CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Reengineering Java Game Variants into a Compositional Product Line

An empirical case study identifying activities and effort involved in a reengineering process

Master's thesis in Computer science and engineering

JAMEL DEBBICHE
OSKAR LIGNELL

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Reengineering Java game variants into a Compositional Product Line

An empirical case study identifying
activities and effort involved in a reengineering process

Jamel Debbiche, Oskar Lignell

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Reengineering Java game variants into a Compositional Product Line
An empirical case study identifying activities and effort involved in a reengineering process
JAMEL DEBBICHE, OSKAR LIGNELL

Supervisor: Thorsten Berger, Department of Computer Science and Engineering
Examiner: Jennifer Horkoff, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Reengineering Java game variants into a Compositional Product Line
An empirical case study identifying activities and effort involved in a reengineering process

JAMEL DEBBICHE, OSKAR LIGNELL
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Compositional Software Product Line Engineering is known to be tidious but powerful approach to migrate existing systems into SPL. This paper analyses the pros and cons of compositional SPLE strategies and also attempts to migrate five related Java games into an SPL while outlining the necessary activities to perform such migration. This paper also presents how to measure migration efforts of each activity. Lastly, the results of the migration process is compared to the result of another Master thesis that also conducts an SPL migration but using the annotative approach.

Keywords: Software Product Line, Reengineering, Migration.

Acknowledgements

The researchers would like to extend their gratitude towards supervisor Thorsten Berger and Jacob Kruger whom provided assistance with direction of the research, as well as contributing to valuable discussions. The researchers would also like to thank ApoGames for providing the dataset used in this research.

Jamel Debbiche, Oskar Lignell, Gothenburg, May 2019

Contents

List of Figures

List of Tables

1
Introduction

Software product line engineering (SPLE) is a set of methods, tools and practices that takes several related software products and to engineer them under their common assets [1]. This is to take advantage and reuse these common assets instead of the need to re-create them for every product. A software product line (SPL) is most often made from a family of related software systems that has went through a process of reengineering [3].

Already since 1990s, SPLs has gained popularity in the industry [1]. This is to combat the need to rewrite common parts for every new product, and to enable high customization while maintaining mass production. It is done by separating the software into different features where the customer can choose a set of features and generate their own product based on their unique requirements. This means that SPLs enables individualism while still retaining the ability to mass produce [1].

These advantages of SPLs makes it worthwhile for organizations to reengineer their already-developed products into SPLs, not only to take advantage of the commonalities, but also to provide customers a wider range of configuration options. In this thesis, we explore the activities, resources and methodologies that are necessary to perform such reengineering.

1.1 Problem Statement

Commonly, reusing software artifacts is done in an ad-hoc manner, also known as "clone-and-own methodology" [4]. This cloning results in a large amount of duplicated code that is ultimately expensive to maintain. When, for example, a bug is found in one of the clones it has to be maintained in all of the cloned versions. Similarly, when optimizing a portion of the duplicate code, you need to make sure to evolve all the variants that has that portion of code.

Since it is mostly already existing systems that is reengineered into SPLs [5] and because of the problems that come with clone-and-own, we believe that this calls for a strategy where a set of activities that dictates how to transform a family of related software into a product line to be established. In addition to identifying the activities, it is important to measure the efforts of each activity in order to estimate the resources necessary for this migration.

Currently, not that much empirical data is available on different efforts and costs evolving around migrating existing systems to an SPL. It is stated in literature that the integration part during a reengineering process needs further research, in order to bring SPL results to a broader practice [6]. This means that a company cannot identify whether or not it can or even how to extend to an SPL. Therefore, there is a need to understand all the efforts and costs involved. With this study, detailed qualitative (such as activities involved) and quantitative (using certain metrics such as number of hours to perform each activity) empirical data are provided that is gathered from logging activities and efforts of the migration process.

1.2 Purpose of the Study

The purpose of this study is to migrate an existing family of software into a software product line and also to identify different costs, in terms of effort, related to a reengineering process. This is achieved by using a dataset of five Java games publicly available on BitBucket[1]. After reviewing existing literature within the area it was discovered that there is a need for further research. Studies have identified open issues such as the need for new metrics and measures in terms of efforts, as well as other challenges such as feature location, migration to software product line and more [3] [7].

The goal is to understand what kind of activities are involved in the migration process, from start to finish and what are the efforts necessary to accomplish the identified activities. By doing this, a detailed dataset is provided that contains the different phases of a migration and its efforts that can help companies measure the feasibility of such migration. The strategy that is provided can also be beneficial for researchers as they can test its applicability on different domains. Hence, both researchers and companies can benefit our findings, where a researcher will have more reliable data, in terms of activities and their efforts. And an organization will have more indicators on whether or not it is worthwhile to reengineer their existing systems into an SPL.

1.2.1 Research Questions

As mentioned before, this study attempts to identify the efforts and activities necessary for migrating Java games into a software product line. This is done by thoroughly logging the entire reengineering process.

From this, one main objective is defined: Identify the activities and their related efforts needed for migrating clones of Java games into a compositional software product line. The following research questions can be derived from the objective:

RQ.1 What are pros and cons of current migration strategies based on literature? A migration process can be made in different ways. In order to be able

[1]Source code: https://bitbucket.org/Jacob_Krueger/apogamessrc/src/7b8c7973b595?at=master

to achieve a good result when making a migration, it is necessary to know what is better and worse with using different strategies. This can be understood by a literature review before deciding on which strategy to use.

RQ.2 How to measure migration effort? Efforts necessary for the migration process is an important factor to consider for companies before implementing the reengineering. Measuring efforts helps organizations decide whether or not the migration process is worth the investment. It can be solved by designing a logging template based on relevant cost models for each activity in the migration process.

RQ.3 What are the activities involved in a compositional SPL reengineering? It can be unclear what the migration process explicitly entail. To get a detailed understanding on compositional reengineering, it is necessary to identify the performed activities for this type of migration. These activities will provide all steps that has to be done in order to migrate the existing software.

RQ.4 What are different efforts of the activities? After understanding how to measure effort, an activity needs to be mapped to the relevant efforts. By doing this, it is possible to see what activities and which part of the reengineering process that is resource intensive.

1.3 Reading Instructions

This thesis involves many concepts regarding SPL:s and the reengineering process that can be confusing given their overlapping definitions. Section 3.5 contains a detailed description concerning the reengineering process in practice. Headings of each subsection corresponds to a performed activity whose efforts can be read in appendix A.3. Section 2.2 provides a more theoretical understanding of the reengineering process, with detailed descriptions of relevant terminologies that are of high importance in the SPL migration field.

2
Background

This chapter is divided into four main sections. The first section introduces SPL and SPLE implementations, more specifically compositional SPLs and the tool used in this thesis. Secondly, it clarifies and defines important terms. The third section gives an introduction to the various SPL migration strategies used in previous literature and the last section describes cost models that were used as the foundation for the logging artifact.

2.1 Software Product Lines

Software Product Line is defined by Clements and Northrop as "*a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*" [8]. SPLE encourages the extraction of common software artifacts in order to take advantage of reusing these software components and hence maximizing the possible configuration of a software system.

Over the years, SPL has displayed several advantages in dimensions like, business, architecture, process and organization [9]. Some of the most important advantages are reduced costs, improved quality, reduced time to market and tailor-made software. This is because of separating commonalities and variabilities into reusable software components, which enables customization while still allowing mass production. Individual configurations enables companies to provide a plethora of options that can cover all the specific requirements given by their customers.

The adoption of SPLE extends to large-scale software systems. This is mainly achieved by significantly lowering the costs of maintainance and creation of new variants from the product line. This remedies the main drawback of the clone-and-own methodology [1]. One of the main aspect of SPLE is the separation of the domain level from the application level, this is further explained in the following section.

2.1.1 Domain and Application Engineering

Developing a single software product means that the development only considers the requirements of that system and its life-cycle[1]. This changes when it comes to SPLE, since the product line is expected to accomodate a high number of con-

figurations that increases overtime as features are added. This significantly extends the life-cycle. In order to be able to continuously develop on top of an SPL, the domain of the product must be clearly understood. Because of this, there is strong focus on domain knowledge where the domain and the application engineering are considered as separate aspects of SPLE [1].

In summary, domain engineering entails all the activities that assist in understanding the domain in which the software system operates in. It is also identifying all the common software artifacts that are to be reused by all the products[1]. In other terms, all the features that exists in every variant. The application aspect on the other hand, takes care of the product specific software artifacts that with the common base, can create a specific product to satisfy a specific customer. To summarize, SPLE is about dividing a software into reusable features, which some belong to the domain level, and others to the application level [1]. Figure 2.1 provides an overview of SPLE in context of Problem and Solution space.

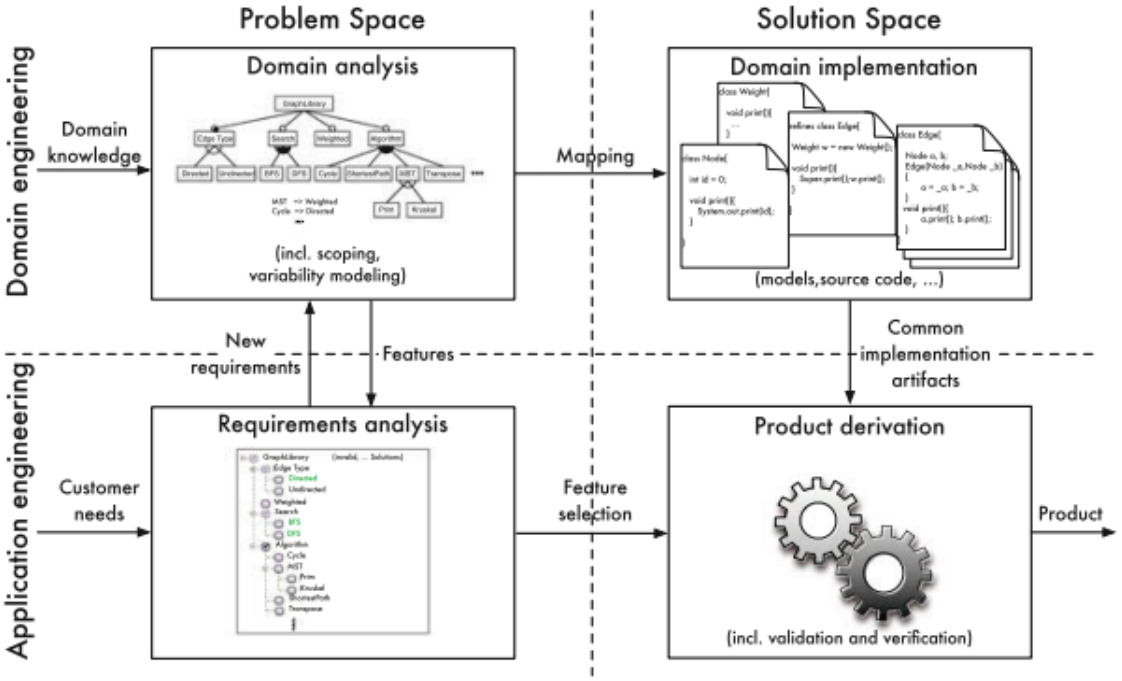

Figure 2.1: Overview of an engineering process for software product lines [1]

The idea is to collect all the common features in a common code base and separate the variability from it. This way a company may configure any desired product using the common base with a complimentary selection of compatible features [1]. This is why SPLE often takes a feature oriented approach and Feature Oriented Software Product Line is one of the well-established methodologies towards SPLE [10]. Like all the other methodologies, it considers the distinction between the domain and application engineering by focusing on four main areas of SPL:

- Domain Analysis
- Requirement Analysis
- Domain Implementation

- Product Derivation

The term 'feature' has several different definitions in the academic world [11], however, in the context of SPLE, the definition that best covers the commonality and variability concepts is provided by Apel et al. [1], which is "*A feature is a characteristic or end-user-visible behavior of a software system. Features are used in product-line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all phases of the software life cycle.*". In the next section, the different approaches of SPL Implementation are explored.

