffen M. Olbrich, Daniela S. Cruzes, and Dag I.K. Sjberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[25] Tosin Daniel Oyetoyan, Daniela S. Cruzas, and Reidar Conradi. A study of cyclic dependencies on defect profile of software components. 2013.

[26] Tosin Daniel Oyetoyan, Daniela S. Cruzas, and Reidar Conradi. Decision support approaches for refactoring components in dependency cycles. 2014.

[27] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects. Vol. 2*. John Wiley & Sons, 2013.

[28] S.M.A. Shah, J. Dietrich, and C. McCartin. Making smart moves to untangle programs. In *Software Maintenance and Reengineering (CSMR)*, pages 359–364, Szeged, Hungary, 2012. IEEE.

[29] Patrick Smacchia. Ndepend, product features, 2013. URL: http://www.ndepend.com/Features.aspx.

[30] Nikolaos Tsantalis, Alexander Chatzigeorgiou, Theodoros Chaikalis, and Marios Fokaefs. Jdeodorant, 2013. URL: http://www.jdeodorant.com.

[31] John Venable. A framework for design science research activities. In *Information Resources Management Association International Conference*, pages 184–187. Idea Group Inc., 2006.

[32] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects with program dependencies. 2009.

Part VI

Appendices

Appendix A

Installation Guide

A.1 Requirements

This program is a plugin for Microsft Visual Studio, so a valid license is needed. This plugin requires Visual Studio 2013 to work. In addition to several different C# libraries, this plugin also uses a java tool for analysis and simulation. Because of this, the plugin needs the IKVM library to run the java tool. When installing this tool you must therefore also have a copy of the dynamic link libraries of IKVM that are needed. These dynamic link libraries are: *IKVM.OpenJDK.Core.dll* and *IKVM.OpenJDK.Runtime.dll*. These dll's are found on sourceforge at *http://sourceforge.net/projects/ikvm/files/*.

A.2 Installation

The binaries for this plugin should include all the necessary files except for the IKVM files arleady discussed. In addition to the libraries, there is also a .vsix file located inside the source folder. This is the installer for the plugin, or the VSIX Package, as they are called. Simly douck click on this file and a dialog will apear. This dialog will contain all the installations of Visual Studio that support the installation of the plugin. Choose the preferable installation and click "Apply". This prcess can take a few seconds to finish. After the installation is done you need to locate the installation folder. This folder will be located in the %AppData path in your environment. The %AppData folder is usually located inside C:/Users/*YourUser*/AppData, note that this folder usually is hidden. Inside the AppData folder you must locate the folder %AppData/Local/Microsoft/VisualStudio/*VersionofVisualStudio*/Extensions. This folder will include a randomly generated folder which contains the binaries of the installation. You must manually check every folder and find the one that includes all the binary files from the installation folder. Once you have located the folder, place the two aforementioned dll's into this folder. You should now be set to go.

Appendix B

User Guide

B.1 Loading a solution

Start the version of Visual Studio in which you installed the plugin. Load a solution into Visual Studio. Locate the Dependency Simulator at View → Other Windows → Dependency Editor. A new window will appear with several options. First, you should click Load Solution. Be sure that there is an active solution loaded into the environment. This will draw a graph of all the classes in the solution and the dependencies between them.

B.2 Graph Manipulation

You can now manipulate this graph to some degree: You can add or remove nodes(Nodes doesn't always spawn where you want them to be, usually in the upper leftmost corner). You can add or remove edges between nodes. Add edges by choosing ”Add Edge” in the ”Click Action”. Remove edges by right clicking on them and then click ”Remove Edge”. If the edge that you are removing is cyclic, a new window will appear, asking which edge should be removed. Read carefully about which nodes are source and target so that the correct edge is removed. You can undo and redo these aforementioned actions. Note that the current actions will be reset when you add a simulation to the report because this can potentially change the whole structure of the graph. You can edit references for any given edge. An edge can consist of many references: Method return type, mehod invocation, member field etc. This is reflected in the underlying mapstructure which this tool uses to simulate refactorings. If the edge your editing is cyclic, a new window will appear, asking which edge that you want to edit. Be sure to pick the right one. You can view info about each edge which basically will tell you what sort of referencetypes this edge consists of and if it's cyclic or not. You can also list all the incoming and outgoing edges by clicking ”List Edges”. By choosing ”Change View” the underlying fileview will be

presented. Click again to change back.

