Marea

A Tool for Breaking Dependency Cycles Between Packages

Master Thesis

Bledar Aga

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

27. Januar 2015

Prof. Dr. Oscar Nierstrasz
MSc. Andrea Caracciolo

Software Composition Group
Institut für Informatik und angewandte Mathematik
University of Bern, Switzerland

Abstract

Placing classes, methods, dependencies in wrong packages may generate architectural problems such as dependency cycles. Developers, maintainers and testers very often have to deal with dependency cycles that compromise the modularity of their systems, prevent proper reuse, increase the cost of maintenance and increase the cost of tests. We argue that reengineers do not have adequate tools that support removing the dependencies forming cycles at the package level.

We propose Marea, a tool that helps reengineers maintain their object-oriented systems without cyclic dependencies between packages. In our approach, we analyse object-oriented systems by detecting and suggesting to the user which refactoring operations should be used to remove undesirable dependencies. Marea suggests the best sequence of refactoring operations based on the results of a model-based simulation. Moreover, the best path is identified by applying a customised profit function adapted to the user's needs. Our approach has been validated on real-world Java open source systems.

Keywords: Dependency cycles, package cycles, refactoring simulation, software analysis, software quality.

Contents

1 Introduction

In an object-oriented paradigm, we use classes and packages to build structured software systems. A class represents a template for creating or instantiating specific objects within a program. The relationships between classes represent how classes and objects are interconnected together. These interconnections are also called dependencies since a class is connected to another and it is also dependent on it. Software developers organise classes into packages to reduce the complexity of large systems. The dependencies that start from a package and return back to the same package result in cyclic dependencies. For example, if package *A* depends on package *B*, package *B* depends on package *C* and package *C* depends on package *A* then packages *A, B* and *C* are in a cyclic dependency. Dependency cycles arise from bad design, maintenance and evolution of the software. Activities such as placing classes in wrong packages, adding and removing new classes, methods and properties lead to dependency cycles. In software engineering dependency cycles are considered bad because they break the Acyclic Design Principle (ADP)[1], increase the cost of various maintenance tasks and the cost of running tests, compromise the modularity of the system as well as prevent proper reuse.

There are various tools for detecting, visualising and removing cycles in Java systems. In the literature of the analysis of Java systems, cycle detection and visualisation use different approaches such as, cycle detection when they arise, long cycles detection among classes and visualisation of the dependencies among cycles with minimum feedback arc set[2]. Concerning cycle removal one can find best practices for identifying package layered structure and methods refactoring proposed by Fowler et al.[3] such as, *Move Class (MC)* and *Move Method (MM)*. *MC* and *MM* are common methods for removing cycles but on the other hand they may cause unbalanced packages because

dependencies are moved in one direction. Most of the tools only support *MC* and *MM* and do not provide multi-scenario simulation. We will endeavour to solve this problem by introducing refactoring strategies based on the Dependency Inversion Principle(DIP)[1]. The use of the new strategies remove undesirable dependencies and also make the systems loosely coupled. A multi-scenario simulation is an approach for simulating the possible refactoring strategies in order to compare between them and choose the best ones. As a result a tree is created where each node corresponds to a refactoring operation. A profit is calculated in every node based on the customised profit function. This approach minimises the overall effort of cycle removal by offering an automated support tool which can guide the user to choose the optimal sequence of refactoring operations. Our approach, Marea, suggests the best refactoring strategy to the user according to the customised profit function.

1.1 Solution in a Nutshell

In this thesis we propose a tool, called Marea to help the developers maintain their object-oriented systems without cyclic dependencies and facilitate the life of testers. To break the cycles we offer the reengineers suggestions on how to remove undesirable dependencies using several refactoring methods. These suggestions are based on multi-scenario simulation. We are considering object-oriented systems implemented in different programming languages like C++, Java and Smalltalk. Our approach is applicable to software that uses the cited languages, however our tool has been evaluated for Java systems. The main features of Marea include cycles detection, refactoring strategies simulation, creating a decision tree for the possible refactoring methods and suggesting the best applicable refactoring to the user as well.

1.2 Benefits

The benefit of using our tool is that the changes are not made directly to the source code. Instead, we build a model that reflects the main structural characteristics of a target system in a separate platform called Moose[1]. In this way, the reengineers will keep control on their original systems and execute the refactoring process by following the analysis phase performed in Marea. Avoiding a fully automated refactoring process benefits the reengineers to remove the cycles from their systems without any changes in functionality. Furthermore, using Marea can benefit reengineers to maintain systems without structural problems, having suggestions for solutions and improving the software quality.

[1] `http://www.moosetechnology.org`

1.3 Outline

This thesis is structured as follows: In Chapter 2 we provide the definition, the source and the cost of dependency cycles. Following that we justify the need for a semi-automatic tool for removing cycles. In Chapter 3 we review other existing solutions and describe their limitations. In Chapter 4 we present our approach and solution in detail. In Chapter 5 we show the results of applying Marea in a real world Java projects. Finally, Chapter 6 summarises our work and addresses possible future work.

2
Cyclic Dependencies

In this chapter we start with terminology, later we describe which kind of dependencies can be encountered in a system and give a definition of dependency cycle. We answer the questions: What is the origin of dependency cycles? and: Why are dependency cycles bad? We also show the need for a semi-automatic tool for breaking dependency cycles.

2.1 Terminology

In this section we define the terminology used to describe our solution.

Strongly Connected Component (SCC). In graph theory, strongly connected components are graphs where every node (in our case, packages) is reachable from every other node. In Figure 2.1, all nodes are in a single SCC. We use Tarjan's algorithm[4] to detect the strongly connected components.

Cycle. It is a circular dependency between two or more packages where the nodes do not appear more than once in the sequence of the path that forms the cycle. In Figure 2.2 we notice three different cycles: A-B-E, A-B-C and C-D. Cycles are of two types:

- *Direct Cycle* is a cycle with two nodes depending on each other. Figure 2.2 shows the direct cycle C-D.
- *Indirect Cycle* is a cycle with more than 2 nodes. The cycles A-B-E and A-B-C are indirect cycles as shown in Figure 2.2.

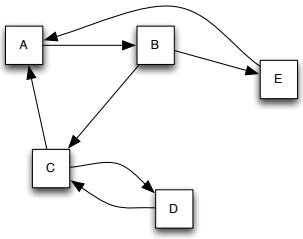

Figure 2.1: Strongly Connected Component.

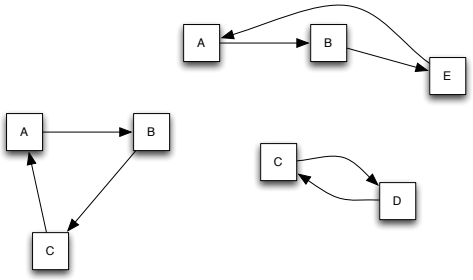

Figure 2.2: Three cycles of the same Strongly Connected Component.

Concrete dependency (CD). A reference dependency or an inheritance dependency or a method invocation dependency.

Logical dependency (LD). A conceptual dependency between two packages that contains the concrete dependencies.

Shared dependency (SD). A logical dependency that is presents in at least two cycles. In Figure 2.2 , the logical dependency between A and B is shared by the two cycles A-B-E and A-B-C.

Depth-first search (DFS). DFS is a graph traversal technique that starts from an arbitrary node (root in case of a tree) and explore as far as possible along each branch before backtracking.

2.2 Types of dependencies

Before defining what we mean by dependency cycles, we describe the types of dependencies considered in our approach. We have 3 different kinds of dependencies: reference, inheritance and invocation dependencies. To explain these kinds of dependencies, we suppose that we have a class named *Button* and a class named *Light*. The light turns on or off if the button is pressed.

- **Reference Dependency**. There is a reference dependency from the *Button* class to the *Light* class if the *Button* class contains an attribute or a local variable or a parameter or a return type of the *Light* class. This subcategory of dependencies are described in the examples below:

  - *Class Variable Dependency* is caused by an attribute of the *Button* class with the type of the *Light* class.

```
public class Button {
   private Light light;
}
```

  - *Initialised Class Variable Dependency*[1] is caused by an initialised attribute of the *Button* class with the type of the *Light* class.

```
public class Button {
   private Light light = new Light();
}
```

  - *Local Variable Dependency* results from a variable with method scope in the *Button* class with the type of the *Light* class.

```
public class Button {
   public boolean checkLight() {
      Light light;
      ...
   }
}
```

  - *Initialised Local Variable Dependency* results from an initialised variable with method scope in the *Button* class with the type of the *Light* class.

```
public class Button {
   public boolean checkLight() {
      Light light = new Light();
      ...
   }
}
```

  - *Parameter Dependency* results from a parameter of a method of the *Button* class with the type of the *Light* class.

[1] We separate the initialised and not initialised dependencies because, as we show in Chapter 4, we use different refactoring strategies to break them.

```
public class Button {
      private Light light;

      public void setLight(Light light){
         this.light = light;
      }
}
```

– *Return Type Dependency* results from the return type of *Light* of a method in the *Button* class.

```
public class Button {
      private Light light;

      public Light getLight(){
         return light;
      }
}
```

- **Inheritance Dependency.** There is an inheritance dependency from the *Button* class to the *Light* class if the *Button* class is a direct subclass of the *Light* class, or the *Button* class implements the *Light* interface (in case of *Light* is an interface).

```
public class  ColoredButton extends Button{
   ...
}

public class Light implements Switchable{
   ...
}
```

- **Method Invocation Dependency.** There is a method invocation that goes from the *Button* class to the *Light* class if there is a method in the *Button* class invoking a method of the *Light* class.

```
public class Button {
   Light light;
   public void press {
      light.turnOnOff();
      ...
   }
}
```

2.3 Dependency Cycles at the Package Level

Let us explain how dependencies result in a dependency cycle. In the following example, Listing 1 and Listing 2 present a control panel that has a set of buttons for turning on and off the lights. Pressing a button will turn the light on or off depending on the state of the light. Each button is controlled by a control panel .

ControlPanel is part of the package *control* and has the following dependencies towards the package *components*: class variable dependency, return type dependency, parameter and method invocation dependency.

```
package control;
import components.Button;

public class ControlPanel {
   private Button buttons[]; //class variable dependency
   private String serialNumber;

   public Button[] getButtons() { //return type dependency
      return buttons;
   }
   public void setButtons(Button[] buttons) { //parameter dependency
      this.buttons = buttons;
   }
   public String getSerialNumber() {
      return serialNumber;
   }
   public void setSerialNumber(String serialNumber) {
      this.serialNumber = serialNumber;
   }
   public int getIndex(String buttonSerialNumber){
      //find and return button index
   }
   public void pressButton(String buttonSerialNumber){
      buttons[getIndex(buttonSerialNumber)].press(); // method invocation depender
   }
}
```

Listing 1: The package *control* with 4 dependencies on the package *components*.

Button is part of the package *components* and is dependent on the package *ControlPanel* because of the following dependencies: class variable dependency, parameter and method invocation dependency.

```
package components;
import control.ControlPanel;

public class Button {
   private ControlPanel controlledBy; //class variable dependency
   private Light light;
```

```
    private String serialNumber;

    public Button(ControlPanel controlledBy) { //parameter dependency
       this.controlledBy = controlledBy;
    }
    public String getSerialNumber() {
       return serialNumber;
    }
    public void setSerialNumber(String serialNumber) {
       this.serialNumber = serialNumber;
    }
    public String controlledBy(){
       return controlledBy.getSerialNumber(); //method invocation dependency
    }
    public void press(){
       light.turnOnOff();
    }
}
```

Listing 2: The package *components* with 3 dependencies on the package *control*.