2.1.2 Different Approaches of SPL Adoption

The approach to adopt software product lines is very situational, meaning it depends whether there is an already existing system to be migrated or if the system will be created from scratch. If it is the former, this is known as an extractive approach[1]. It will also depend on the artifacts that exists, for instance what are the documentation available etc.

This thesis concerns the migration of five related existing software products, hence the use of the extractive approach, or in other terms, reengineering. However, the extent of examining the four aforementioned areas depends largely on resources available. For instance, in this research, the only resource present is the source code of the five related products. Customers or the original developer of the five Java games are not accessible, nor any high-level materials such as domain models or list of requirements. This means that it is not possible to conduct any activity within the Requirement Analysis phase. In the next section, some of previous attempts on the adoption of an extractive approach in SPLE is provided.

2.1.2.1 Previous Attempts in SPL Reengineering

Studies have tried to reengineer applications by using different techniques to find clones and migrate it into an SPL. For instance, one study migrated cloned product variants into an SPL by using code clone detection [12]. This identifies commonalities which afterwards are extracted into shared artifacts. Results showed that LOC are reduced by approximately 15% overall. The authors also state that migration tasks are challenging and at the moment not well supported.

Another study by Balazinska, et al. tried to measure reengineering opportunities by having a clone classification scheme [13]. They mention that the research focus has turned from investigating clone detection into trying to find actions for software restructuring based on clone detection. The authors concluded that to decide if a system is worth reengineering is more complex than just based on how much of the code is cloned. In Alcatel-Lucent, an industrial case study was conducted where they did a reengineering project towards an SPL. The project was performed with agile principles. It was concluded that by taking on the project with an incremental and iterative approach, SPL reengineering can be cost-effective and successful [14].

Therefore, this study applies an iterative approach as well. The authors of this thesis are also familiar with performing a project in an agile way, which helps in order to have a good result.

2.1.3 Compositional Software Product Line

After having defined the SPL implementation as an extractive approach, we now define our re-engineering methodology. In other terms, how to transform the existing systems into an SPL. There are several ways to transform a software system into an SPL, all of them can be grouped under either a compositional or an annotative approach [15]. This study uses the compositional approach which breaks down features into physically separated code units in accordance to a feature model [16]. Once this is done, a variant can be generated by selecting a valid configuration.

The generation is resulted by superimposing code units responsible for the features selected. The concept of superimposition is described in the section below. This means that feature location and composition is a crucial step in compositional SPL. Feature composition is usually done with the assistance of SPL tools [16]. In this study, the tool FeatureHouse is used as it is one of the most recent tools for compositional SPLE and it is a continuation of the tool AHEAD[2].

2.1.3.1 FeatureHouse

The framework FeatureHouse works for several different programming languages such as Java, C#, C among others. It is an asset for software composition and uses the concept of superimposition [17]. FeatureHouse structures software fragments as a general model called a *feature structure tree* (FST), which gives a hierarchical structure for a fragment that represents packages and classes along with its methods and fields [2]. It uses FSTs to achieve the superimposition concept. Figure 2.2 shows the structure of an FST.

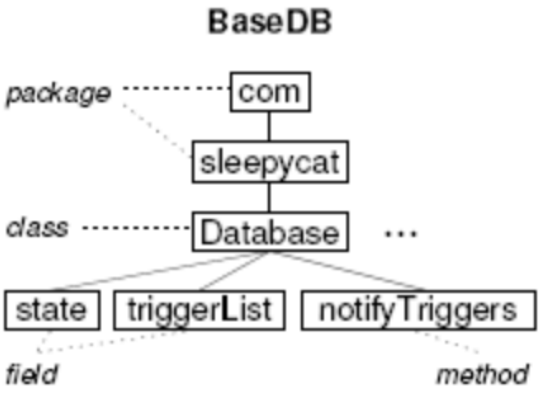

Figure 2.2: Example of a Feature Structure Tree (FST) [2]

The following example describes superimposition: For a given class, the code in that class can belong to *feature-x* and *feature-y*. During the reengineeing, the class is divided into several files with identical file name where code fragments will be inserted into its respective feature. If the developer generates a variant that includes *feature-x* and *feature-y*, then the two files will merge to a single file. This can extent to the method level. Meaning one method can be divided between two features. This is done by having the same method definition in both files which is merged when a variant is generated. An illustration of the process can be seen in Figure 2.3 where the method *notifyTrigger()* is merged using superimposition. It is done by calling the FeatureHouse method *original()* carrying the same parameters as the method *notifyTrigger()*.

```
1  package com.sleepycat;
2  public class Database {
3    private void acquireReadLock() throws DatabaseException { ... }
4    private void releaseReadLock() throws DatabaseException { ... }
5    protected void notifyTriggers(Locker locker, DatabaseEntry priKey,
6      DatabaseEntry oldData, DatabaseEntry newData) throws DatabaseException {
7        acquireReadLock();
8        original(locker,priKey,oldData,newData);
9        releaseReadLock();
10   } // 50 further lines of code...
11 }
```

●

```
1  package com.sleepycat;
2  public class Database {
3    private DbState state;
4    private List triggerList;
5    protected void notifyTriggers(Locker locker, DatabaseEntry priKey,
6      DatabaseEntry oldData, DatabaseEntry newData) throws DatabaseException {
7        for(int i=0; i<triggerList.size(); i+=1) {
8          DatabaseTrigger trigger = (DatabaseTrigger)triggerList.get(i);
9          trigger.databaseUpdated(this, locker, priKey, oldData, newData);
10       }
11   } // over 650 further lines of code...
12 }
```

=

```
1  package com.sleepycat;
2  public class Database {
3    private DbState state;
4    private List triggerList;
5    private void acquireReadLock() throws DatabaseException { ... }
6    private void releaseReadLock() throws DatabaseException { ... }
7    protected void notifyTriggers(Locker locker, DatabaseEntry priKey,
8      DatabaseEntry oldData, DatabaseEntry newData) throws DatabaseException {
9        acquireReadLock();
10       for(int i=0; i<triggerList.size(); i+=1) {
11         DatabaseTrigger trigger = (DatabaseTrigger)triggerList.get(i);
12         trigger.databaseUpdated(this, locker, priKey, oldData, newData);
13       }
14       releaseReadLock();
15   } // over 700 further lines of code...
16 }
```

Figure 2.3: Example of Superimposition of a Java method [2]

2.1.3.2 Differences of Annotative and Compositional Approach

As oppose to compositional, the annotative approach does not actually break down the code into features but it defines features in the source code itself. Features are usually surrounded by *#IFDEF* and *#ENDIF* which are later recognized by a language dependent pre-processor, so only the features selected at the configuration state are executed [15].

This difference affect mainly three areas of SPLE: modularity, granularity and SPL adoption[16]. Modularity is low with the annotative approach since the source code is kept the same, while the compositional technique actually divides source code into

feature modules. Hence, increasing the modularity. Granularity however is increased for annotative as the use of *#IFDEF* and *#ENDIF* can be used at any level (classes, methods or statement level). However, in a compositional approach the developer must manually break down the code and divide each code fragment responsible for each feature to its own feature module. Hence, it requires much more work than the annotative method. Lastly, SPL adoption using a compositional approach can be quite unnerving for companies. This is since the compositional approach necessitate that the company changes its existing source code, and at time, that change can be drastic [18]. An annotative approach only introduces annotation in the existing code, but with reduced feature traceability and modularity [15]. Because of this, even though the compositional approach is considered to be tedious and costly, it is still considered superior in the academic community[16].

2.2 Clarification of Important Terms

This section describes in detail the differences and relationships between some of the terms that this Master Thesis is based upon. It is important that the reader can differentiate between these, otherwise many of the concepts and approaches that later is described may be misinterpreted. Figure 2.4 describes the relationships between the important terms with the help of UML.

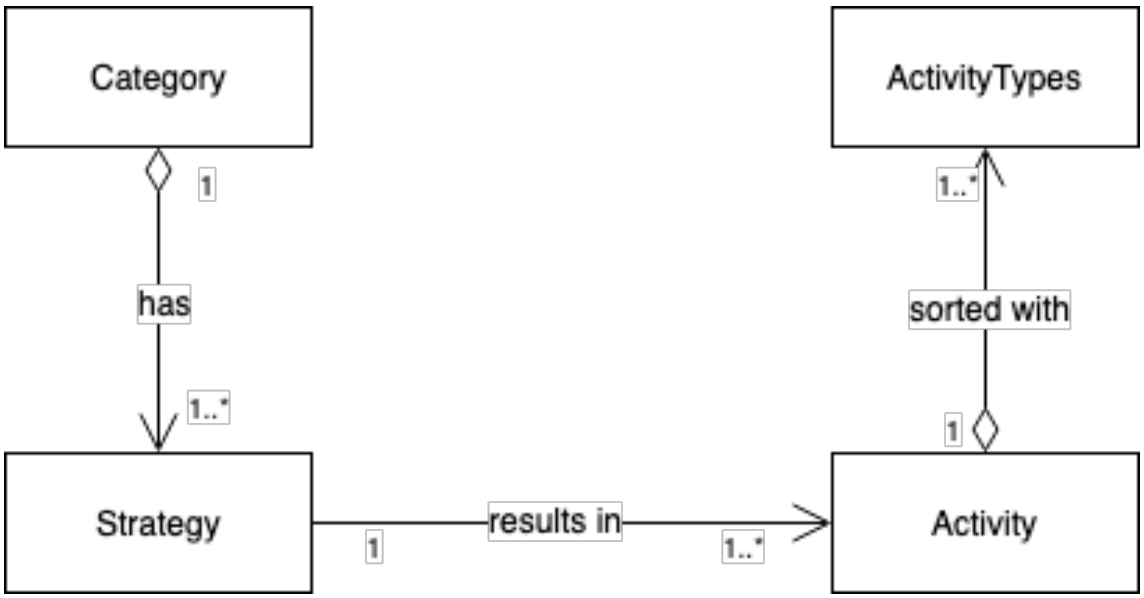

Figure 2.4: Activity and strategy relationships

2.2.1 Activity

The definition of an activity is something that is done in practice, necessary for a successful reengineering process. It is logged using the logging template, which can be read in section 3.4. The granularity of an activity has been discussed among group members in two Master Theses and respective supervisors with knowledge about this subject. An example of an activity can be found in Appendix A.2 where a matching activity type (see 2.2.2) has been set in order to sort it accordingly. It is also possible to get a better understanding of activity granularity by looking through all performed activities in Appendix A.3.