B.3 Simulate Refactoring

To simulate a refactoring you must first choose ”Simulate Refactoring” in the top menu. Now you will see a new window called ”ParameterInputView”. As the title suggests, here you should input some parameters that the tool will need to generate suitable candidates to be refactored. The TopK parameters should represent the top most percentile of classes for each parameter. For instance, if the TopKCRSS is set to 20, then the tool will generate candidates that is among the top 20 percentile concerning CRSS(reachability). The main class should be the entry point class in the application. The easiest way to set the main class is right clicking on a node(class) and select ”Set as Main Class”. The sorting priority tells the tool what priority it should select candidates after. Next you will have three options. System Refactoring, which tries to refactor the system as a whole. When choosing system refactoring you will have to input the number of refactorings. Cycle Refactoring, refactor specific cycles that are present in the code structure. By choosing cycle refactoring you will need to choose which cycle to refactor. After choosing which cycle, you must choose which edges in the cycle that should be refactored(double click on the select column to choose all edges). Usually , selecting all these edges will give the best result. Single Component Refactoring, is used to refactor one component. Single component refactoring will present you with a list of potential candidates, and you must choose which you want to refactor(by double clicking on the select column you will choose all components).

B.4 Simulation Results

After the tool has simulated a refactoring, a simulation result view will appear. This view will present several metrics concerning the overall software quality of the solution. The mean reachability and standard deviation for the reachability of all components, before and after refactoring, are shown. The number of edges and the number of components, before and after, are also presented. On the left there will be a histogram with reachability data from the simulation. Under each column there will be a range. This range represents the reachability of the components. Then the actual value of the column will represent how many components whose reachability is within that range. If the result is satisfactory, then you can click ”Add to Report”, which will update the model and add the result to the report, which I will explain later. If the result is not satisfactory then choose ”Dispose Result” or simply exit the view.

B.5 Report and Instructions

When a result has been added, first you should update the view. This is done by clicking ”Recalculate” in the menu. In order to view the steps required in the code to get the situation that is now represented in the graphical view, you must access the report by clicking ”Show Report”. This will open up a new window. This window includes four different columns: A source component, which is the node that will be manipulated. An action that should be applied to the source. An optional target will represent the node that the action can be performed upon. And lastly the strategy represents the strategy that was applied to the source.

Appendix C

Algorithms

C.1 Variable Identifiers

```
private void AddIdentifier(string name, string value,bool isMethod){
    if (!_currentIdentifiers.ContainsKey(name)) {
        _currentIdentifiers.Add(name, value);
        if (isMethod) {
            if (_methodIdentifiers.ContainsKey(name))
                _methodIdentifiers.Remove(name);
            _methodIdentifiers.Add(name, value);
        }
    }
    else if(isMethod) {
        string oldValue = _currentIdentifiers[name];
        if(!_overridenIdentifiers.ContainsKey(name))
            _overridenIdentifiers.Add(name, oldValue);

        _currentIdentifiers.Remove(name);
        _currentIdentifiers.Add(name, c);
        if (_methodIdentifiers.ContainsKey(name))
            _methodIdentifiers.Remove(name);

        _methodIdentifiers.Add(name, c);
    }
}
```

Figure C.1: The method that adds an identifier to the dictionary of identifiers

C.2 Dependency Calculation

```
/// <summary>
/// This algorithm loops through every CodeReference found in a class
/// and finds its corresponding Class in a Dictionary of classes.
/// Every CodeReference includes a Class where only the name is
/// is set. This name is gathered from the parser and is set to the
/// identifier name of a Node in the abstract syntax tree.
/// The full name of the class is then calculated based on the
/// using statements and the current namespace the identifier node
/// is found in.
/// </summary>
public void CalculateDependencies()
{
    _classList = _model.ClassList;
    if (_classList.Count <= 0)
        return;

    foreach (BaseClass c in _classList.Values)
    {
        List<CodeReference> newReferences = new List<CodeReference>();
        foreach (CodeReference r in c.References)
        {
            FindDependency(c,newReferences,r);
        }
        c.References = newReferences;
        _model.FinaliseClass(c);
    }
}
```