As we mentioned, *ControlPanel*, and consequently *control*, has 4 dependencies towards *components* and *components* has 3 dependencies towards *control*. This creates a circular dependency or dependency cycle at the package level.

2.4 The Origin of Dependency Cycles

Classes are necessary but insufficient to keep the code organised, for this we use the Principles of Package Architecture[1]. The question arises: How do we choose which classes belong to which packages? The wrong placement of the classes breaks the design of the system. However, there are three Package Cohesion Principles[5] that help the software engineers deal with this matter:

- The Release Reuse Equivalency Principle (REP)[5]: the granule of reuse is the granule of release. This principle suggests to software engineers to group reusable classes together into packages, hence any release will have a version of reusable elements. When the author upgrades the system with the new release, the users should be able to use the old versions. Therefore, a good criterion is grouping reused classes into packages.

- The Common Closure Principle (CCP)[5]: classes that change together, belong together. The more packages change in any release, the greater the job is to rebuild, test and deploy the release. In this case, it is better to keep classes that change together and minimise the work for any release.

- The Common Reuse Principle (CRP)[5]: classes that are not reused together should not be grouped together. When a package changes, all the clients of that package must revalidate even if they do not use anything within the package that has changed. This principle recommends that classes that aren't reused should not belong with classes that are reused since any change in the reused group would cost a lot for the clients that depend on the not reused classes.

Software engineers may fall into structural problems even if they follow these three design principles because these principles are mutually exclusive and cannot always be simultaneously satisfied. The REP and CRP principles help reusers, whereas the CCP helps maintainers. The CCP strives to make packages bigger because when all classes are in one package, the only one package will never change[1]. The CRP tries to make packages smaller because in any release the classes that are not reused are grouped in new packages[1].

When software evolves, it becomes bigger and more complex to understand. Adding a new class, method or anything that creates a dependency in the wrong place will generate structural problems for the system such as cyclic dependencies.

In maintenance the changes of the software are more concentrated on keeping the availability of the software alive then keeping the design clean. Let us explain better with the following example. Suppose we are maintaining a system for a big company of insurance and there is a bug in production. The first think we care is to brink the system alive fixing the bug with the fastest solution that we have. On the other hand, fastest solutions may not be the best choose for the design. Therefore, there is a high risk of introducing cycles.

2.5 The Cost of Dependency Cycles

There are many good reasons why to keep our systems free of cyclic dependencies. Let us clarify it with the below example.

Figure 2.3 left side, shows a package diagram taken from the JHotDraw framework[2]. Suppose that the engineers work on the *figures* package and deliver a new release. Before release, they have to build it with the latest version of *standard* and run some tests. In this case, *figures* does not depend on other packages so the engineers can test and release with a minimal amount of work. In reality, the diagram has another dependency from *standard* $\rightarrow$ *contrib* and the situation changes. Let us consider the right side of the diagram in the Figure 2.3.

In order for the engineers to deliver the *figures* package, they have to test *figures, standard, contrib* and *application* as well. This is a bad practice as the amount of work has been increased due to the dependency *standard* $\rightarrow$ *contrib*. This example has

[2] `http://www.jhotdraw.org`

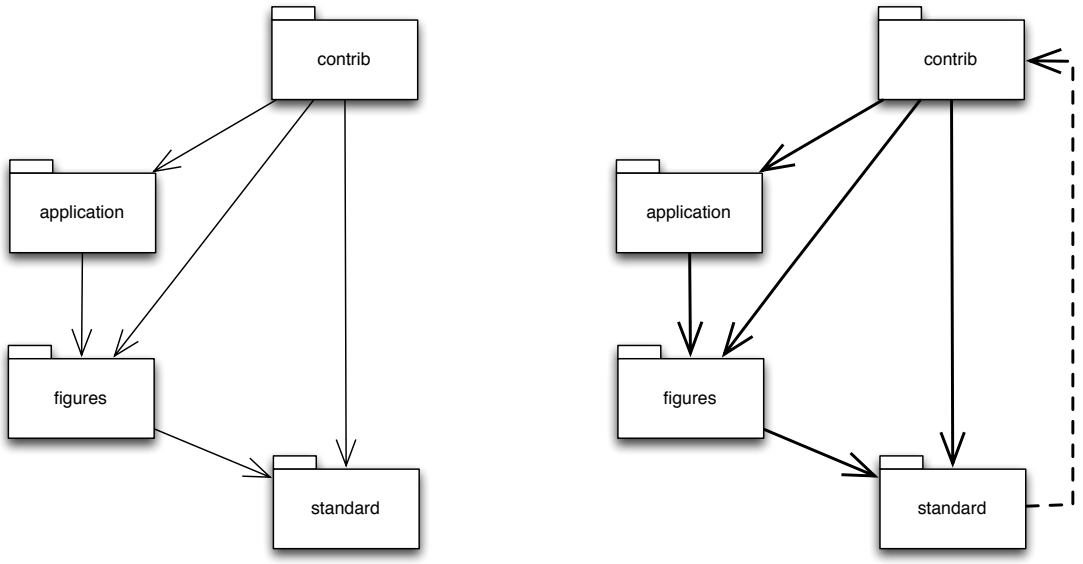

Figure 2.3: JHotDraw package diagram. Acyclic dependencies on the left. Cyclic dependencies on the right.

motivated us to provide the reengineers with a solution to find cycle dependencies and suggest refactoring methods to break them.

In Section 2.4 we introduced one of the reasons why we obtain dependency cycles when classes are not placed in the right package. This reason consequently compromises the modularity of the system and packages are not well organised. Refactoring the code to break the cycles helps to keep the systems well structured.

Our motivation focuses on the package level because we believe that cycles at the class level are not necessarily harmful. Let us consider the following example: Class *Owner* and class *Vehicle*, one owner has many vehicles and a vehicle has an owner. So we expect that the *Owner* class has a collection of *Vehicle* or a method that returns a collection of *Vehicle* and the *Vehicle* class has a property of type *Owner* or method that returns the *Owner*. As we can see in this simple example, there are cases where dependency cycles between classes are necessary and not particularly harmful.

2.6 The Need For a Semi-automated Tool

Our goal is to provide the software engineers with a tool to obtain a good structure of code with good design and architecture. To satisfy engineers' needs we thought of two different solutions such as: a fully-automated tool and a semi-automated tool. We found out that a semi-automated tool is an appropriate one to deal with the cycles. With

semi-automated, we mean that the tool gets the source code as an input to analyse and interacts with the user by suggesting the best refactoring strategy for breaking the desired or suggested cycles. In contrast with a fully-automated tool, Marea analyses all the refactoring options to break the cycles found in the system.

A semi-automated tool is suitable because the automatic refactoring of the source code may change the complete structure of the system. Besides, it offers the user the choice of refactoring. An automated tool would apply refactoring operations without paying attention to the logical structure of the system which can be in contrast to the expectations of the engineers. We want to suggest the best refactoring operations and conserve the logical structure of the system.

3
Related Work

There are two different categories of solutions related to dependency cycles. The first category is related to tools that detect and visualise cycles. The second refers to tools that detect, visualise and break the cycles. Let us describe them in the next two sections.

3.1 Cycle Detection and Visualisation

The tools presented in this category were developed to detect and display dependency cycles visually. Although these tools do not support cycle removal, they are still of interest to us since they support different ways and algorithms for detecting cycles.

3.1.1 JooJ

JooJ[6] is a tool that specialises in detecting dependency cycles when they arise. It is an Eclipse[1] plugin developed for Java programs that detects code cycles when developers write their code. One of the advantages of this tool is that it prevents the cycles before they are in the system. Most of the other tools, detect and break the cycles afterwards. However, Laval claimed in his thesis[7] that there was no empirical study that validates this approach for removing cycles. Also he claimed that possible selected dependencies should not be removed because they are valid in the domain of the program.

[1] `http://www.eclipse.org`

3.1.2 Jepends

Jepends[8] is another tool which was developed to analyse Java source code in order to identify classes as possible refactoring candidates. The tool detects dependency cycles at class level, particularly long cycles among Java classes in a program. The Jepends tool centres around long cycles between classes because developers should understand every class in the cycle which result in higher cost of maintenance. Similarly, when developers test they should test every class in the cycle. The main objective of this tool is to detect cycles which is regarded as the starting point for refactoring.

3.1.3 STAN

STAN[9] focuses on visualising design, understanding code, measuring quality and reporting design flaws. In terms of dependency cycles, STAN considers the SCC theory in order to visualise and highlight dependencies with the minimum feedback arc set[2]. This approach identifies light edges to remove or reverse in favour of dark ones by selecting a minimum weight set of edges to break a SCC.

3.2 Cycle Removal

In this section we will discuss the tools related to cycle detection and cycle removal. This category of tools is more related to our work since we detect and propose solutions to break the cycles.

3.2.1 Hautus

Hautus presented the Package Structure Analysis Tool called PASTA[10]. The tool analyses the modular structure of Java programs and focuses on avoiding cycles in the dependency graph and layering. While PASTA visualises package structures it also enables refactoring the structure by drag and drop of types and sub packages from one package to another, updating and visualising the UML diagram. PASTA has limited number of refactoring methods (i.e. *MC* and *MM*). Hautus mentioned that the use of interfaces and abstract factories for breaking cycles could be future work. We will see in Chapter 4 that these strategies are implemented in our solution.

3.2.2 Lattix

Lattix[11] is a commercial tool based on Dependency Structure Matrix (DSM)[12] approach and layering organisation. It allows one to visualise, analyse and manage a software system's architecture. Moreover the main function of this tool is analysing the

dependencies between software artefacts in a project. It is also used to measure quality with key metrics. In addition it allows one to change the system architecture and track the changes as well as recalculating quality metrics based on the change.

Laval in his earlier work[7] commented that the Lattix approach does not deal adequately with cycle consideration and the identification of an effective layered structure in presence of cycles. In contrast we have found that the refactors to break the cycles are limited in moving and merging packages.

3.2.3 Structure101

Structure101[13] is another tool that aims at building layer organisation and analysing dependencies for violations of the layered architectural style. The tool uses feedback arc set to highlight the undesirable dependencies presented in a Tangle[2]. It enables developers to use refactoring methods such as *Move Class* and *Move Method* for a better reorganisation of the dependencies. However, the interaction with the developer to remove dependency cycles and refactor methods is minimal.

3.2.4 ECOO

ECOO[7] is an approach that was also developed for large software architectures. It identifies undesirable dependencies and proposes solutions to avoid modularity problem. Laval argued in his work that there was a lack of approaches to perform the following tasks: identifying and solving the problems; avoiding the degradation of the system; and minimising change costs as well. Consequently he developed the following solutions:

- ECELL allows the user to visualise and understand deeply the content of a package dependency.
- EDSM[14] was built around ECELL to provide micro-macro reading. The aim of EDSM is to identify some problems (i.e. dependencies that create cycles) that arise in the package architecture by highlighting cycles between packages. In this way the reengineers could have enough information to make decisions about fixing the undesirable dependencies.
- CYCLETABLE[15] is also a component used by the ECOO approach for visualising information about the cycles in more details. It highlights dependencies that have high impact on cycles by decomposing SCC. This helps the reengineers to remove cycles with minimum effort.