An activity is rather high-level. This is because if low-level activities are logged, a lot of documentation would be redundant. We assume that readers are familiar with practicalities such as *creating a class/method* or *refactoring* and thereby understands that these practices will occur during the reengineering process. Hence, activities are described at a higher level. To avoid having too abstract activities, activity types (see 2.2.2) exists to prevent this and also to sort the performed activities. These rules provide some guidelines at what level of abstraction an activity should have.

2.2.2 Activity Types

In order to be able to classify performed activities into relevant reengineering areas and also be able to compare results with another Master's Thesis, different activity types have been created. These are based on the SPLE process seen in Figure 2.1, as well as discussions similar to the discussions that resulted in the activity granularity (see Section 2.2.1). The different types can thereby be seen as different steps during *domain engineering* and *application engineering* (Section 2.1.1). All activity types can be seen below.

SPLE training - Any activity that involved researching specific literature of SPLE, including different approaches of SPLE, such as strategies to apply compositional or annotative approach to transform and existing software system into an SPL.

Data cleansing - Could be removing unused code or translating comments to english. Activities that are not of general character (should be filtered out during comparison analysis)

Domain analysis - Identifying commonalities within variants and map it to the domain level.

Feature identification - Finding functionality that could be classed as a feature.

Diffing - Activity that revolves around finding the differences between clones.

Architecture identification - Any activity that revolves around identifying the architecture - i.e creating class diagrams.

Feature location - Activities that relates to identifying which code unit represent what feature.

Feature modeling - Mapping all identified feature into a feature model.

Transformation - Any activity that has to do with code modification to, for example, separate features into separate code units.

Quality assurance - Activities such as running and testing games and game-functionalities after each iteration are classified as quality assurance activities.

2.2.3 Category and Strategy

A category can have multiple strategies, as seen in Figure 2.4. This can be comparable to activity types, where it is stated at a higher level of abstraction. The contained strategies are then a certain way you perform a category. These strategies are more concrete things that you do. Section 2.3 describes different categories with some of their strategies that were found during a literature review. The strategy itself results in performed activities.

2.3 Pre-study: Migration strategies

Before starting the reengineering process, a pre-study with a literature review about existing migration strategies is conducted. This is to contrast and compare the different strategies and decide which strategy, or perhaps a mixture of strategies, that is best suited for our dataset. The current literature available does not show consistent results in terms of strategies, and authors often provide different conclusions as to how one should carry out the migration process. Some systematic mappings of reengineering strategies has been performed [3]. This literature review will help answering RQ.1.

2.3.1 Phases

It is claimed by Assuncao et al. that there is no established, or concrete strategy when it comes to migrating existing systems to SPLs [3]. There are not even a set of phases that are recognized and clearly defined. During their mapping study, they could extract three steps that often occurred.

1. Identify features existing in a set of systems or map features to their implementation
2. Analyze available artefacts and information to propose a possible SPL representation
3. Perform modifications in the artefacts to obtain the SPL

In contrast to Assuncao et al., Anwikar et al. states that there are three main phases while performing a migration [19]. These phases are known as *detection*, *analysis*

and *transformation*. In the first phase, they observe the source code and gets information such as how functionality and architecture is structured. During analysis, information from the detection phase is used to redesign feature functionality such that features are separated and follows layered-code principles. The final phase, transformation, is where the system is actually migrated to the new design from previous phases.

From these descriptions provided by Assuncao et al. and Anwikar et al. [3] [19], this study uses the following terms and definitions to refer to different phases of the migration process:

1. **Detection:** Identify features and structure in the system variants
2. **Analysis:** Analyze variants and design a possible SPL
3. **Transformation:** Modify variants to obtain a SPL

2.3.2 Top-down vs. Bottom-up approach

It is possible to approach the migration process in different ways as well, and not only focus on strategies. Top-down and bottom-up defines how one can identify features. With the top-down approach features are first located at a coarse or rough granularity, to continue downwards to make the feature more fine grained [20]. Meaning that a feature in the beginning is not defined by certain methods or LOC:s, but rather in what variant and what classes the feature is present. Later in the process the feature is located in a lower level, such as which functions are responsible for the said feature. A bottom-up approach, is simply put, where you approach the problem the other way around. One specific variant is picked and in detail finding features directly in the source code, to later on identify common features when all variants have been searched through [5].

2.3.3 Strategies

When it comes to the reengineering strategies, literature classifies all the strategies into five categories [3]. It is also important to mention that some papers uses a combination of strategies, which is called a *hybrid* strategy [21]. The five categories are listed below and are ranked in order of most used in research papers [3]:

1. Static Analysis
2. Expert Driven
3. Information Retrieval
4. Dynamic Analysis
5. Search-based

Not all of these types consider all the three phases of migrations, which means if such category is chosen, then there must be an assumption that some phases have already been performed before the migration process.

For instance, the categories Dynamic Analysis and Information Retrieval only consider the first two phases; Detection and Analysis. Additionally, Information Retrieval is more focused on larger systems as it spends most of the resources on mining all sort of data relevant to the system. Strategies within the Search-based category, while being the least used strategies, focus on creating and optimizing variability models and for existing systems[3].

As each category focuses on different aspects of the migration process, it might be necessary to create a hybrid strategy. For instance, by utilizing tools such as ObjectAid [22] to create class diagrams and reverse engineer the design of the Java games, hence using a Dynamic approach [23]. Additionally, one may apply a search-based strategy to extract variabilities of the system, hence using a hybrid strategy.

Different hybrid recommendations exists in papers, such as Dynamic Analysis combined with Static Analysis [24][25]. Another combination could be Static Analysis and Information Retrieval [26].

All the categories are described in further detail below.

2.3.3.1 Static Analysis

These types of strategies are the most used in literature [3], they are usually used during early stages in development [27]. Given its widespread usage [3], many tools are based on static analysis to automate the process of finding defects within the code. These tools can handle large industrial applications [28]. Also, these strategies can be applied to either a whole software system or a single file. Moreover, is it not necessary that the software development process has been finalized [29], i.e. Analysis can be performed during development.

During Static Analysis, the focus is on the source code while not executing the software. This means that the purpose is to analyze the code structure. It could for example be done with a strategy such as control flow analysis to determine what paths that are possible for the software to take [27]. Hence, knowing how a feature propagates in the system. Another strategy example is symbolic analysis, where the program variables are the focus and can be the source for feature identification [29].

An advantage of static analysis strategies is that the software system does not need to be executed. This is because the software system that is to be migrated to an SPL may not always be in an executable state. Using these strategies, one can identify the code structure of a system and infer its architecture. This information can aid the development to identify the functionalities of the system as well as the quality attributes that needs to be carried over in the SPL migration.

2.3.3.2 Dynamic Analysis

During Dynamic Analysis the software is executed, in contrast to Static Analysis where it is not. It focuses on finding execution traces of the software for differ-

ent features [30]. This is done by generating feature-specific scenarios. By running these scenarios, it is possible to extract and analyze the code blocks that represent a given feature. In order to generate scenarios, one must have domain and application knowledge. They are also derived from relevant documents to the system [31]. This technique helps both, in locating features in source code but also in increasing software comprehension and the result depends on the test scenario quality from which the execution traces are collected. It could lead to difficulties in industrial projects because of non-existent execution environments for legacy systems [19].

This category tackle the migration process from a top-down approach where it gathers information from running software as oppose to static analysis strategies that uses a bottom-up approach which means the source of information comes from the source code.

2.3.3.3 Expert Driven

An expert driven strategy means that persons involved possess a level of expertise, mostly on the system and domain in focus of the reengineering process [32]. The experts involvement is often to evaluate strategies and analyze results, this can involve software engineers, software architects, developers, stakeholders, etc. Hence, these types of strategies can be very resource intensive. The experts can also be involved during any phase of the process to finish the migration quicker [33].

2.3.3.4 Information Retrieval

Similar to Static and Dynamic Analysis are these strategies concerning the detection and retrieval of software features in an existing system. For Information Retrieval is it usually done in four steps [34]:

1. Search for common artifacts
2. Group detected artifacts into configurable components
3. Identify the variabilities and the dependencies of features
4. Create feature model

All of these steps can be accomplished in various ways. This usually depends on research preference, area of expertise, artifacts available (such as source code, documentation) and the tools in their disposal. For instance, commercial tools can be used for information retrieval or it can be done manually if not tools are available. Various strategies under Information Retrieval are Latent Semantic Indexing (LSI), Concept Analysis (CA), Execution Scenario (ES) and Trace Intersection (TI).

The main difference between this category and Dynamic and Static Analysis categories is that strategies within this category focuses on semantics. This means that these strategies treats the source code as a document [35]. Such strategy will detect commonly used words in textual artifacts and hence helps in identifying commonalities. However, these strategies suffer from obvious drawbacks which are polysemy (a word with several different meanings), synonymy and keywords that are either misspelled or abbreviated [35].

2.3.3.5 Search-based

Search-based Software Product Line Engineering (SBSPLE) is the intersection between Search Based Software Engineering (SBSE) and SPLE. This intersection is especially useful when a software system contains a large amount of feature with complex relationships [36].

Relating SBSPLE to Pohl's SPLE framework [37], SBSE is mostly used during the Domain Testing to test different feature combination derived from the feature model, or during Application Requirement Engineering [36]. This is used to detect any dead feature and / or test satisfiability of the feature model created.

2.4 Cost Models

One of the main issues with the implementation of a software product line is that it requires a large upfront investment [38]. This makes organizations hesitant to migrate their existing systems. For this reason, it is important to estimate the efforts of the reengineering and more importantly to break down the migration process into activities in order to pinpoint the most resource-intensive activities.

Estimations of monetary values are not measured in this study. However, we estimate various effort metrics needed to accomplish the reengineering process. This can be, for example, the duration of each activity. The estimation is given in how many person hours it takes to finish an activity. To identify relevant effort metrics, the logging metrics are based on previous, well-established cost models used in SPLE: SIMPLE [39], COPLIMO [40] and InCoME [41]. Metrics from the cost models that are most useful for our scenario is taken into consideration during the measurement design process (see Section 3.4).

The following sub-sections gives a short introduction to each of the cost models and their approach to estimating SPLE costs, while Section 3.4 and Table 4.2 describes and shows how metrics are mapped to the measurement design.

2.4.1 SIMPLE

While most cost models offer calculation-based results, SIMPLE pinpoints the important tasks in migrating a system to a SPL [39]. It defines four development costs:

- C_{org}: This entails organizational costs including the training and reorganization necessary before the implementation of SPL.
- C_{cab}: Core Asset Base costs concern the initial phase of reengineering; including commonality and variability analysis, architectural tasks etc...
- C_{unique}: This entails all product-specific requirements.
- C_{reuse}: This represents costs reusing assets, like testing and identifying assets to be reused.