Figure C.2: The method that calculates the dependencies

```
private void FindDependency(BaseClass c,
                            List<CodeReference> newReferences,
                            CodeReference r)
{
    Class c2 = (Class)r.Target;

    if (c2.Namespace == null)
    {
        string namesp = "";
        string name = "";
        string[] arr = SplitNamespace(c2.Name);
        string[] namespaces = null;

        if (arr == null) //c2.Name only contains the Class name,
        //so the name is either from using statements or within
        //the namespace of c, or in the global namespace
        {
            name = c2.Name;
            if (c.Namespace.IsClassInNamespace(name))
            {
                string fullName = c.Namespace.Name
                                        + "." + c2.Name;
                if (c.FullName == fullName)
                    return;
                newReferences.Add(AddEdge(c, fullName, r));
                return;
            }
            else if (_namespaceList[GLOBAL].IsClassInNamespace(name))
            {
                string fullName = _namespaceList[GLOBAL].Name
                                        + "." + c2.Name;
                if (c.FullName == fullName)
                    return;
                newReferences.Add(AddEdge(c, fullName, r));
                return;
            }
            else
            {
                namespaces = c.FullName.Split('.');
```

Figure C.3: The method that finds the full class name for a referenced class inside another class, and adds it to the list of dependencies.

```
            if (namespaces != null)
            {
                string acc = "";
                foreach (string s in namespaces)
                {
                    if (acc == "")
                        acc = s;
                    else
                        acc += '.' + s;

                    CodeReference temp;
                    if ( (temp= CheckNamespace(acc, c, c2.Name, r))
                                                != null)
                    {
                        newReferences.Add(temp);
                        return;
                    }
                }
            }
            if (c.GetType() == typeof(PartialClass))
            {
                CheckNamespaces(c, c2.Name, r, newReferences,
                                r.SourceFile.UsingNamespaces);
            }
            else
            {
                Class cl = (Class)c;
                CheckNamespaces(c, c2.Name, r, newReferences,
                                cl.SourceFile.UsingNamespaces);
            }

        }
    }
```

Figure C.4: Continuation of figure C.3

```
        else
        {
            name = arr[1];
            namesp = arr[0];

            namespaces = namesp.Split('.');
            string temp = c.Namespace.Name + "." + namesp;

            CodeReference cr;
            if ((cr = CheckNamespace(temp, c, name, r)) != null)
            {
                newReferences.Add(cr);
                return;
            }
            else if (namespaces != null)
            {
                string acc = "";
                foreach (string s in namespaces)
                {
                    if (acc == "")
                        acc = s;
                    else
                        acc += "." + s;

                    CodeReference cr2;
                    if((cr2=CheckNamespace(acc,c,name,r)) != null)
                    {
                        newReferences.Add(cr2);
                        return;
                    }
                }
            }
        }
    }
}
```

Figure C.5: Continuation of figure C.4

```
private void CheckNamespaces(BaseClass c,string refClassName,
                             CodeReference r,List<CodeReference> l,
                                             List<string> namespaces)
{
    foreach (string ns in namespaces)
    {
        CodeReference temp;
        if ((temp = CheckNamespace(ns, c, refClassName, r)) != null)
            l.Add(temp);
    }
}
```

Figure C.6: A method that defines some reusable code. Is used within the method FindDependency()

```
private CodeReference CheckNamespace(string ns,BaseClass c,
                                     string refClassName,CodeReference r)
{
    if (_namespaceList.ContainsKey(ns) &&
        _namespaceList[ns].IsClassInNamespace(refClassName))
    {
        string fullName = _namespaceList[ns].Name + "." +refClassName;
        if (c.FullName == fullName)
            return null;
        return AddEdge(c, fullName, r);
    }
    return null;
}
```

Figure C.7: A method that defines some reusable code which is used within the method FindDependency()