[2]Definition of a Tangle `http://structure101.com/help/java/structure101/Content/xs/tangle.html`

- OZONE[16] provides a semi-automated algorithm operated on EDSM and CYCLYTABLE. It proposes dependencies that can be removed to break the cycles using the layering organisation. The considered dependencies are separated in to two categories: dependencies that are in direct cycles and shared dependencies. Therefore, both categories of dependencies have a high impact of removing the cycles.
- ORION[17] is an interactive tool to simulate arbitrary refactoring operations in source code models. It provides the possibility to simulate the changes of the source code by creating multiple versions of models.

Our work is somehow the continuation of Laval's work. We reuse part of his modules such as: EDMS to detect the dependencies and Orion for simulation. However Laval's work was more focus on visualisation than cycle removal. As we show in Chapter 4, we had to extended Orion with new refactoring methods.

3.3 Limitations of Existing Approaches

Considering all the tools mentioned above has encouraged us to develop a new approach for breaking dependency cycles. All the tools in both categories have provided great approaches for dependency cycles from different perspectives such as visualisation, detecting and breaking cycles. These approaches are limited to particular techniques such as drag and drop for moving classes and methods. Besides, they do not provide a sufficient way to propose the best refactoring technique for breaking the unwanted dependencies. Our current work improves over existing refactoring techniques, simulating the possibilities of refactoring, comparing the source code models using metrics as well as proposing the reengineers the best methods of change for removing the unwanted dependencies.

4
Marea

In nature the cyclic rise and fall of the sea level is caused by the gravitational pull of the sun and moon. This phenomenon is called *Tide* and, translated in Italian, *Marea*. The reason why we called our tool Marea is that, as in the natural phenomenon, we also deal with cycles. In this chapter we discuss in detail the solution that Marea proposes. We start the next section with Overview and then we detail the various phases of Marea.

4.1 Overview

Marea is a semi-automated tool that takes an input extracted from the source code of Java systems. The input is transformed by the Moose framework to a model for analysis. The model complies to the FAMIX[1] meta-model that enables analysis with queries and navigation. Figure 4.1 shows that Marea consists of 4 different phases. We briefly describe these phases as follows:

1. **Analysis.** In the analysis phase Marea detects all the cycles between packages and extracts all the types of dependencies. After extracting the dependencies and linking them with the appropriate cycles, Marea sorts the logical dependencies based on the ratio between shared and concrete dependencies. The idea of sorting the logical dependencies is to consider the first dependencies that allows Marea to break as many cycles as possible with minimal effort.

[1] `http://www.themoosebook.org/book/internals/famix`

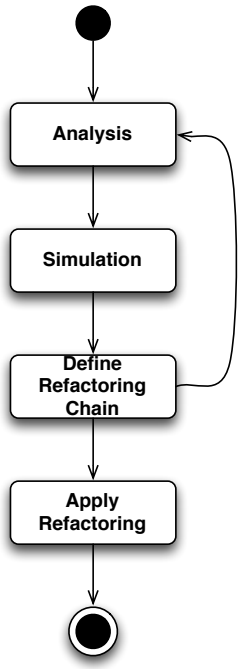

Figure 4.1: Overview of Marea.

2. **Simulation.** As soon as Marea has all the dependencies and finds out which dependency to start from, it simulates the possible refactoring strategies for the selected logical dependency. The logical dependency has one or more concrete dependencies and is part of at least one cycle. Removing a logical dependency implies breaking the cycles where the dependency is present. Because there are many concrete dependencies and each one can be broken in multiple ways, we end up with different sequences of operations. Using one or another has different impact in the number of the cycles, structure of the source code and design of the system. The simulation allows one to choose the best way to refactor the system by considering the minimum impact of change.

3. **Define Refactoring Chain.** For every simulation of breaking a logical dependency we suggest the chain of refactoring operations with minimum impact of change to the user. However, the user chooses what is more appropriate according to his point of view. This phase could also be automated throughout selecting always the tool's suggestions. When the refactoring chain is defined, the user will be able to decide whether to end the execution or continue. If there are more cycles, the process will resume from the first phase.

4. **Apply Refactoring.** As explained earlier, Marea is based on the Moose framework

and no change is reflected in the source code. Actual refactoring can be performed in an additional complementary phase. We produce a tree containing the simulated actions for each logical dependency. Considering that the user may break many logical dependencies, there are many decision trees. These trees can be imported as an input file for a tool (e.g. an plugin Eclipse), that will apply the real refactoring to the source code.

4.2 Analysis

The starting point of Marea is the analysis phase which includes: cycle detection, extraction of dependencies, building the data model that holds essential data (cycles, logical dependencies and shared dependencies) and sorting the logical dependencies based on the ratio between shared and concrete dependencies.

4.2.1 Cycle Detection

For cycle detection we use Tarjan's algorithm[4]. This algorithm is based on depth first search (DFS). DFS starts from an arbitrary node and traverses every node of the graph only once. The traversed nodes are indexed in order they are discovered. While returning from the recursion of DFS, every node is assigned with the least index that can be reached from the node that DFS is considering. Nodes with the same assigned index (while returning from the recursion) are located in the same strongly connected component. Figure 4.2 provides an illustration of Tarjan's algorithm. This algorithm is an efficient method to detect SCCs in linear time as a function of the number of nodes and edges of a graph, i.e. $O(|V| + |E|)$.

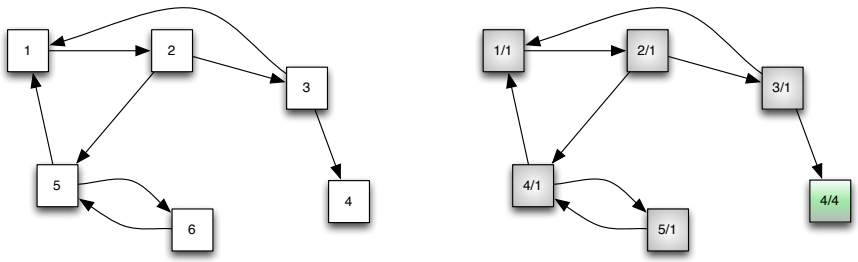

Figure 4.2: Left side: Nodes are indexed as they are explored by DFS. Right side: Nodes are re-indexed at the return of the recursion of DFS, those with the same assigned index are in the same SCC.

A Java input project with cycles is displayed in Figure 4.3. We consider the SCCs disconnected because the dependencies that connect the SCCs are out of scope since they

do not form cycles. An SCC is composed of one or more cycles and if a cycle exists in an SCC, it does not exist in another SCC. We extract the cycles from an SCC using the algorithm that checks for each edge of the SCC if there is a cycle and, if there is, it returns for each edge the smallest cycle. Bigger cycles are ignored because breaking the smaller cycle will automatically break the bigger one. Breaking all the cycles in a SCC will make the SCC disconnected. Based on graph theory disconnecting all the SCCs makes the entire graph acyclic and this is our goal, to eliminate cyclic dependencies into an input project.

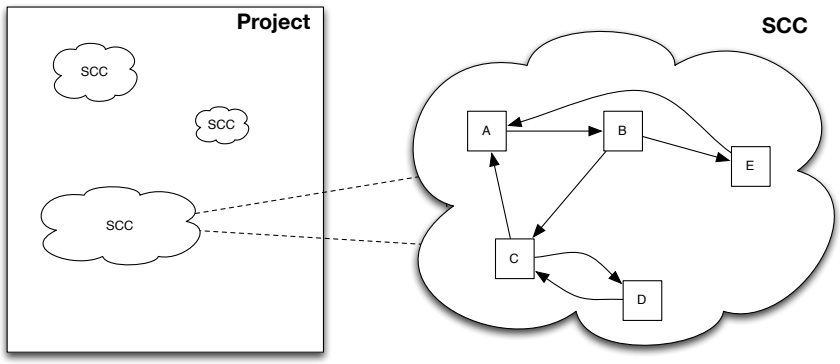

Figure 4.3: An input project seen as a set of strongly connected components.

4.2.2 Extraction of Dependencies

As shown in Figure 4.4, Marea extracts the logical dependencies from each cycle within an SCC. A logical dependency is composed of the concrete dependencies between 2 packages. The concrete dependencies are present in different classes and are extracted to form every logical dependency. Furthermore, the logical dependencies that are present in many cycles are considered as shared dependencies.

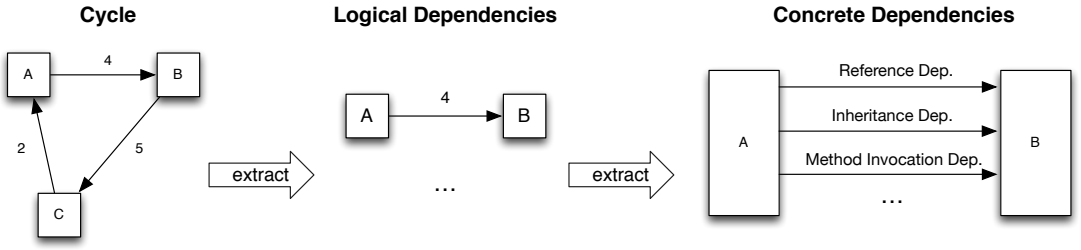

Figure 4.4: Extraction of logical dependencies from a cycle.

4.2.3 Build the Data Model

The Moose framework and the Famix meta-model allow us to query an input model and navigate over the entities of the model. For instance, if we need all the outgoing dependencies of a class we can use the method *queryOutgoingDependencies* from the *Famix-Core* package. Famix provides us the way to extract the information from a model. However, we have to organise the data based on our needs. Considering the example above we have all the outgoing dependencies but not their types. Therefore, in order to have exactly what we need we have to extract more information and build a separate data model. In this project, we designed our data model representing the information of our scope. For the analysis phase we implemented the algorithm that extracts the needed data using Famix queries and part of the DSM component implemented by Laval.

4.2.4 Sorting Logical Dependencies

The last step of the analysis phase is to prepare the input for the simulation phase. Now we have all the cycles with all the dependencies, but the question is: Which dependency can we first break for cycle removal? The answer to this question is to let the user choose a starting point. In this case the user decides which undesirable dependencies should be removed for breaking a cycle so he knows what he is doing. Otherwise, a random choice may produce an inefficient output.

A logical starting point for breaking a cycle is to consider shared dependencies because they are present in at least two cycles. Even though this process may sometimes introduce new cycles, breaking a shared dependency will directly break the exact amount of cycles the dependency is part of. New cycles are introduced because moving a dependency from one package to another may generate new ones somewhere else. Let us reconsider the package diagram in Figure 4.5. The logical dependency *standard* $\rightarrow$ *contrib* is shared between the cycles *standard* $\rightarrow$ *contrib*, *standard* $\rightarrow$ *contrib* $\rightarrow$ *figures* and *standard* $\rightarrow$ *contrib* $\rightarrow$ *application* $\rightarrow$ *figures*. Suppose that *standard* $\rightarrow$ *contrib* contains a concrete dependency of class variable and the package *figures* depends on the class that contains the class variable dependency. Breaking the logical dependency *standard* $\rightarrow$ *contrib* using *MC* operation will break the cycles where the *standard* $\rightarrow$ *contrib* is shared, but it will also introduce new cycles such us *figure* $\rightarrow$ *contrib* and *figure* $\rightarrow$ *contrib* $\rightarrow$ *application* because the package *figure* has a new dependency towards the package *contrib*.