Since the migration process is performed by the two authors, the organizational costs are insignificant in this research. However, the measurement design must take into account the three remaining costs when assessing the effort of the reengineering.

SIMPLE recognizes maintenance costs as the evolution costs of the SPL (C_{evo}). The most notable consideration here is C_{cab} which concerns the costs of updating the asset base. These cost measures are emitted in this research as our purpose is to reengineer an existing system into SPL, but no maintenance is done.

2.4.2 COPLIMO

COPLIMO is another cost model that is based on COCOMO II. While this model has been developed around aircraft and spacecraft domains, it has also been implemented and tested successfully on different domains [40]. COPLIMO focuses on two main costs: Relative Cost of Writing for Reuse (RCWR) and Relative Cost of Reuse (RCR). The former is concerned with the costs of developing software to be reused and the latter with the cost of reusing the software in a new or different product line.

This cost model considers a plethora of metrics that can be used for the creation of the logging artefact. Most notably, the Adaptation Adjustment Modifier (AAF) which includes Software Understanding (SU) which is affected by the Domain Analysis phase, it also uses the lines of code modified, known as Percent Code Modified (CM) and Percent Design Modified (DM).

2.4.3 InCoME

The Integrated Cost Model for Product Line Engineering (InCoME) is a cost model that is possible to use for different estimation scenarios because of several input parameters [41]. It has different layers that separates different kind of factors. There are three layers: *Cost Factors Layer*, *Viewpoint Layer* and *Investment Analysis Layer*. The first layer estimates costs that is forwarded to the next layers. Costs are estimated on seven factors:

- **Organizational:** Upfront investments to establish SPL infrastructure
- **Core Asset Base:** Costs to build reusable assets for a certain domain
- **Unique Parts:** Costs for developing unique parts of a product in a SPL
- **Reuse Level:** Level of reuse when using reusable assets in a product
- **Stand-Alone:** Costs to build a product outside of the product line
- **Product Evolution:** Costs to evolve a standalone product
- **Asset Evolution:** Costs to evolve the core asset base

When all costs are forwarded to the *Viewpoint Layer* those are calculated to show savings within the three PLE cycles, *domain engineering*, *product engineering* and *corporate engineering*. The results are categorized by viewpoints and afterwards, the third layer calculates for three economic functions *Net Present Value*, *Return on Investment* and *Payback Value*.

Results from the calculations are shown in Person Months or Person Hours. Important parameters that is used are: an *Investment Cycle (Y)*, a *Start Date (SD)* and a *Discount Rate (d)* that reflects time value of money.

3
Methods

This chapter describes the empirical case study with all steps involved. It also explains the collaborative aspect of this thesis. Below is an illustration of the steps performed during the methodology.

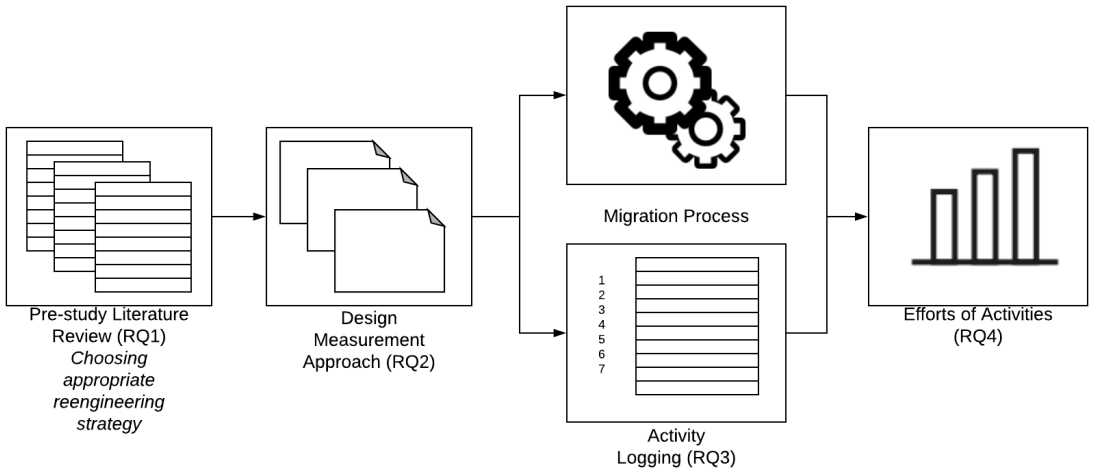

Figure 3.1: Illustration of Reengineering Process

At the start, a literature review is conducted to contrast and compare different strategies. Determining the pros and cons of different strategies helps selecting the most optimal strategy for the migration process. Thereafter, a measurement approach is designed by using well-established cost models in order to build a logging template. Once this is done, the migration process begins where the logging template is used to identify activities and log their efforts. Lastly, efforts of the entire migration process are presented based on the performed activities.

3.1 Collaboration

This empirical case study is performed in cooperation with another Master's Thesis. The other thesis is also conducting a reengineering of existing systems into an SPL, but with a different approach (annotative instead of compositional) and on a different dataset which means both studies will have a list of performed activities in the end. Their dataset consists of five Android games provided by the same developer as in this study. The strategy chosen for the actual migration process might

differ, since that is done individually, but the measurement approach mentioned in section 3.3 is designed together in order to have the same approach when estimating different efforts. This helps in the end-process where the two studies compares their results, since there is similar level of details in the measurements.

As the migration strategies of the two theses can differ, the activities making up these strategies can differ too which may complicate the process of comparing the results of both theses. To counter this, activity types are defined, where each team will tag each activity with one or more activity types. Once the migration is finished, activities with the same types are compared. The activity types are listed in Section 2.2.2.

3.2 Dataset

The dataset used in this study is a collection of Java games provided by ApoGames[1]. There are 20 Java games and five Android games provided where each game consists of 3000 to 10000 lines of code (LOC). To make our research comparable to the collaborators' thesis, the size of our dataset is limited to their. Since their dataset only contains five Android games, we select five Java games to facilitate the comparison between the two theses.

The Java games serve as a valuable dataset for this migration process since all of them have common software artifacts that can be found in most software systems. For instance, all of the games have a user interface, persistent data and also a complex logic layer that defines the game rules. Moreover, this dataset has been used in previous SPL research [12]. To add, the complex logic layer adds another step of complexity to this research since the layer further complicates the understanding of the code which is usually the case in an industrial setting [42].

3.2.1 Selection Process of the Five Java Game Variants

The first step in selecting games was to reverse engineer Java code from Jar files. This process failed for some of the game variants. In other variants the generated files did not compile, hence these games were excluded.

The second step was to run and test the games. In this step, several games crashed while performing some functions such as starting the game editor, or loading a game. This reduced the number of variants to 12. Furthermore, games that exceeded the 10000 lines of code were eliminated since they were considered larger than the Android games.

From the games that performed without error, they can be divided in two categories regarding their controls. Most of these games used a keyboard, hence these commonalities were taken as an advantage. Additionally, games with high level of

[1] ApoGame website: http://apo-games.de/

variabilities were excluded, for instance games with no menu, or with no actual player (such as ApoSuduko).

The selection process showed that many of the variants are very different. They also contain technical problems appearing at compile-time and execution-time. Hence, five games remained which had both common and different features, but also shared a similar project structure. The games are the following (*Variant ID* - *Variant name* (*x LOC*)):

- V1 - ApoCheating (3960 LOC)
- V2 - ApoIcarus (5851 LOC)
- V3 - ApoNotSoSimple (7558 LOC)
- V4 - ApoSnake (6557 LOC)
- V5 - ApoStarz (6454 LOC)

3.3 Selection of a Migration Strategy

Every researcher adopts their own approach to detect, analyze and transform an existing system into an SPL. Hence, after conducting the pre-study (see Section 2.3), one can see that there is no concrete strategy that is well-established or applicable on all SPL migrations, this goes in line with the conclusion of Assuncao et al[3]. Table 4.1 summarizes pros and cons found during the pre-study.

3.3.1 Applicability of Existing Strategies

While there seems to be a plethora of strategies and previous attempts at SPL implementations using an extractive approach, the applicability of these strategies is low. This is because it is dependent on the available resources, where in this study, the only resource is the variants source code. When it comes to Expert Driven strategies, none of the authors in this thesis is considered an expert in the domain hence these strategies are not applicable. Additionally, these strategies consider a large amount of experts in different area of the product lines, for instance, domain experts, engineers, testers and so forth which are not present in this research.

For strategies that are described and applied in literature, they usually utilize tools that are either discontinued or not possible to launch such as *LEADT* [43], *CIDE* [44] and others are outdated or commercial such as *BigLever Software Gears* [45]. In addition, even with the availability of one tool But4REUSE [46], which is a tool focusing on bottom-up technologies that detects commonalities between variants, the output given by the tool does not provide any useful information about our dataset. This is because the tools infers variabilities and commonalities from several inputs; in our case, the only input is the source code. Hence, the output is just a collection of most used keywords which can range from class names to variables and function names, the output of the tool is shown in 3.2.

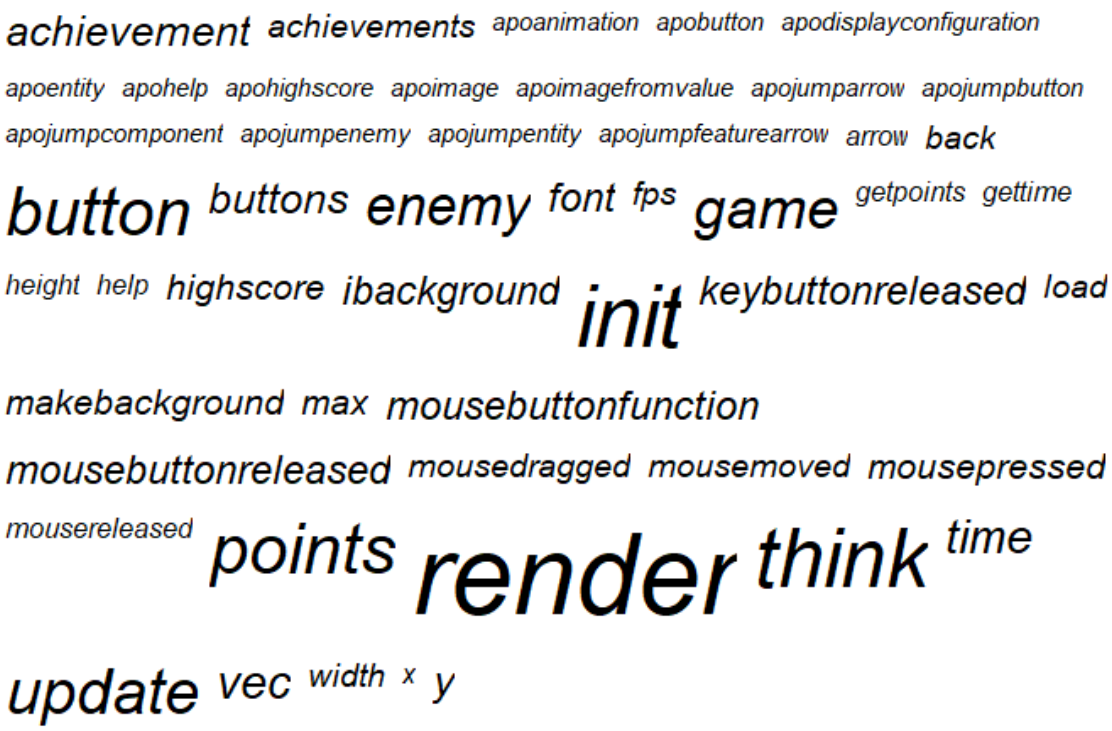

Figure 3.2: Output of running one example variant via But4Reuse tool

Moreover, But4REUSE was expected to provide accurate results according to previous literature [5]. Figure 3.3 and 3.4 shows outputs from tests with other strategies, which gave nothing that could help during the migration process. It did just list new features for each of the files and none of the variants were using the same feature.

	Block 00	Block 01	Block 02	Block 03	Block 04	Block 05	Block 06	Block 07	Block 08	Block 09	Block 10	Block 11	Block 12	Block 13	Block 14	Block 15	Block 16	Block 17	Block 18	Bloc 19
apoStarz%2Feditor%2FApoStarzEditor.java	X																			
apoStarz%2Fentity%2FApoStarzBlock.java		X																		
apoStarz%2Fentity%2FApoStarzEntity.java			X																	
apoStarz%2Fentity%2FApoStarzFire.java				X																
apoStarz%2Fentity%2FApoStarzGoal.java					X															
apoStarz%2Fentity%2FApoStarzStar.java						X														
apoStarz%2Fgame%2FApoStarzGame.java							X													
apoStarz%2Fgame%2FApoStarzHighscore.java								X												
apoStarz%2Fgame%2FApoStarzHighscoreLevel.java									X											
apoStarz%2Fgame%2FApoStarzIO.java										X										
apoStarz%2Fgame%2FApoStarzTutorial.java											X									
apoStarz%2Flevel%2FApoStarzLevel.java												X								
apoStarz%2Flevel%2FApoStarzLevelLoad.java													X							
apoStarz%2Fsolver%2FApoStarzSolver.java														X						
apoStarz%2FApoStarzApplet.java															X					
apoStarz%2FApoStarzConstants.java																X				
apoStarz%2FApoStarzImages.java																	X			
apoStarz%2FApoStarzMain.java																		X		
apoSnake%2Fentity%2FApoLevelChooserButton.java																			X	
apoSnake%2Fentity%2FApoSnakeEntity.java																				X

Figure 3.3: Output during Formal Concept Analysis on two variants

If the output seen in these figures would be correct, our dataset would not contain any commonalities. Hence, it was safe to assume the tools did not function properly. Figure 3.4 shows some commonalities (yellow color), but that does not reflect what the tool is supposed to highlight.

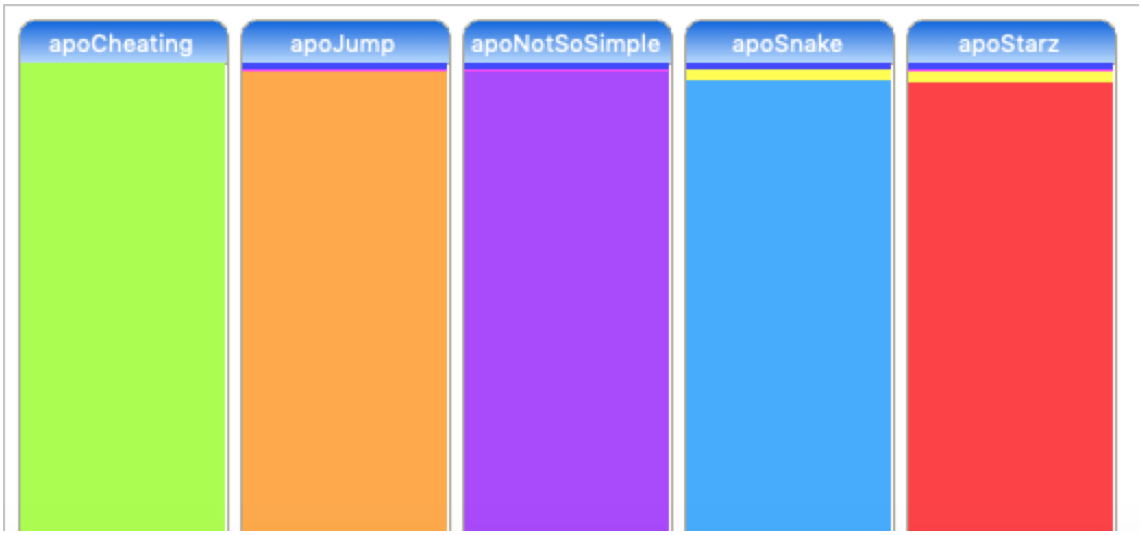

Figure 3.4: Feature identification where no variant uses the same feature (color)

After noticing commonalities, a word cloud representing the yellow feature was created. It showed that the part with most commonality was a *.gitignore* file. This showed once again, as previously stated, that the output was not helpful. Given the poor results provided by the tools, the following Section 3.3.2 describes what strategy and approach that is used for the migration process.

3.3.2 Choosing an appropriate migration strategy

As migration strategies depends on tools and resources available, the migration process for this thesis depends on strategies that are solely dependent on source code (only available resource from the existing variants), which is also the case for 50% of industrial SPL implementations [5]. Since the source code of the variants is also executable, this means that both a *top-down* and a *bottom-up approach* can be adopted. In literature, this is known as a *sandwich approach* [47]. This means that some aspects of dynamic and static analysis can be tailored to better suit our data, this hybrid strategy is also recommended by the literature[24],[25]. For instance, to execute a top-down approach, the software can be executed to identify features on the domain level. This means that each game is played and all the features are explored, mapped and compared between variants to identify common features on the domain level. This can be described as a dynamic analysis strategy. From this, a preliminary feature model can be created.

An additional strategy to identify commonalities between variants is to conduct pairwise comparison of the source code and code structure. This is a form of static analysis strategy. Moreover, systematically reading the source code helps locating the identified feature in the source code. This is combined with yet another dynamic analysis strategy, where breakpoints are used at individual source code statements responsible for each feature.

Furthermore, another static analysis strategy that helps in understanding the relationships and hierarchies among features, is to reverse engineer class diagrams from the source code and analyze all the associations and nesting of different source code artifacts.

3.4 Design of the Measurement Approach

This section has been collaboratively written by the two Master's Thesis groups. The measurement approach was designed together by having three joint meetings and discussing how to log each activity. This is so that a triangulation between acquired data from the two teams can be made. The actual strategy might be different, which means that activities involved might differ as well. The triangulation of data is therefore only made for common parts, based on activity types. If no similar data exists, a comparison of the outcome of the studies is the only thing that is made to compare the SPLs.

The metrics that are used in the log are based on the previously described (see Section 2.4) well-established cost models SIMPLE, COPLIMO and InCoME [39][40][41] and their respective metrics. The designed approach and the data that is collected helps in answering RQ.2.

A log entry contains information about an activity. Each activity that is written down has a unique ID and a name in order to be able to identify it. Activity IDs

can be useful once the migration process is finished when presenting different data. There is also a start and end date for when the activity was performed along with a detailed description about the activity. The description explains the activity at such level of detail so that a person can understand what has been done in that activity. One should also know which tools has been used in the activity.

The COPLIMO and InCoME cost models estimates effort in terms of person month [40][41], logging of activities is therefore made as a minimum each person day. Since an activity can take longer than one person day to finish, the log for an activity will not be complete until the activity is finished. This means that the log is updated in an iterative process, where for example number of hours are incremented and other fields are filled in when appropriate. Data that can be derived and logged from a person day are: hours spent and in what environments those were spent in, e.g. in an IDE. This logging of duration is also based on the InCoME cost model where it takes into consideration the start and end date of the investment in its calculation of SPL migration costs [41].

Other data in a log entry consists of number of LOC and files that has been added, removed or edited. This is based on COPLIMO that uses parameters such as percent of code that has been modified (CM) and portion of software that must be modified to work well (AFRAC) [40]. Metrics such as number of LOC can help in trying to describe how large an activity is, as well as its complexity. This data is taken from commits, which means a log also contains number of commits and commit IDs. Commit frequency follows common version control practices and thereby be of small size and commits occur often [48]. Not only are the commits of a small size as advocated by common practice, each commit also follows the Angular commit message guidelines [49] to increase consistency. This means that there is a certain format one should follow, it includes a heading, body and a footer where the heading has a type, a scope and a subject. A template of how a commit message should be written can be seen below.

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

The logging template also includes information about what tools that have been used during the activity, e.g. a plugin for the Eclipse IDE. It also shows input and output artefacts for an activity, where an input could be the source code and output is a class diagram. The artifacts show what have been used in order to perform the activity, as well as what the activity produced.

Lastly, notes about an activity are written down. In this area, personal experience is expressed about the importance and complexity of the activity as well as if there are any dependencies to other activities that has been performed prior to the current. This is served as qualitative data.

The final template for logging activities can be seen in the Result chapter 4.2 together with an example activity in Appendix A.2. See Table 4.2 for all mappings between cost models and the logging template.

3.5 The Reengineering Process

All necessary preparations are completed from the previous steps and the implementation part of the reengineering process can begin. During this part, activities involved in a compositional SPL reengineering are identified, hence answering RQ.3 while logging each activity in parallel. Once this is done, the logging data is analyzed to answer RQ.4. As mentioned in Section 3.3.2, the reengineering process is conducted with a hybrid strategy. The strategy includes:

- Running the Java games
- Mapping features to the domain level
- Creating feature model
- Reverse engineering class diagrams
- Pairwise comparison of variants
- Systematic source code reading

The strategy does not strictly follow the three phases as they are from top to bottom. Activities from different phases are combined during different phases in the reengineering process. This means there might be an activity categorized as the detection phase that is combined iteratively with another activity from the transformation phase. The phases are therefore conducted in an agile way, instead of a waterfall approach.

3.5.1 Detection phase

Activities related to this phase mainly consists of the top-down approach. They try to identify features on domain level, which in later phases are used in combination with activities that takes a bottom-up approach.

3.5.1.1 Running games

In order to get an understanding of the games and their domain, all the five game variants were played as a first activity. Notes were taken for any observations of potential features. The notes were a description of each game and a bullet list of features. Each feature got a description to show how that feature works in that particular game, while the feature name was more abstract so that it potentially could be used on domain level. An example of how notes could look is seen below.

GameName
description of the game

Features
- Character
  - Move using arrow keys on keyboard
  - Press and hold space bar to perform an action
- Enemy
  - Some stand still, some travels left/right
  - Game is lost if character touches enemy

3.5.1.2 Mapping features to domain

The data retrieved from running games was compared game by game in order to find commonalities. By doing this, features that are common in many of the variants can be mapped to domain level. Comparisons of feature names as well as feature descriptions help when trying to understand what a possible domain feature could be. This could be seen as a top-down approach, where the game information extracted is similar to documentation. This process resulted in a list of domain-level features that can be seen in Section 4.3.1.1.