However, in some instances, shared dependencies may contain a large number of concrete dependencies. Hence, breaking a shared dependency at this point will increase the amount of work needed compared to a shared dependency that has few concrete dependencies. Consequently, shared dependencies are not a good practice. To solve this matter we need to consider the number of concrete dependencies present in a logical dependency. Our tool addresses this matter by first considering the logical dependencies

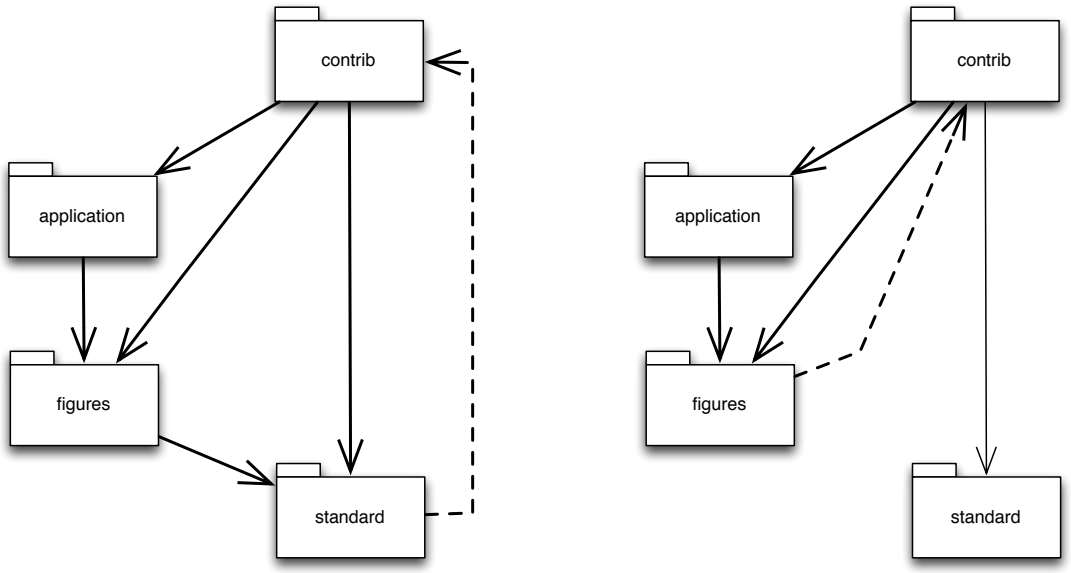

Figure 4.5: Breaking a LD introducing of new cycles

with the highest ratio between the number of shared dependencies and the number of concrete dependencies as explained in the following formula:

$$\frac{|\text{ Shared Dependencies }|}{|\text{ Concrete Dependencies }|} \tag{4.1}$$

4.3 Simulation

We have seen in the analysis phase how Marea proposes the input for the simulation. We already know that the input is a logical dependency with one or more concrete dependencies. In this phase we explore the possible options for breaking the logical dependency with different refactoring strategies. The removal of the logical dependency breaks all the cycles where the logical dependency is present.

In general we have an output from the simulation phase that contains different ways to break the logical dependency since we use different strategies to remove several concrete dependencies. During the simulation Marea generates a decision tree for the possible ways of refactoring. As we see in the next phase, the user can choose the best way of refactoring from the tree. Consequently this section is organised into 2 subsections: Refactoring Methods and Decision Tree.

4.3.1 Refactoring Methods

For breaking dependencies between packages, 4 types of refactoring are utilised:

1. **Move Class (MC).** This refactoring method moves a class from one package to another. Figure 4.6 shows how class variable dependency is broken when moving the *Button* class from package *control* to *components*. As a result, *Button* will be part of *components* instead of *control*).

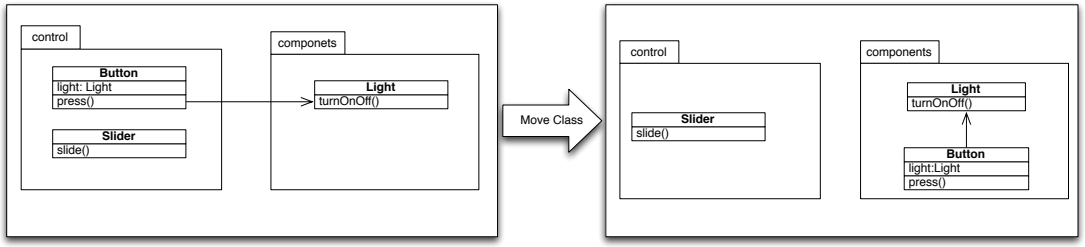

Figure 4.6: Move Class refactoring method.

2. **Move Method (MM).** This refactoring method moves a method from one class to another. In our case the method is moved between two different packages since we consider the cycles at the package level. Move method is used to break dependencies related to a method of a class. For instance, the invocation dependency can be broken by moving the method *press(Light light)*{...} from *Button* to *Light* (Figure 4.7).

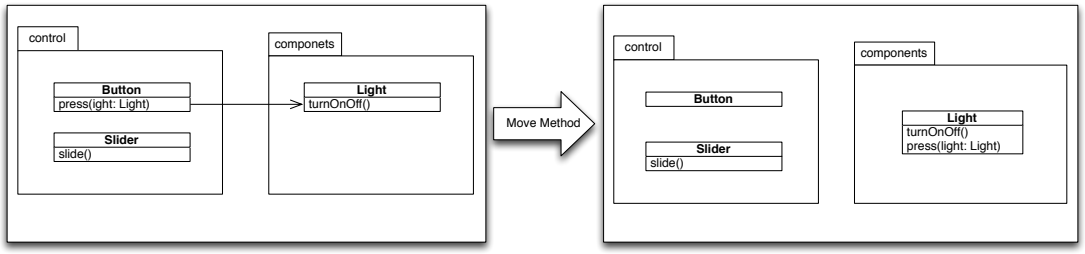

Figure 4.7: Move Method refactoring method.

3. **Abstract Server Pattern (ASP)**[18]. Depend upon Abstractions. Do not depend upon concretions. We use this simple pattern to remove a dependency that depends on a concrete class by changing the target to an interface. Practically, *ASP* allows one to change the direction of the dependence by adding an interface in the

dependent package and implement it from the class depended upon. For instance: in order to break the class variable dependency, we should add the interface *Switchable* to *control* and implement it in *Light* (Figure 4.8). This refactoring method is used in case the reference dependency is not initialised. If it is initialised *ASP* is combined with Dependency Injection as explained next.

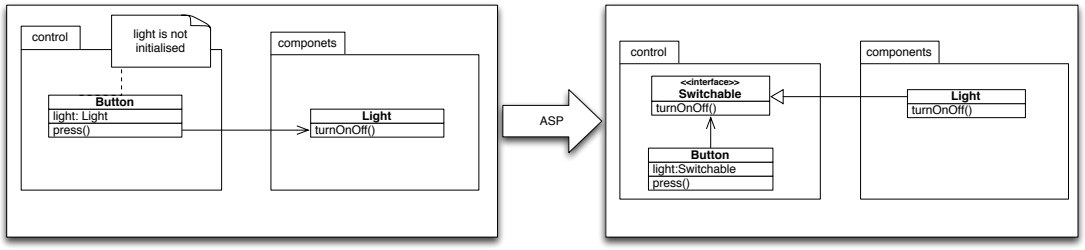

Figure 4.8: ASP utilised for a class variable dependency.

4. **Abstract Server Pattern + Dependency Injection**[19] **(ASP+DI)**. This refactoring method is an extension of the *ASP* refactoring that allows one breaking dependencies where the initialisation occurs. Through *ASP* the dependency changes its type from static to dynamic. In more details, the direction of the dependency changes to the opposite. However, the dependency will persist since the initialisation of the object exists. In order to break the initialisation dependence, we use *DI* pattern. Figure 4.9 shows an example to clarify how this method works. The *Inject* class creates the *Light* object and inject it through the *setLight(light:Switchable)* method. It is not necessary to simulate the injection of the objects because the injector creates dependencies only in one direction and no classes depend on it. We have shown a way how to inject the objects. However, there are frameworks (i.e. *Spring*[2]) that enable engineers to implement the *DI*. We separate *ASP* and *ASP + DI* because they have different implementations in Marea. For the *DI* we have to remove the initialised object dependency.

The refactoring methods that we described above are not applicable for all the types of dependencies. Table 4.1 shows all the possibilities of the use of refactoring per dependency.

In our context *Move Method* and *ASP* are limited in some cases. This is due to the limitations of the Moose framework, the refactoring engine of Eclipse (i.e. not all *MMs* are refactored by Eclipse) and the impossibility of the refactoring method itself. Moose framework is limited because it does not provide all the information that we need to simulate complex refactoring operation such as: the use of factory method, the use

[2] `http://spring.io`

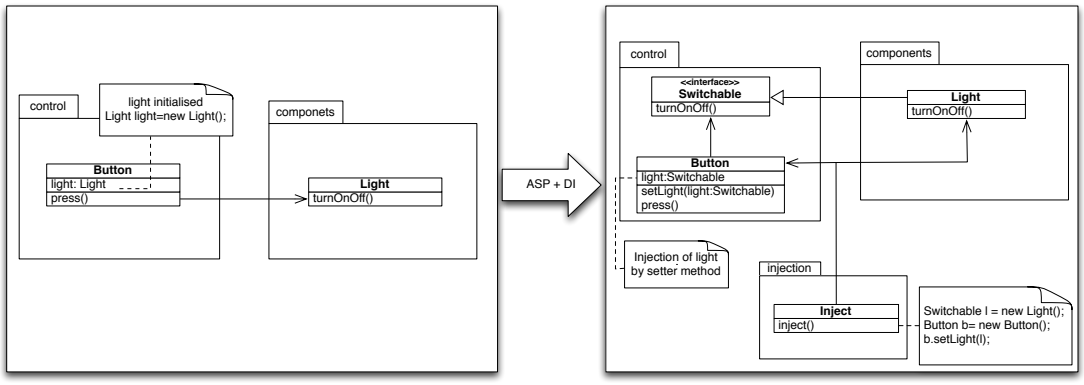

Figure 4.9: ASP + DI utilised for a initialised class variable dependency.

—	Move Class	Move Method	ASP	ASP + DI
Inheritance	Yes	No	No	No
Class Variable	Yes	No	Yes	No
Initialised Class Variable	Yes	No	No	Yes
Local Variable	Yes	Yes (Limited)	Yes	No
Initialised Local Variable	Yes	Yes (Limited)	No	Yes
Parameter	Yes	Yes (Limited)	Yes	No
Return Type	Yes	Yes (Limited)	Yes	No
Invocation	Yes	Yes (Limited)	Yes (Limited)	No

Table 4.1: Applicability of the refactoring methods.

of accessors and the clone of objects (See the examples below). Moose considers the methods as a black box. The use of complex refactoring methods requires to read all the statements of a method and extract assignments and values of objects. To do so, we need to use an external parser to extract the information and change the signature and the content of a method. These operations are not covered in this thesis. According to the *MM* refactoring method it is limited in case the method that moves has one of the following properties:

- **The method is a constructor.** It is not possible to move a constructor of a class to another.
- **The method returns *this*.** If the method that moves returns *this*, then the refactoring is not possible because *this* is bound to the class that contains that method.