3.5.1.3 Creating a feature model

Creating the feature model was an iterative process, as it needed to satisfy the configuration of the five variants and also cover all the features that were provided in these variants. Additionally, the feature model must also be flexible to support the creations of other products. The feature model was created using FeatureIDE which is an Eclipse plugin for feature-oriented development [50].

The approach switches to bottom-up in later stages of the reengineering process. Hence, the feature model might need to be redesigned. This makes it important to be open for change and thereby have an iterative mindset as mentioned earlier. Even though, at this stage, features in the feature model accurately shows what exists in the games it might be difficult to extract source code to match these features. The results of the feature model can be found in Appendix A.6.

3.5.1.4 Reverse engineering class diagrams

To have a better understanding of code artifacts such as classes, methods and variables and their relationship, class diagrams were generated from the source code using Visual Paradigm [51]. This process helped in identifying commonalities and variant specific parts for the games on the source code level by finding similar class and method names in the variants. By reverse-engineering class diagrams it also gave an overview of the variants' architecture. Furthermore, inspecting dependencies between Java classes can help in understanding how a feature can scatter across different classes.

3.5.2 Analysis phase

With the help of the class diagrams, feature model and Eclipse IDE, the starting point of every feature is identified. By using breakpoints, it is possible to map features and how they spread in the source code. Every feature is appointed to a starting class and starting statement, and all related classes where the feature is also executed. All this information is logged and later used in the transformation phase.

3.5.2.1 Pairwise Comparison of Variants

In order to further study commonalities between variants, tools were used to compare class names and class contents of all variants against each other. The tool used was Code Compare[2], where it provided three types of outputs which can be seen in Figure 3.5:

- Identical class names and contents - White color code
- Identical class names but different content - Blue color code
- Unique class names - Red&Green color code

By using the tool, multiple classes could immediately be classified as identical. Moreover, the five variants studied in this study come from a collection of over 20 Java games with similar file structure. This means that a portion of the duplicated code can be unused by these five variants and used in other Jave games which were not selected in this study. For this reason, the source code is analyzed to detect any dead code, this analysis is explained in Section 3.5.2.2.

Pairwise comparison was used in later stages in combination with other activities as well, but with another focus. Initially, without any activity combination, the comparison focused on a top-down approach where only the tool (Code Compare) output was of importance. Meaning that if two variants had a class with the same name but different content, the differences were not studied but only the class was flagged as different. When combined with other activities, the pairwise comparison took a bottom-up approach instead, where a file from each variant was compared in detail to see at which LOCs were similar and which were different. More information about this process can be read in the subsection 3.5.2.3. Figure 3.5 shows how pairwise comparison could look during the first iteration. Hence, the tool output during the first iteration was on a high level. Some notes taken from these outputs can be seen in Figure 3.6

[2]Code Compare website: https://www.devart.com/codecompare/

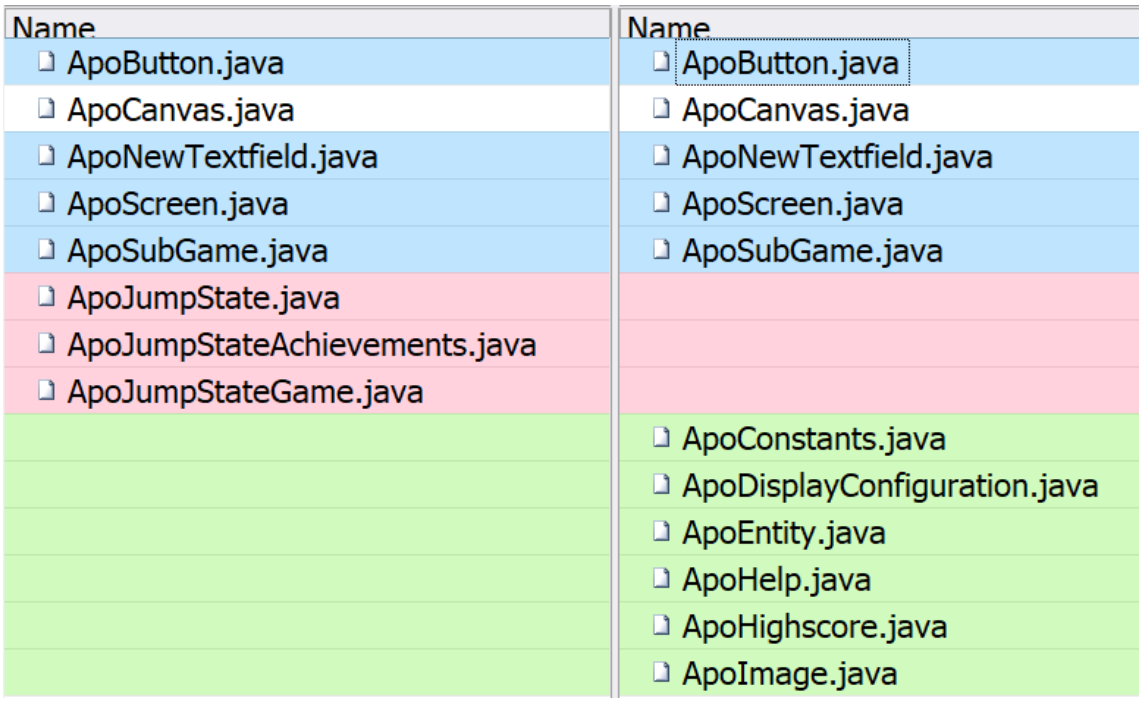

Figure 3.5: Example pairwise comparison. Blue: Same name different content, White: Identical, Red and Green: Different file names unknown content

Variant #	Variant 1	Variant 2	Variant 3	Variant 4	Variant 5	c(2,3)	c(2,4)	c(2,5)
Variant Name	ApoCheating	ApoIcarus	ApoNotSoSimple	ApoSnake	ApoStarz			
Total Number of Classes	49	59	57	58	49			
Number of Identical Classes						24	17	10
Number of Class with same Name						7	13	11

Figure 3.6: Notes in an excel sheet from the pairwise comparison

3.5.2.2 Code Cleansing

This part of the migration process was not planned, however, when inspecting the code and running the game, it was clear that the source code contained a lot of dead code. Failure to remove such code can be costly as we would have spent time and effort transforming code blocks that would have no positive impact on the SPL.
An Eclipse plugin called UCDetector (Unnecessary Code Detector) was used to detect dead code. This activity also made the pairwise comparison easier since all the code that remains is used in at least one feature. Locating the dead code had to be done in an iterative manner. UCDetector scans each variants' source code where it finds methods and classes that are never referenced. It creates a *.html* file with all findings. An excerpt of the output can be seen in Figure 3.7.

Nr	Java	Marker	Description	References**	Author	Location*
1			Change visibility of Class "ApoJumpImages" to default - May cause compile errors!	-		apoJump.ApoJumpImages.declaration(ApoJumpImages.java:14)
2			Method "ApoJumpImages.getLogo(BufferedImage)" has 0 references	0		apoJump.ApoJumpImages.getLogo(ApoJumpImages.java:40)
3			Method "ApoJumpImages.getImageMouseOver(BufferedImage)" has 0 references	0		apoJump.ApoJumpImages.getImageMouseOver(ApoJumpImages

Figure 3.7: Example of how UCDetector indicates dead code in its .html file

This file is read manually and the code regarding e.g. a non-referenced method is

then removed manually as well. Every time a part has been removed, the variant is compiled and ran to ensure nothing broke by removing the code part. When all dead code had been removed, UCDetector was executed again on the same variant. Some of the removed source code might have had references to other methods or variables and when that code was removed, new dead code would emerge. This process kept going until a scan by UCDetector showed that no dead code exists. The steps performed to find and remove all dead code for each of the variants were:

- Scan variant with UCDetector
- Open generated .html
- Identify class/method/variable with 0 references
- Go to relevant LOC and remove it
- Compile and run variant to make sure nothing stopped working

Another activity involving code cleansing was to translate code comments. All the original comments in the source code were written in German, hence these were translated into English to improve the software comprehension phase.

3.5.2.3 Systematic Source Code Reading

All the variants' source code were systematically read. First, to get an overview by going through all classes to try and understand what role they have. To get a more detailed view afterwards, a low level approach were used with debugging to understand algorithms. In order to try and understand what code is relevant for a certain feature, the systematic reading was combined with other performed activities. By doing this, the focus of systematic reading mainly was about mapping features to source code. Relevant code for a feature was identified by first using the feature model to choose what feature to find and then applying what could be seen as Dynamic Analysis. A variant was put into debug-mode and breakpoints were activated for all methods within a class. If the execution stopped at some point while the game variant performed the previously chosen feature, notes were taken about that breakpoint such as class and method name. Figure 3.8 shows parts of the derived methods and classes in one of the variants for the *Menu* feature.

ApoNotSoSimple	ApoNotSoSimpleMenu.java	init(),render(Graphics2D),think(int)
	ApoNotSoSimpleModel.java	getGame(),mouseMoved(int,int),mousePressed(int,int,boolean)
	ApoNotSoSimpleButtons.java	init()
	ApoNotSoSimplePanel.java	init(),setMenu(),setButtonVisible(boolean[]),mousePressed(int,int,boole
	ApoNotSoSimpleComponent.java	renderButtons(Graphics2D),isShowFps(),load(),setShowFps(boolean),
	ApoNotSoSimpleOptions.java	init(),render(Graphics2D),mouseButtonFunction(String)
	ApoNotSoSimpleOptionsButton.java	init(),render(Graphics2D,int,int)
	ApoNotSoSimpleLevelChooser.java	init(),render(Graphics2D),mousePressed(int,int,boolean),mouseButton
	ApoNotSoSimpleUserLevels.java	loadHighscore()
	ApoNotSoSimpleCredits.java	init(),render(Graphics2D),mouseButtonFunction(String)

Figure 3.8: Notes from feature location for the *Menu* feature

As mentioned earlier, the pairwise comparison activity was used in combination with source code reading. Figure 3.9 shows an example of two variants with a similar file name and structure, but an *init()* method with variant specific source code.