- **The method has access to a variable with class-scope**. If the method that moves contains any access to a variable that is different from a parameter or a local variable, then the refactoring is not possible because the access is related to the class that contains the method.
- **The method has an invocation to a static method defined in the same class.** If the method has an invocation to a static method defined in the same class, then the refactoring is not possible.

Regarding the *ASP* refactoring method, it is also limited in the case of the invocation method dependency because of the following points:

- **The invoked method is static.** Static methods are not supported by the interface.
- **The invoked method is a constructor** Constructor is not supported by the interface.
- **The invoked method is the call of the super constructor (i.e. super())**. Constructors are not supported by the interface.

In the perfect world the limited operations could be implemented using other strategies such as: the use of Factory Method Pattern[20] in case the method is a constructor (Listing 3, Listing 4), creating and returning a clone object instead of *this* (Listing 5, Listing 6) and changing the class access variable by using their accessors (Listing 7, Listing 8).

```
package control;
import components.Light;

public class Button {
   private Light light;

   public Button(Light light) {
      this.light = light;
   }
}
```

Listing 3: The *Button* class and its *Button(Light light)* constructor are dependent on the *Light* class.

```
package components;
import control.Button;

public class FactoryButton {
   Button makeButton(Light light){
      Button b = new Button();
      b.setLight(light);
      return b;
   }
}
```

Listing 4: Refactoring the *Button(Light light)* constructor with the factory method in the same package where *Light* is located.

```
package control;
import components.Light;

public class Button {

   public Button getButton(){
      Light light = new Light();
      ...
      return this;
   }
}
```

Listing 5: The *Button* class and the *getButton()* method are dependent on the *Light* class. The *getButton()* method returns *this*.

```
package components;
import control.Button;

public class Light {
  public Button getButton(Button button)
  {
   Light light = new Light();
   ...
   return button;
   }
}
```

Listing 6: Refactoring the *getButton()* method with the *MM* operation. The *getButton()* method returns the *button* parameter instead of *this*.

```
package control;
import components.Light;

public class Button {
   private Light light;
   private Slider slider;
   //setter and getter of slider

   public void press(){
      slider.slide();
      light.turnOnOff();
   }
}
```

Listing 7: The *Button* class and the *press()* method are dependent on the *Light* class. The *press()* method has the *slider* access class.

```
package components;
import control.Button;

public class Light {
   public void turnOnOff(){
      //turn on/off the light
   }

   public void press(Button button){
      button.getSlider().slide();
      turnOnOff();
   }
}
```

Listing 8: Refactoring the *press()* method with the *MM* operation. The *press()* method uses the *getSlider()* accessor to access *slider*.

4.3.2 Implementation of Refactoring Methods

The simulation of the refactoring operations requires one to apply the same changes to the model. The easiest approach is to duplicate the original model for each refactoring operation. This is expensive in terms of memory and time. Laval proposed a solution based on FAMIX, called Orion. Orion allows one to define incremental versions of a model. Only the entities which are directly changed are redefined and copied into the new version and the other entities are simply referenced. A reference table keeps track of all the versions of the entities. The table is copied from the parent model when creating the new version and is modified by actions such as entity creation, entity change and

entity deletion. More details can be found in a publication by Laval et al.[17].

Laval implemented basic operations like add and remove entity as well as more complex operations like *Move Class* and *Move Method*. We extended Orion by implementing the new refactoring strategies *ASP* and *ASP+DI* and we also adapted the existing operations for all the types of the dependencies that we have. The implementation of the new refactoring operations are build on existing version of Orion that you find in the following link[3].

4.3.3 Decision Tree

The simulation starts from a parent model and new versions are generated for each refactoring strategy. Considering the concrete dependencies and the fact that many refactoring strategies can be used for removing each of them, we have a tree of models rooted with the original one.

Let us consider the following example: Suppose we want to break a logical dependency composed of 2 concrete dependencies. It is possible to use 2 strategies for the former and 3 strategies for the latter. By applying 2 strategies to the first dependency we obtain 2 new models corresponding to the nodes at the first level of the tree. The models at the first level will be the parents of the models at the second level and children of the root. As a result, 6 branches of 6 new versions will eventually be generated as a tree of models.

Having different versions for each simulation of a refactoring strategy allows us to compare the actions and calculate a profit for each node. Therefore the tree contains more information and allows us to decide the best path with the best profit. In this way we generate a decision tree for each logical dependency as an input.

The decision tree is a powerful approach for obtaining all the different alternatives of simulated refactoring. As soon as we have all the simulations for a given input the user will be able to choose the path suiting his needs. Figure 4.10 explains the general tree with respect to the generated profit of each action.

Marea does not check for new introduction of dependencies because both cycles and dependencies are reextracted, and the current logical dependency in the simulation is updated in every refactoring operation. Each node of the tree represents a refactoring action for a concrete dependency. For every node we generate a new Orion model by modifying the model corresponding to the parent node. Also, the nodes contain the profit of the single operation that is calculated step by step for all the operations. The profit is calculated considering the root of the tree as the starting reference. The calculation of the profit is based on 3 metrics:

1. **Number of Cycles**. We want to reduce the total number of cycles.

[3] `http://www.squeaksource.com/@PjpQDnUTENDpIwKi/EYeqJcp5`

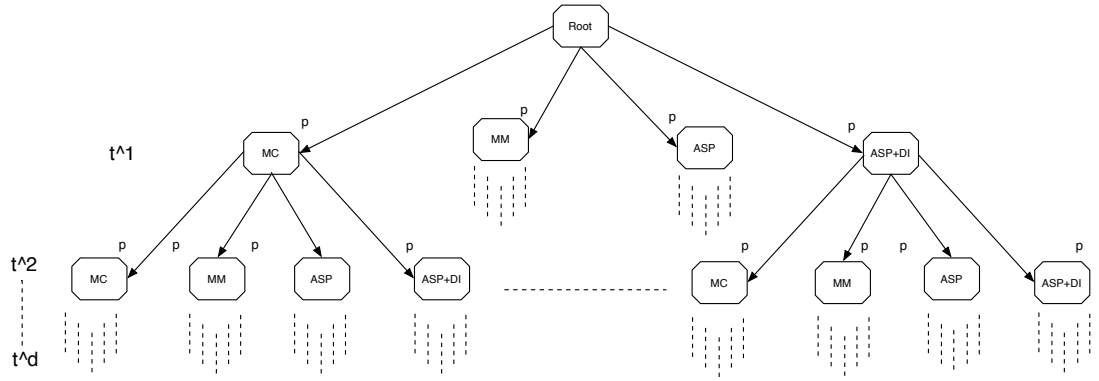

Figure 4.10: Simulation of all the possibilities of refactoring methods.

2. **Tree Depth**. We want to reduce the number of steps (refactoring actions) for breaking the logical dependency.

3. **Quality Metric(s)**. We want to improve the quality of the code.

The first metric concerns reducing the total number of cycles. The second metric is the distance from the root to the node. The third is the quality metric which might use the *instability* metric[21] or the *abstractness* metric[21] or both as our goal is to improve the quality of the software. The stability metric is a consequence of the Stable Dependencies Principle[21], depending on the direction of stability. A package is considered stable if other packages depend on it. On the other hand a package is instable if no other packages depend on it. In order to calculate the instability of a package we use the following formula:

$$I = \frac{\text{Ce}}{\text{Ce} + \text{Ca}} \tag{4.2}$$

where:

Ca, afferent coupling. The number of classes outside the package that depend upon classes inside the package.

Ce, efferent coupling. The number of classes outside the package that classes inside the package depend upon.

I Instability. This metric is calculated by Formula 4.2 that has the range: [0,1]. Value 0 the package is stable, 1 the package is instable. If *Ca* and *Ce* are 0 then *I* = 0 because there are no outgoing and incoming dependencies into the package so the package is stable.

The *abstractness* metric is derived from the Stable Abstractions Principle[21]: stable packages should be abstract packages. Below is the formula for calculating this metric:

$$A = \frac{\text{Na}}{\text{Nc}} \tag{4.3}$$

where:

Nc. Number of classes in the package.

Na. Number of abstract classes in the package. Remember, an abstract class is a class with at least one pure interface, and cannot be instantiated.

A. Abstractness is calculated by Formula 4.3 and has the range: [0,1]. Value 0 means that the package contains no abstract classes and 1 means that the package contains only abstract classes. If *Nc* is 0 then *A* = 0 because the package is empty, thus no abstract classes.

To satisfy our requirements we favor nodes (remember each node represent a simulated refactoring operation) with an instability metric near to 0 (we want to obtain stable packages) and abstractness metric near to 1 (stable packages should be abstract packages). On the other hand we penalise the nodes that have a large number of operations and the nodes with a large number of cycles. The refactoring operations involve 2 packages, from and to. For this reason we calculate the average of instability and abstractness between those 2 packages. Formula 4.4 calculates the profit based on the facts above:

$$P = w_c \times \frac{1}{\text{\# cycles + 1}} + w_d \times \frac{1}{\text{depth}} + w_i \times \frac{\text{(1- } I_{from}\text{) + (1- } I_{to}\text{)}}{2} + w_a \times \frac{(A_{from}) + (A_{to})}{2} \tag{4.4}$$

where:

w_c is a constant weight for the cycles.

w_d is a constant weight for the depth.

w_i is a constant weight for the instability.

w_a is a constant weight for the abstractness.

Let us reconsider the example of Section 2.3 for calculating the profit of the logical dependency *components* → *control*. Remember that in this example there is one cycle between the *components* and the *control* packages. Marea selected the *components* → *control* logical dependency as an input for breaking the cycle because it has less concrete dependencies than *control* → *components* (3 versus 4 concrete dependencies). Figure 4.11 shows the decision tree with the best profit calculated for the simulation of the refactoring of the 3 concrete dependencies. The best profit is the path on the left side of the tree and it is calculated in Formula 4.5:

$$P = 1 \times \frac{1}{0 + 1} + 1 \times \frac{1}{1} + 1 \times \frac{(1 - 3/4) + (1 - 3/3)}{2} + 1 \times \frac{1/2 + 0/2}{2} \tag{4.5}$$

where:

The *Button* class was moved from *components* to *control* and in *components* the *Switchable* interface implemented by *Light* is present. After the *MC* operation, the number of classes and interfaces located in *components* is 2 and the number of classes located in *control* is 2.

w_c = w_d = w_i = w_a = *1*. In this case the same value is given to all the weighs.

 # cycles = 0. The *MC* operation breaks all the concrete dependencies present in the logical dependency.

depth = 1. The node is one step from the root.

I_{from} = *3/4*. *Ce* = 3 and *Ca* = 1.

I_{to} = *3/3*. *Ce* = 3 and *Ca* = 0.

A_{from} = *1/2*. *Na* = 1 and *Nc* = 2.

A_{to} = *0/2*. *Na* = 0 and *Nc* = 2.

As the best path is chosen based on the profit it is important to point out that the calculation of profit is flexible. The shown formula can be reimplemented based on the user's needs by using other metrics and considerations. This means that there is a possibility to plug different Metrics into Marea if they are implemented.

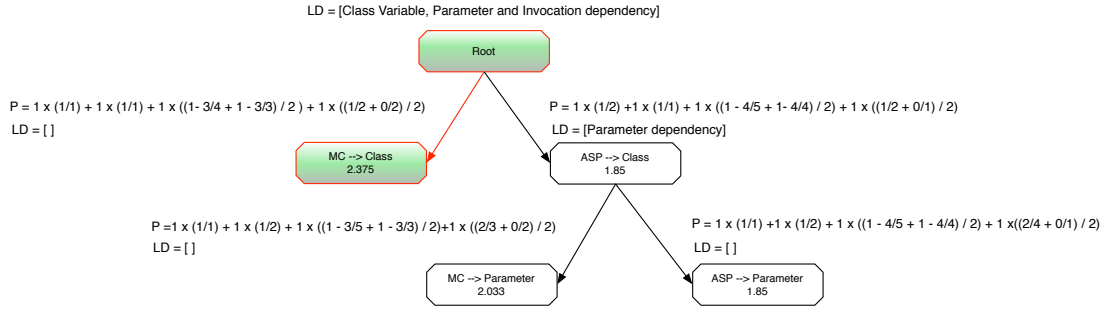

Figure 4.11: Calculation of the profit for the LD *components* → *control*.

4.4 Define Refactoring Chain

The decision tree represents the possible refactoring operations (different paths of the tree) of a certain logical dependency. Each path of the tree is characterised by the profit that is calculated for each node. Because the total amount of the profit is calculated for every refactoring operation, the leaves of the tree contain the significant profit value of the path. The best path is identified by following the steps from the leaf with the biggest value to the root. In our implementation we visualise the decision tree and also highlight the best path for the user. The user has the option whether to select the suggested path or another based on his need. The decision can also be automated by selecting always the best path.

4.5 Apply Refactoring

As mentioned earlier Marea simulates the refactoring operations on the Moose framework from the source code separately. The simulation has the advantage to preserve the original version of the source code. On the other side the desired refactoring operations are not directly applied to the source code. In our case to apply the refactoring chain, the user has to rely on external tools. The most widely used development tool for Java projects is Eclipse and our suggested refactoring operations are compatible with the refactoring operations supported by Eclipse[4], except for *ASP* and *DI* which have to be applied manually. The advantage of this phase is that the developer or maintainer knows exactly what he will change in his projects.

[4]Eclipse supports *Move Class* and *Move Method*. *Move Method* is limited to certain cases that we described in Section 4.3.1

5 The Validation

In this chapter we outline the results of our work. We validate the results of Marea by testing it on three Java projects. The first project is a web-based application that uses the Spring MVC framework[1]. The second project is the Google Web Toolkit (GWT)[2] component, gwt-cal[3]. This project is a web-based calendar component implemented in Java. The third project is an open source project called JHotDraw[4] that is used for creating technical and structured graphics.

5.1 No Presence of Cycles

In this case the considered project is a web based application for renting rooms, developed by bachelor students of the University of Bern. The application uses the Spring MVC framework. Basically Spring implements the principle of Inversion of Control[22] through the Dependency Injection pattern. In addition, the Spring MVC framework implements the MVC architectural pattern. The layers of an application that is built on Spring MVC are well defined and the dependencies between classes will, most likely, go in one direction. Moreover the Dependency Injection reduces the number of unnecessary dependencies. In our test, Marea found no presence of cycles. The correctness of the result was confirmed through manual inspection.

[1] `http://spring.io`
[2] `http://www.gwtproject.org`
[3] `https://code.google.com/p/gwt-cal/`
[4] `http://www.jhotdraw.org`

5.2 Presence of Cycles

In this section we validate Marea with GWT Calendar. The input consists of 27 total packages and 8641 Famix entities. Excluding the system packages like java::* and javax::* we have 14 packages and 92 classes. We ignore the Java API packages because they do not have back dependencies to the implemented packages, and therefore do not form cycles.

Remember that we use Tarjan's algorithm to detect the strongly connected components (SCCs), and that Marea extracts the cycles from each SCC. Considering an SCC, Marea checks for each edge if there is a cycle and, if there is, it returns for each edge the smallest cycle. Bigger cycles are ignored because breaking the smaller cycle will automatically break the bigger one. After this consideration let us start analysing the input following step by step the simulation of cycle removal until we transform the input in a acyclic graph.

At the first step Marea finds 7 cycles formed by 12 logical dependencies all within an SCC. After detecting the cycles and extracting the dependencies, Marea sorts in descendent order the logical dependencies. The first logical dependency is the one that has the biggest ratio between shared dependencies and concrete dependencies. Figure 5.1 shows the graph with the cycles. The logical dependency *client* $\rightarrow$ *monthview* is the one with the biggest ratio. It has the biggest value because is the most shared dependency (amongst 3 cycles, remember that bigger cycles are ignored) and has only one concrete dependency. The concrete dependency is an initialised class variable and, as indicated in Table 4.1, there are two possibilities of refactoring: *MC* and *ASP+DI*. Marea simulates the 2 refactoring methods and as result both methods break the logical dependency and the cycles in which it is involved. At the end of the simulation, Marea returns a tree with 3 nodes (root included) and suggest the *MC* refactoring as it has a higher profit. The profit for the *MC* is higher because it breaks a larger number of cycles than *ASP+DI*. We chose *ASP+DI* as we do not want to change to location of the class even the profit is not the best.

We have seen the first round of simulation and now we are going further to detect again the cycles and the logical dependencies for the next round. Figure 5.2 shows the situation of the graph after Marea has broken the first logical dependency and the correspondent cycles. We can notice that the number of cycles is the same as the previous situation but the cycles are different. Again the logical dependency highlighted is the most convenient since it breaks the biggest number of cycles with the minimum effort (one concrete dependency). This case is very similar to the previous one. The logical dependency has one initialised class dependency and 2 applicable refactoring methods. Like the previous case Marea returns the expected decision tree with 3 nodes. The profit for both nodes is closer although *ASP+DI* is a bit higher. The number of broken cycles is the same for both refactoring methods, as we were expecting. The choice was for *ASP+DI* and now let us consider the next round.

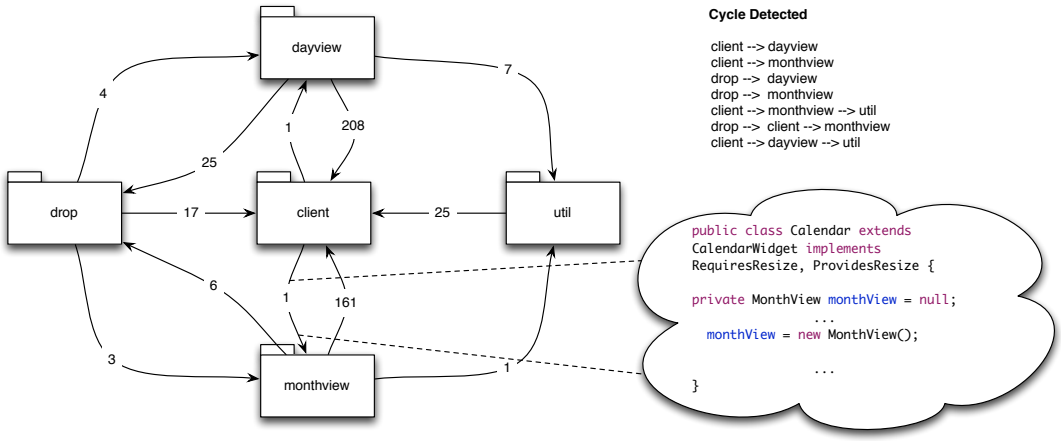

Figure 5.1: Cycles graph with the first LD to be simulated.

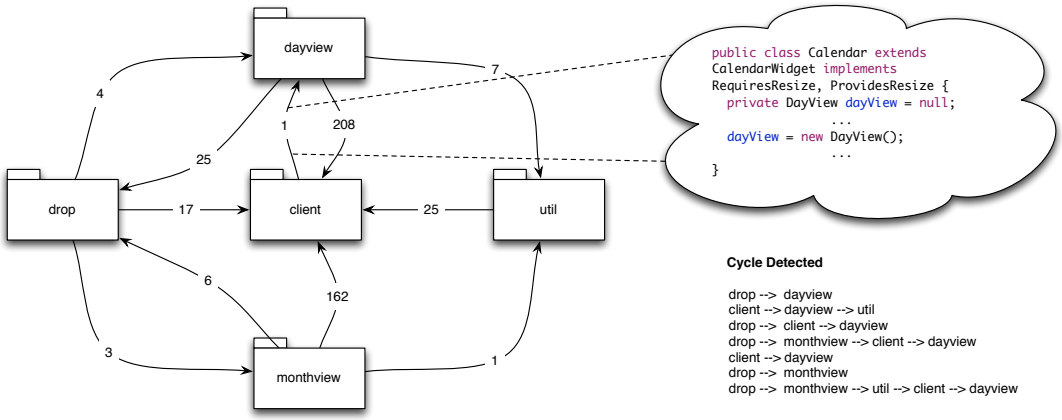

Figure 5.2: Cycles graph in the second round of simulation

Figure 5.3 shows the graph with the involved cycles and the packages involved the initial SCC. For this case Marea detected 2 cycles and 4 logical dependencies where all of them are shared equally between cycles. As we expected Marea suggested the logical dependency *drop* $\rightarrow$ *monthview* since it has few concrete dependencies (3). We checked manually in the source code and we found the method invocation dependencies indicated in the right of Figure 5.3 corresponding to the concrete dependencies found by Marea. In this case we only have one refactoring option: *MC*. The result is correct since the concrete dependencies are in the same class and only *MC* refactoring can be applied. Move method is not applicable because the *onMove()* method contains accesses that are referring to the class where the method is located. For this case *ASP* is not supported as

the invocation is coming from a casted variable and this is a case we can currently not handle and will be considered in future work.

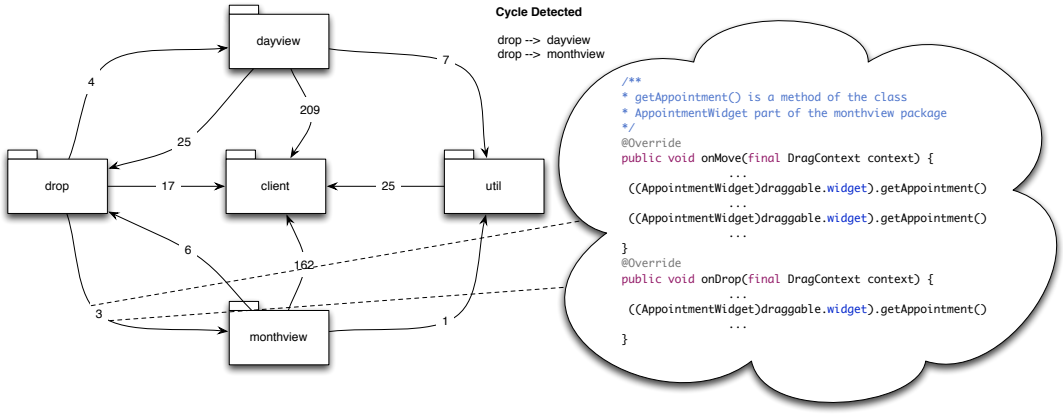

Figure 5.3: Graph cycles in the third round of simulation.