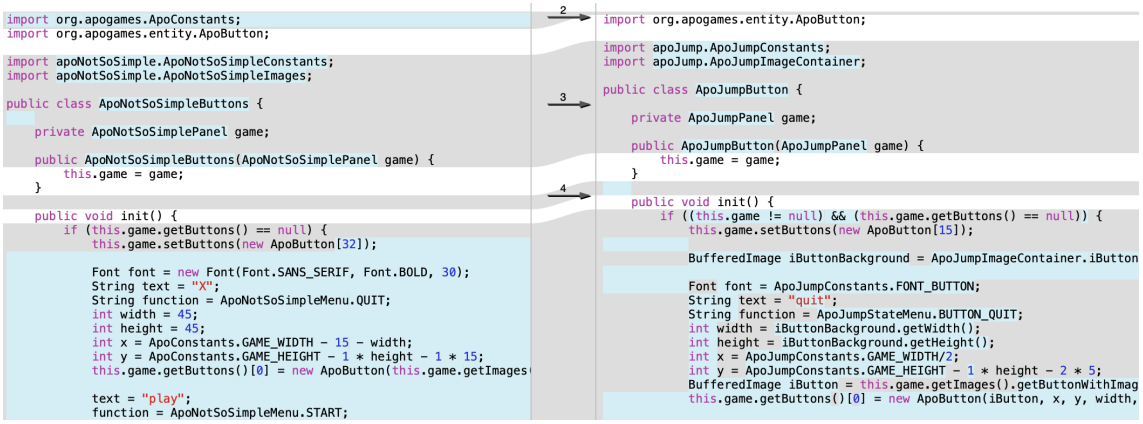

Figure 3.9: Example of how a detailed pairwise comparison could look

Additionally, Java classes were manually inspected where it was suspected that there would be similarities that the tool Code Compare could not detect during the first iteration of pairwise comparison. For instance, Java classes called *Variant1Panel.java* and *Variant2Panel.java* could serve the same purpose for different variants (initiating the panel). Hence, these files were compared and commonalities were extracted as well. However, this is not always the case, since the developer did not always use the same naming in all variants, hence *VariantXPanel.java*, might be called *VariantYModel.java* in another variant, which further complicates the process.

3.5.3 Transformation phase

The process of transforming variants into a Compositional SPL was iterative. Focus was mainly on one variant at a time, where code was refactored and extracted into one feature at a time. Because of the process' iterative focus, Sections 3.5.3.2 and 3.5.3.3 summarizes what was done repeatedly during the transformation phase.

3.5.3.1 Setting up a Product Line

The first step in the transformation phase was to set up the structure by generating all the feature folders using the previously created feature model. Every folder represents a feature and the top feature from the feature model represent the most common feature. The top feature would be where files that are the same between variants are stored. As one go down in the feature model, the commonality decreases. Parts of the project structure and the feature folders can be seen in Figure 3.10. The Product Line was implemented in Eclipse [52] with the plugin FeatureIDE [50] and FeatureHouse [2].

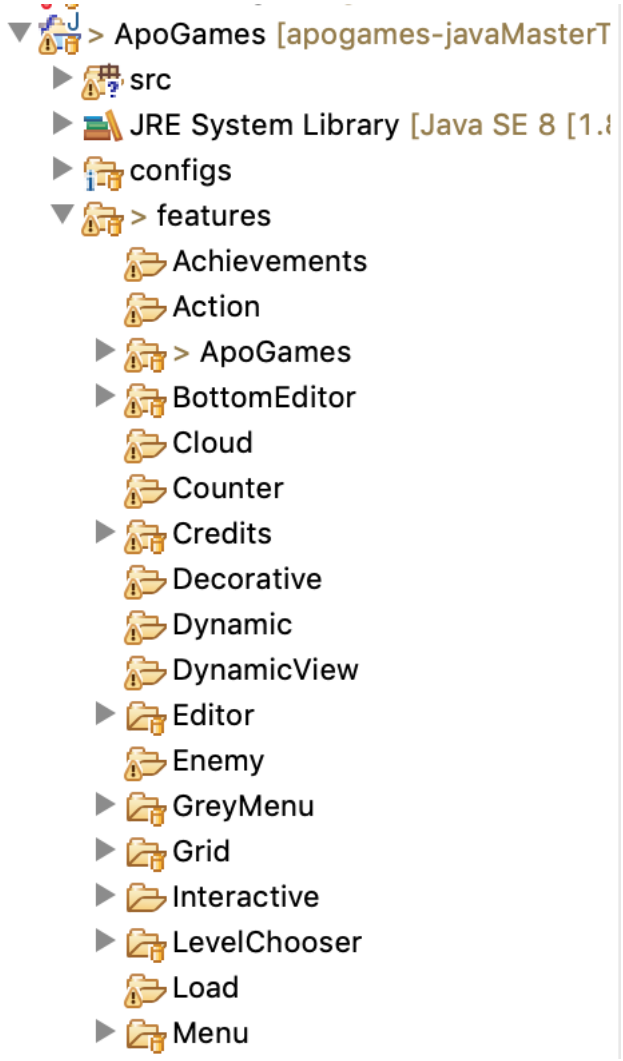

Figure 3.10: Parts of the project and its feature folders

3.5.3.2 Extracting Features

The information gathered during diffing and systematic code reading was used when features were going to be extracted from a variant into the SPL. With information about similar files, one file at a time from different variants were opened again as they were during the systematic source code reading. They supported the early process of extracting common code into the top feature, called *ApoGames* (See Figure A.2 for feature model). Figures 3.11 and 3.12 shows parts from the combined diffing and systematic code reading. As seen in the figures, a LOC that have grey highlighter is identical and a non-grey LOC is not.

```
public ApoButton( BufferedImage iBackground, int x, int y, int width,
    super(iBackground, x, y, width, height);

    this.function   = function;
    this.bOver      = false;
    this.bPressed   = false;

    this.wait       = 0;
    this.maxWait    = 0;
    this.bWait      = false;
    this.bFirstWait = true;
}
```

Figure 3.11: Parts of ApoButton.java in variant V3

```
public ApoButton( BufferedImage iBackground, int x, int y, int width,
    super( iBackground, x, y, width, height );

    this.function   = function;
    this.bOver      = false;
    this.bPressed   = false;

    super.setBOpaque(false);
    this.wait       = 0;
    this.maxWait    = 0;
    this.bWait      = false;
    this.bFirstWait = true;
}
```

Figure 3.12: Parts of ApoButton.java in variant V4

All common code was extracted into a feature high up in the tree hierarchy, since this would be a feature used by any feature-configuration. Because of how superimposition works, variant specific source code could then be extracted to correct feature further down in the feature model in a class with the same name as in the *ApoGames* feature. Source code in that feature would thereby be appended to the same file in the generated source code. By doing this, a configured game from the feature model would maintain same structure as an original variant with packages and classes. Figures 3.13 and 3.14 show the package structure similarities between a generated game from the SPL and one of the original game variants (V4). This similarity can be justified as, while specific SPL architecture methodologies exist [53], SPL can still carry the same architecture as that of a single-system [9].

Figure 3.13: Package structure for a SPL generated game

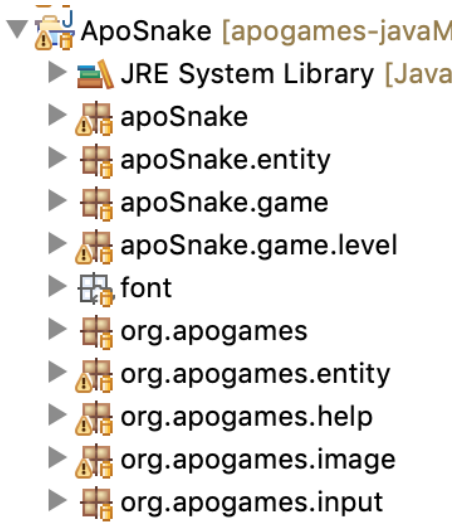

Figure 3.14: Package structure for an original game

3.5.3.3 Feature Refactoring

Many parts of the original code were hard-coded. An example is an array representing what buttons were supposed to be visible. A certain position in the array corresponded to a specific button, which led to *NullpointerException* when a feature-configuration with other buttons than an original game variant was picked. This meant that in order to be able to configure a game without a certain part, or switch to a similar part from a different variant, refactoring had to be done. For this particular problem, data structures and types were changed. Figure 3.15 shows original code where a button in the menu is added at index five.

```
text = "editor";
function = ApoNotSoSimpleMenu.EDITOR;
width = 250;
height = 60;
x = ApoConstants.GAME_WIDTH/2 - width/2;
y = ApoConstants.GAME_HEIGHT/2 + 70;
this.game.getButtons()[5] = new ApoButton(this.game.getImages().getButtonImageSimple(width * 3, height, text,
```

Figure 3.15: Original code for storing buttons

The variable storing buttons was turned into a *HashMap<String, ApoButton>*, where the string is the button function. This changed instantiating of a button to what can be seen in Figure 3.16.

```
text = "editor";
function = ApoGameMenu.EDITOR;
width = 250;
height = 60;
x = ApoGameConstants.GAME_WIDTH/2 - width/2;
y = ApoGameConstants.GAME_HEIGHT/2 + 70;
this.game.getButtons().put(ApoGameMenu.EDITOR, new ApoButton(this.game.getImages().getButtonImageSimple(width * 3, height,
```

Figure 3.16: Refactored code for storing buttons

When buttons were changed to be more dynamic, methods had to be refactored as well. Figures 3.17 and 3.18 show changes for a method making buttons visible. The original method takes a list of buttons that should be visible and sets them visible. But because the *HashMap* now contains several extra buttons that were not included in the hard-coded array, all buttons first needs to be set invisible in the refactored version.

```
public void setButtonVisible(boolean[] bVisibile) {
    for (int i = 0; i < this.getButtons().length && i < bVisibile.length; i++) {
        this.getButtons()[i].setBVisible(bVisibile[i]);
    }
```

Figure 3.17: Original method to show buttons

```
public void setButtonVisible(ArrayList<String> bVisibile) {
    for(ApoButton button : this.getButtons().values()) {
        if(button.isBVisible()) {
            button.setBVisible(false);
        }
    }
    for(String button : bVisibile) {
        this.getButtons().get(button).setBVisible(true);
    }
```

Figure 3.18: Refactored method to show buttons

Moreover, code regarding different features had to be moved around as well. When a new variant, sometimes even a new feature, was going to be integrated into the SPL, parts of already implemented code could be necessary to move further down the feature model. This was because what previously counted as common code was no longer common, since the newly implemented had differences. This could be reflected in the feature model as well, where a feature might have been split up into two or a feature received sub-features.

4
Results

The chapter presents results gathered during the reengineering process, as well as from the preparatory steps that were made before the migration.

4.1 Advantages and Drawbacks of Strategies

From reviewing literature, it is clear that there are a plethora of approaches to implement SPLE. Each study takes a slightly different approach that is influenced on several factors. However, all approaches can be gathered in five categories, Table 4.1 provides a summary of the advantages and drawbacks of these categories.

CATEGORY	PROS	CONS
Expert driven	Overall lower rate of failure since experts are involved.	Not applicable for non-experts.
Static analysis	Widely used technique, with established tools.	Tool dependent, otherwise time consuming. Lack of research in transformation phase.
Dynamic analysis	Identifies code execution traces, good to find scattered/tangled features.	Results depend on quality of test scenarios, difficult to apply on legacy systems.
Information retrieval	Ability to deal with large amount of data. Result is often refactored source-code.	Requires high-level artefacts as input to perform well.
Search based	Mostly used for creating variability models.	Deals mostly with analysis phase, complex activity. Does not output a feature map.

Table 4.1: Table summarizing advantages and disadvantages of the categories.

From our migration experience and relating to previous studies [3], we conclude that the most suitable strategy for any migration process is heavily dependent on several factors, which are described in detail in the Discussion section.

4.2 Measurement design

A template with the purpose of measuring activity efforts was designed by studying metrics of different cost models (COPLIMO, InCoME and SIMPLE) and extracting metrics that measure meaningful effort factors such as duration of activities, number of commits and so forth. The logging template is shown in Figure 4.1.