The previous *MC* operation has removed 1 cycle. We now remain with the last cycle, as shown in Figure 5.4. The highlighted logical dependency is obvious because it has fewer concrete dependencies than the opposite logical dependence present in the cycle *drop* $\rightarrow$ *dayview*. The selected logical dependency has been manually verified and the 4 concrete dependencies are corresponding to real dependencies in the source code. Marea simulates this input and returns the tree indicated in Figure 5.5

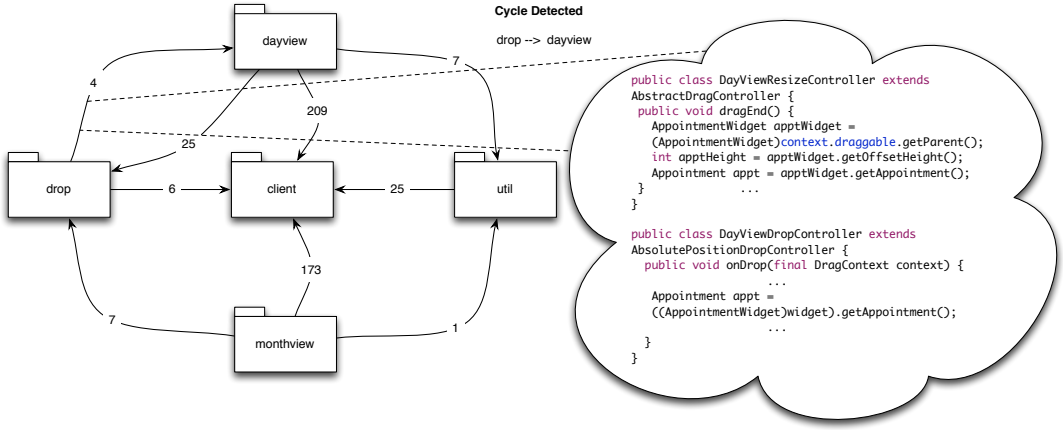

Figure 5.4: Graph cycles in the forth round of simulation.

We have 4 concrete dependencies where 1 is of type local variable and 3 are of type invocation methods. For the local variable dependency it is possible to use *MC* and

ASP. The *MM* is not excluded because the method *dragEnd()* that contains the local variable contains other accesses that are related to the class where the method is located. The 3 invocation method dependencies have the same structure as the previous case so for the same reason it is possible to apply only *MC*. You can notice that the tree has 2 levels: the first suggests operations for breaking the local variable and the second suggests operations for breaking the invocation method. Note that the local variable and the invocation methods are not located in the same class. For this reason the first *MC* is not removing the 3 invocation method dependencies. On the other hand the 3 invocation dependencies are located in the same method so one *MC* operation is enough to remove the 3 of them.

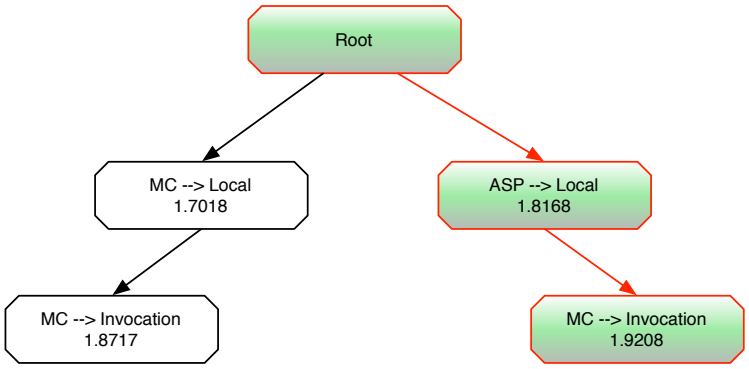

Figure 5.5: Decision tree for the LD *drop* → *dayview*.

After the last simulation we finally reached the point that we do not have cycles in the input project anymore. Figure 5.6 shows the initial graph transformed in a acyclic graph. All the simulated operations have been successfully validated manually by applying them on the source code within Eclipse.

5.3 Presence of Cycles with Many Relations

JHotdraw is our last test example. It consists of 26040 entities (number of entities in the Moose model) and 38 packages in total. The number of considered packages is 11 and the number of considered classes is 140. As you notice from the high number of entities and the low number of packages this input has an elevated number of relationships. Focusing on the decision tree and on the profit we will see that using different metrics the results will change suggesting different paths.

The number of the initial cycles is 13 and the number of involved logical dependencies is 18. After sorting the logical dependencies Marea suggested the logical dependency *standard* → *contrib*. This dependency consists of 5 concrete dependencies, 1 is of type

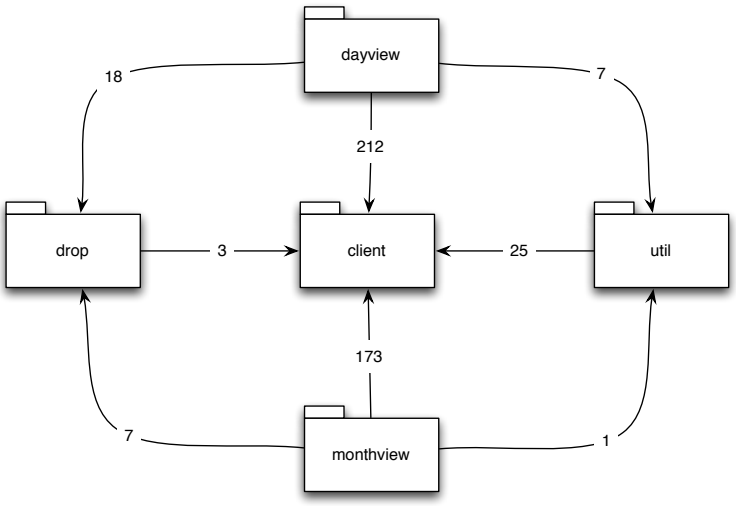

Figure 5.6: Final acyclic graph.

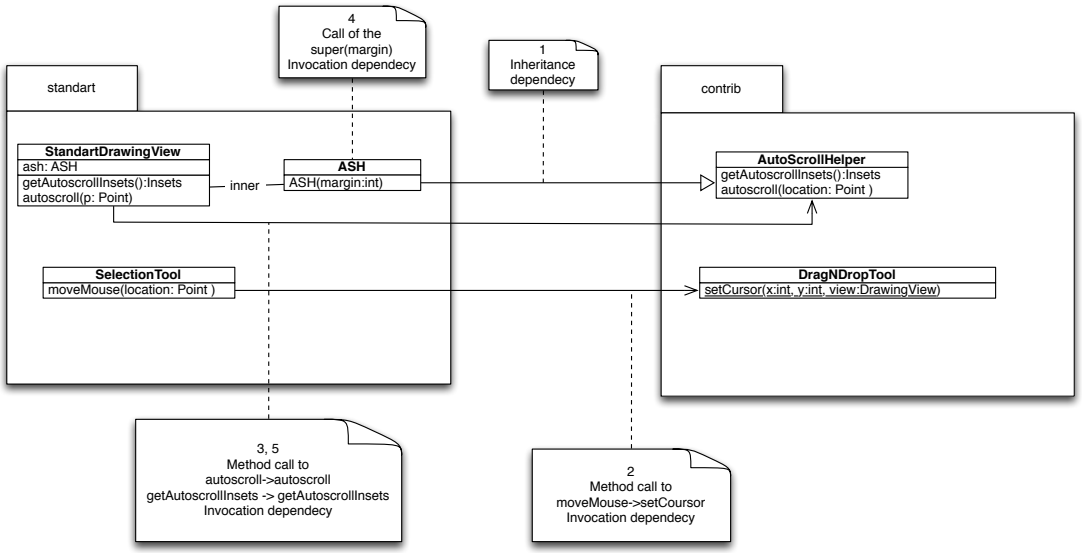

Figure 5.7: Logical dependency with 5 concrete dependencies.

inheritance dependency and 4 are of type invocation method dependencies. Analysing the source code manually we found exactly the 5 dependencies as shown in Figure 5.7.

The simulation of the refactoring operations for the logical dependency that we described above is shown in Figure 5.8. The indicated tree does not include the profit because in this paragraph we are going to validate the single operations. The first input

of simulation is the inheritance dependency of the inner class *StandardDrawingView*. The only possible operation for this concrete dependency is the *MC* refactoring operation. After moving the inner class we can notice that the next input is the initialised class variable dependency. This concrete dependency is related to the attribute *ash* with type *ASH* that is generated after moving the class *ASH* from package *standard* to package *contrib*. For the initialised class variable there are 2 possibilities: *MC* and *ASP+DI*. This is exactly what we can see at the second level of the tree: 2 subtrees referring to the initialised class variable dependency. Both refactoring operations can be used to remove this concrete dependency and in addition they will also remove the invocation dependencies related to the attribute *ash*. At the third level of the tree we find the invocation dependency related to the classes *SelectionTool* $\rightarrow$ *DragNDropTool*. This invocation dependency is refactored by *MC* and *MM*. The *ASP* refactoring for this case is not possible because the invoked method is a static method and static methods cannot be added in an interface. The last level shows a dependency that is generated after moving *SelectionTool* class. The *NullDrawingView* class part of the *standard* package has an invocation of the *SelectionTool(...)* constructor that now is part to the *contrib* package. For this last dependency it is only possible to perform a *MC*, since *MM* and *ASP* are not possible in constructor invocation dependencies.

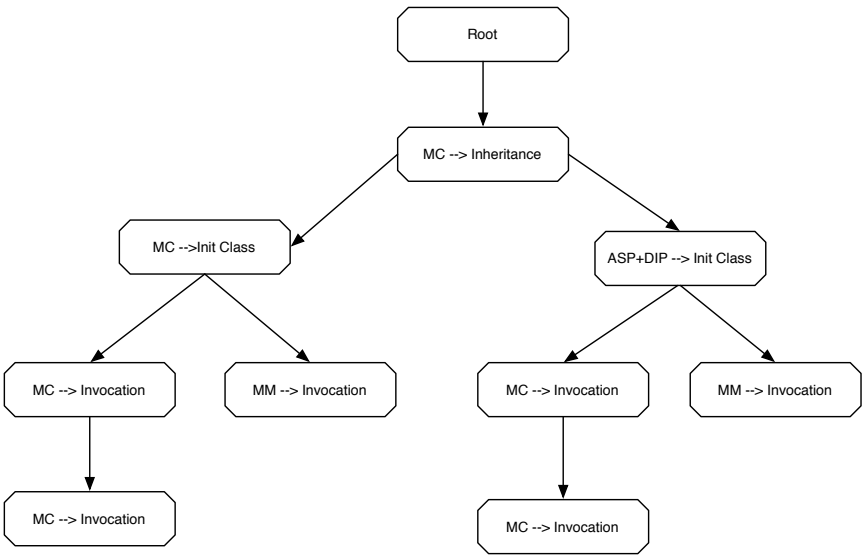

Figure 5.8: Decision tree for the LD shown in Figure 5.7

Considering Formula 4.4, Marea generates the best profit as indicated in Figure 5.9. In this example, we used the same weight constant for each term of the equation, w_c = w_d = w_i = w_a = 1. The best path is indicated in green: in this case we found that the total number of cycles is smaller and the abstractness is higher as ASP is adding a new

interface to the *standard* package. You can notice that the leaves at the last level are penalised as the depth is bigger and as a consequence the number of operations is higher.

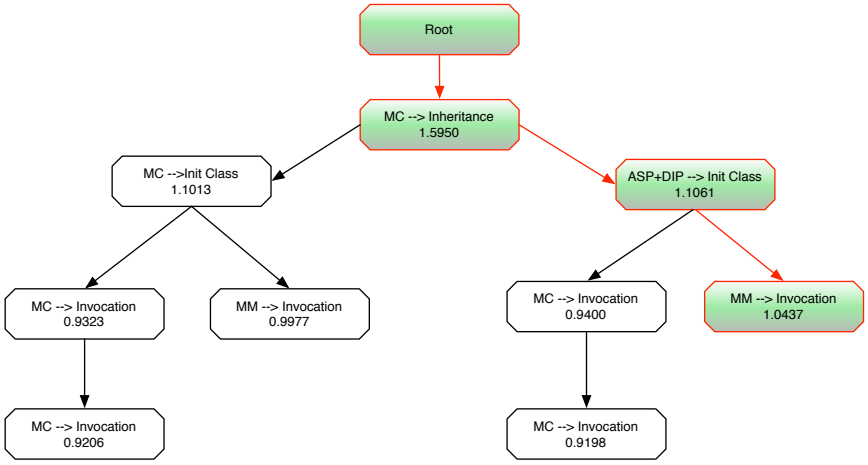

Figure 5.9: Decision tree with the best profit.

For the calculation of the profit we utilised the stability and the abstractness metrics. As mentioned earlier, the user can influence the results by tuning the weights or using other metrics (e.g. using only the stability metric). This makes Marea adaptable to the user needs. Supposing that the user does not care about the abstractness but wants stable packages, he could decide to increase the weight of the stability metric (w_i = 10) and exclude the abstractness metric. In this case he would get different results, as shown in Figure 5.10. In this case the best path is on the left of the tree as the stability is higher.

5.4 Complexity

The complexity of the main operations as a function of the size of the graph is described as following:

Complexity time for cycle detection. In Section 4.2.1 we explained that cycles are extracted from each SCC. We explained as well that to detect the SCCs we used Tarjan's algorithm that operates in linear time, i.e. $O(|V| + |E|)$ where V is the number of vertices and E is the number of edges, in our case, number of packages and number of logical dependencies respectively. The complexity of cycle detection results in the number of SCCs times the time it takes to detect the cycles from an SCC. The worse case to detect the cycles from an SCC is where the whole graph is a strongly connected component. Therefore, to detect the cycles we need to traverse all the nodes and edges of the graph, through the

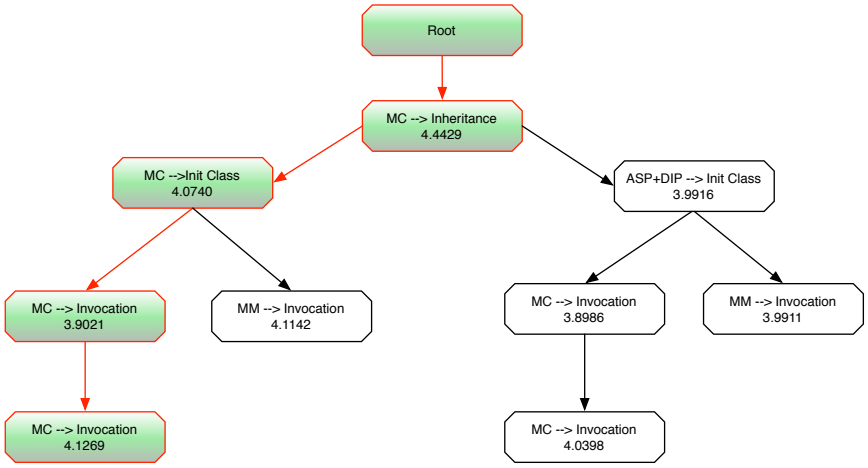

Figure 5.10: Decision tree with the best profit utilising stability metric.

DFS procedure in time $O(|V| + |E|)$. After these considerations, the complexity of cycle detection results in $O(|S|(|V| + |E|))$ where S is the number of strongly connected components.

Complexity time for sorting the logical dependency. The operation of sorting the logical dependencies is needed in order to suggest to the user which logical dependency is better to consider as an input for the simulation. In the previous paragraph we mentioned that the number of logical dependencies is the number of edges of the input graph. Notice that not all the edges are of interest because not all of them are within the cycles. However, in the worse case, all the edges of the graph are within the cycles. The sorting procedure copies the logical dependencies from a collection into a sorted collection. This procedure takes time in $O(|E|)$ since it iterates over all the elements of the collection and adds them into the sorted collection one by one.

Complexity time for simulating the logical dependency. This operation simulates the refactoring methods for the concrete dependencies located in a logical dependency. The execution time for this case depends on both the number of concrete dependencies that are located in the logical dependency and the number of refactoring methods. For this reason there are two considerations for the complexity time of this operation. We describe these considerations as following:

The applicable refactoring methods do not introduce new dependencies. The worst scenario is where all the refactoring methods are applicable to the concrete dependencies located in the logical dependency in process of simulation. Based on Table 4.1 the total number of refactoring methods is 4 where 2 of

them are mutual exclusive (*ASP* and *ASP+DI*). To calculate the complexity of the simulation operation, we have to calculate all the possibilities of refactoring operations of the concrete dependencies. The number of all possibilities of the refactoring operations corresponds to the maximum number of nodes of a ternary tree of height *d* where *d* is the number of concrete dependencies. It is a ternary tree because we have 3 refactoring operations. The maximum number of nodes of a ternary tree is calculated by Formula 5.1

$$Max(N) = 1 + 3 + 9 + ... + 3^d = \sum_{i=0}^{d} 3^i = \frac{3^{d+1} - 1}{2} \quad (5.1)$$

Marea redetects the cycles and resorts the logical dependencies for each node of the tree and the complexity time of these operations we described above. In conclusion, the complexity time of the simulation of the logical dependency is the maximum number of nodes of a ternary tree times the complexity of cycle detection (for big *E* sorting time is ignored), i.e. $O(|Max(N)|(|S|(|V| + |E|)))$.

The applicable refactoring methods introduce new dependencies. In this case, the decision tree may go to infinitely because there is always a new concrete dependency for simulation. Time complexity for the worst case is $O(infinite)$[5].

5.5 Performance

We measured the running time of the main components of Marea for the inputs that we discussed above and reported the results in the Table 5.1. We calculated the simulation time for all the logical dependencies, the cycle detection time and the sorting dependencies time for the inputs that we discussed above. Considering that for each refactoring operation Marea has to redetect the cycles, the simulation time includes as well the time of redetecting the cycles. We notice that the bigger is the number of concrete dependencies in a logical dependency, the higher is the execution time of a simulation as the depth of the tree grows.

[5]Marea stops the execution of a path if the simulation does not ends and looks for other path solutions

Input	LD - CDs	Simulation (s)	Cycle Detection (s)	Sort Dependencies (s)
GWT-Cal	LD - 1	3	1	0
	LD - 1	2		
	LD - 3	1		
	LD - 4	3		
JHotDrow	LD - 5	14	1	0

Table 5.1: Time in seconds of the main operations.

6 Conclusion and Future Work

This chapter concludes the thesis summarising our work and future work.

6.1 Conclusion

Dealing with dependency cycles is a difficult topic, as engineers need to access source code and understand dependencies. Without a proper tool support, developers, maintainers and testers cannot remove these problems from their systems. We implemented Marea to assist engineers in resolving cyclic dependencies at the package level. Marea simulates the refactoring operations in a separate platform without changing the structure of the source code. In this way engineers can have suggestions for removing undesirable dependencies and apply the refactoring operations in case they agree on the changes. This approach allows engineers to improve the modular structure of their object-oriented systems without changing their functionality.

In our solution we implemented common refactoring methods, such as *Move Class* and *Move Method*. In addition to those we implemented other refactoring methods not supported by current tools, such as *Abstract Server Pattern* and *Dependency Injection Pattern*.

The suggestions that we provide are ranked based on profit function that is calculated for each simulated refactoring operation. The profit function is designed to minimise the number of refactoring steps as well as the number of cycles and to increase the overall quality of the code. We show that the suggestions can be customised based on user needs by adapting the profit function.

Our validation with Java based projects shows that our approach works and can be used to refactor a real object-oriented system.

6.2 Future Work

As future work we planned to address the following points:

- User interface interaction. We already implemented the graphical interface of the the simulated tree. A graphical user interface can also be implemented for the whole process of interaction with Marea as future work.
- Automate the refactoring decisions integrated with Eclipse plugin. The suggestions that are given by Marea can be applied to the source code by implementing an Eclipse plugin. The idea is to import the simulated trees (one tree for each logical dependency) from a log file in Eclipse environment. A tree contains the simulated refactoring operations as well as the path that the user decides to apply for breaking a certain logical dependency (the best path is calculated by Marea by default). The plugin will extract the refactoring decisions from the imported file and apply them to the code using the API of Eclipse[1].

[1] *Move Class* and *Move Method* are already implemented in Eclipse

Bibliography

[1] R. C. Martin, "Design principles and design patterns," *Object Mentor*, 01 2000.

[2] P. Eades, X. Lin, and W. F. Smyth, "A fast effective heuristic for the feedback arc set problem," *Information Processing Letters*, vol. 47, pp. 319–323, 1993.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 06 1999.

[4] R. E. Tarjan, "Depth-first search and linear graph algorithms.," pp. 146–160, 1972.

[5] R. C. Martin, "Granularity," *Object Mentor*, 1997.

[6] H. Melton and E. Tempero, "Jooj: Real-time support for avoiding cyclic dependencies," in *Proceedings of the Thirtieth Australasian Conference on Computer Science - Volume 62*, ACSC '07, (Darlinghurst, Australia, Australia), pp. 87–95, Australian Computer Society, Inc., 2007.

[7] J. Laval, *Package Dependencies Analysis and Remediation in Object-Oriented Systems*. PhD thesis, University of Lille, June 2011.

[8] H. Melton and E. Tempero, "Identifying refactoring opportunities by identifying dependency cycles," in *Proceedings of the 29th Australasian Computer Science Conference - Volume 48*, ACSC '06, (Darlinghurst, Australia, Australia), pp. 35–41, Australian Computer Society, Inc., 2006.

[9] "Stan." `http://stan4j.com`.

[10] E. Hautus, "improving java software through package structure analysis," *The 6th IASTED International Conference Software Engineering and Applications*, 2002.

[11] "Lattix." `http://lattix.com/`.

[12] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," *SIGPLAN Not.*, vol. 40, pp. 167–176, Oct. 2005.

[13] "Structure 101." `http://structure101.com/products`.

[14] J. Laval, S. Denier, S. Ducasse, and A. Bergel, "Identifying cycle causes with enriched dependency structural matrix," in *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, pp. 113–122, Oct 2009.

[15] J. Laval, S. Denier, S. Ducasse, *et al.*, "Cycles assessment with cycletable," 2011.

[16] J. Laval, N. Anquetil, and S. Ducasse, "Ozone: Package layered structure identification in presence of cycles," in *Proceedings of the 9th edition of the Workshop BElgian-NEtherlands software eVOLution seminar, BENEVOL*, 2010.

[17] J. Laval, S. Denier, S. Ducasse, and J.-R. Falleri, "Supporting simultaneous versions for software evolution assessment," *Science of Computer Programming*, vol. 76, no. 12, pp. 1177 – 1193, 2011. Special Issue on Software Evolution, Adaptability and Variability.

[18] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[19] M. Fowler, "Inversion of control containers and the dependency injection pattern," 01 2004.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[21] R. C. Martin, "Stability," *Object Mentor*, 1997.

[22] M. Fowler, "Inversion of control," 06 2005.