INFORMATION
- **Activity type**:
- **Activity**:
- **ActivityID**:
- **VariantID**:
- **Start Date**:
- **End Date**:
- **Description**:

DATA
- **Total Hours spent**:
- **Number of commits**:
- **LOC added**:
- **LOC removed**:
- **LOC modified**:
- **Number of files added**:
- **Number of files removed**:
- **Number of files modified**:

ARTEFACTS
- **Input**:
- **Output**:
- **Tools Used**:

ACTIVITY DESCRIPTION
- **Complexity**:
- **Importance**:
- **Dependencies on other activities**:

Figure 4.1: Logging Template

Table 4.2 shows the mappings between cost model factors and the effort metrics in the designed template. The third column represent what design measurement is extracted from which cost model factor (second column), while the first column states which cost model the cost model factor was taken from.

Cost Model	Cost Model factor	Measurement design
COPLIMO	DM - % of design modified	LOC and files added/removed/modified
COPLIMO	CM - % of code modified	LOC and files added/removed/modified
COPLIMO	AFRAC - portion of sw modified to work well	LOC and files added/removed/modified
COPLIMO	PM - person month	Logging every person day
COPLIMO	AA - assessing reusable components	Hours spent
COPLIMO	SU - understandability of software	Hours spent
COPLIMO	UNFM - unfamiliarity with software	Hours spent
COPLIMO	PSIZE - effort, SPL development cost	Hours spent/nr of commits/LOC
InCoME	PM - person month	Logging every person day
InCoME	Y - investment cycle	Start and End date
InCoME	SD - start date	Start and End date
InCoME	D - discount rate, time value of money	Start and End date
SIMPLE	CSWdev - full cost of developing SPL	Hours spent

Table 4.2: Table showing cost model factors and their mapping to the designed measurement template

4.3 Migration Process

This section summarizes the findings from the process of reengineering Java game variants into a compositional SPL. It is important to note that the migration process did not reach total completion, where only three out of five varaints were migrated (see Section 5.1.1 for more information). Additionally, no logging was done during the preparatory steps. This means that neither experimentation with different tools, nor all the effort spent while reading articles and preparing the measurement design were logged.

4.3.1 Activities

The migration process resulted in 12 activities. It is possible that the activities would differ if another migration strategy would have been taken. Table 4.3 shows each activity ID, along with its name and types.

Some activities in Table 4.3 did not affect the migration process. Hence, these are not considered as important nor an actual part of the migration process. It is activities filtered with activity type *SPLE Training* (Read about Compositional SPLs, FeatureHouse research and FeatureIDE research). Additionally, activities such as translating comments and removing dead code did not have any impact on the migration itself, however it did facilitate software comprehension and deleting the unused code sped up the migration process.

4.3.1.1 Running the Games

All variants were executed in the beginning of the project as a domain analysis and to identify features. This resulted in a list of features manually extracted from testing

Activity ID	Activity Name	Activity Types
A1	Running games	Domain analysis, Feature identification
A2	Mapping game features	Domain analysis, Feature identification
A3	Read about Compositional SPLs	SPLE Training
A4	Creating a feature model	Domain analysis, feature modeling
A5	Translating code comments to English	Code cleansing, feature identification
A6	Reverse-engineer class diagrams	Feature location, architecture identification
A7	Finding features in source code	Feature location
A8	FeatureHouse research	SPLE Training
A9	Pairwise comparison of variants	Diffing, feature location
A10	Removing unused code	Code cleansing
A11	FeatureIDE Research	SPLE Training
A12	Transforming source code to feature	Transformation, Quality assurance

Table 4.3: All activities performed during the migration process

the games. Features amongst variants were compared using this list. Appendix A.4 lists all notes, with the format seen in Section 3.5.1.1, about identified features along with a general description of each game. A mapping of game features (activity A2) was made by comparing the game notes seen in A.4. This resulted in a list of common features at domain level that can be seen below.

- Menu
- World
- Character
- Levels
- Highscore
- Editor

4.3.1.2 Creating the Feature Model

Creating the feature model was an iterative process, where features were added/removed along the transformation process. It ended up including 47 features. The final feature model is shown in Appendix A.2 followed by a list defining every feature. Since the migration process ended before all games were implemented, several features remained empty. Hence, features had to be set as *hidden* and additional constraints were added in order to generate products without errors. Appendix A.3 shows the modified feature model. 22 out of 47 features were implemented and this generated 56 different products.

4.3.1.3 Reverse Engineering Class Diagrams

Class diagrams were extracted in order to have a better understanding about the architecture used in different variants, and also to understand the relationships between different software artifacts and potentially detect dependencies between features. All five class diagrams are listed in Appendix A.7. It is clear that all variants follow a similar architecture except for ApoCheating. This increased the degree of variability in the SPL and ultimately, migrating this particular variant became more challenging, but due to time constraint, we did not migrate this variant.

4.3.1.4 Diffing

Results of the pairwise comparison show that around 30% of the source code is duplicated using a clone-and-own approach and some files proves the existence of evolution. Meaning that, for example, some methods have evolved in some variants, while other variants still has outdated methods. Hence, it is needed to manually update the identified methods. Results from the first round of diffing that took place during the top-down approach can be seen in Table 4.4. Unique files refer to file names that are different in each variant. Vx/Vy represents files for variant X and variant Y.

Variant ID	Total files	Identical files	Same name, different content	Unique files
V2/V3	59/57	24	7	28/26
V3/V4	57/58	17	9	31/32
V4/V5	58/49	10	12	36/27
V2/V4	59/58	17	13	29/28
V2/V5	59/49	10	11	38/28
V3/V5	57/49	10	12	35/27

Table 4.4: Comparison of files between variants

During code cleansing, by using UCDetector, it was possible to remove 11670 LOC from a total of 30380 LOC of all five variants. This means that almost 40% of the source code was never used. Another round of diffing was made on some of the variants after code cleansing, see Table 4.5. This was made in a bottom-up approach which led to a more detailed comparison, where the files were examined which gave another parameter: different name with similar content.

Variant ID	Total files	Identical files	Same name, different content	Different name, similar content	Unique files
V2/V3	43/38	7	8	12	16/11
V3/V4	38/32	0	13	14	11/5

Table 4.5: Detailed comparison of files between variants - after code cleansing

The amount of files necessary to examine reduced significantly by code cleansing, as seen by examining Tables 4.4 and 4.5. This reduces necessary efforts for the reengineering process.

4.3.2 Overview of the Migration Process

In Figure 4.2, one can clearly see that during the first few weeks of the reengineering process, there is a focus on every single variant. This is mainly to study and detect commonalities and variabilities. Furthermore, as we progress in the migration process, the focus shifts to a small number of variants as those with considerable variability (in our case, variant 1 (color blue)) are excluded. This was because V1 (ApoCheating) had a considerably different architecture, as previously mentioned in Section 4.3.1.3, than all other games. Hence, the focus was all on the other variants. Already in week 4, a common code base had been migrated from all four variants.

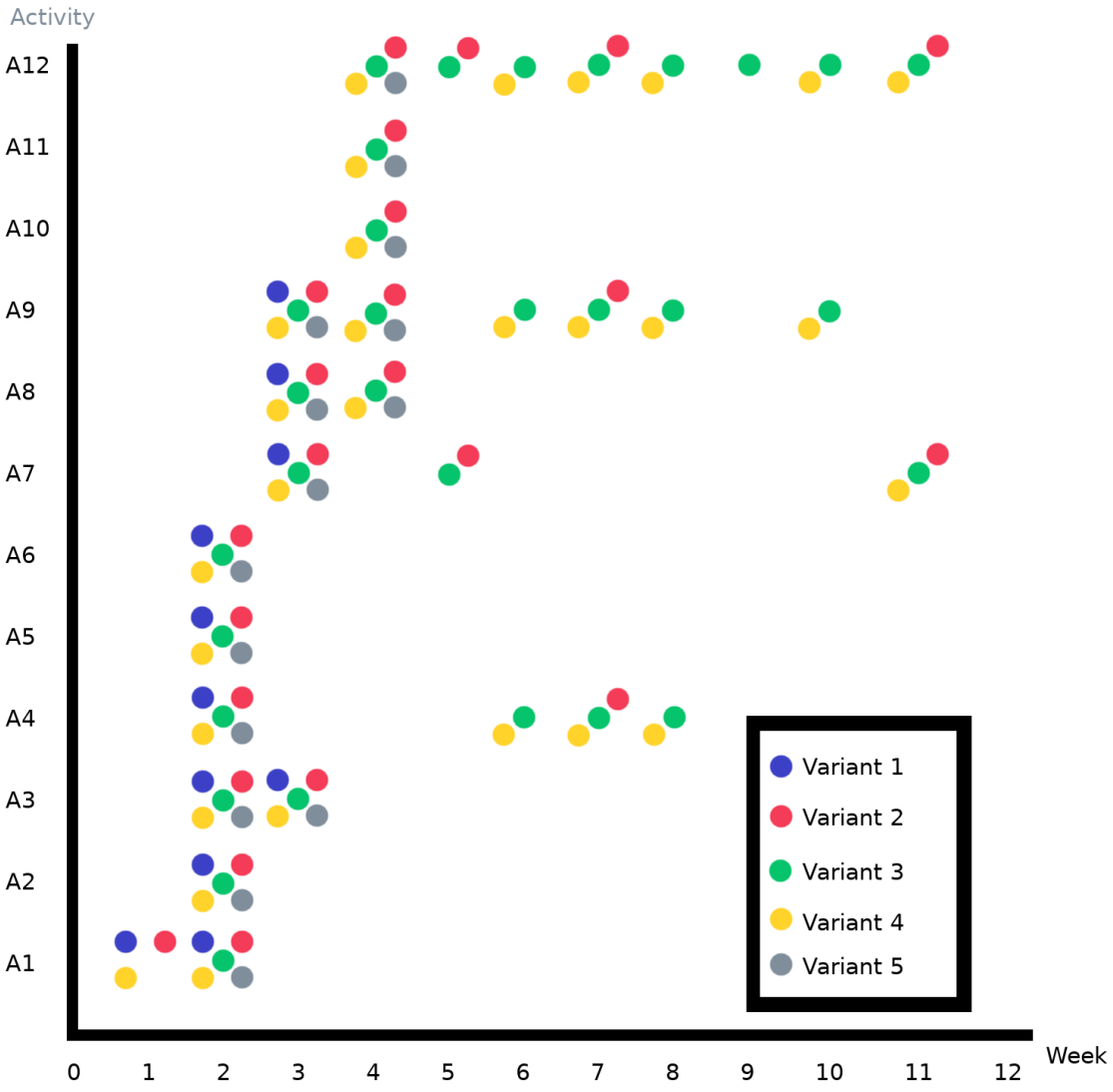

Figure 4.2: Overview of what activity and what variant was considered each week of the migration process

Later in week 5 till week 7, the diffing (A9) occurred where we examine which pair of variants have most commonalities, as those are the easiest to migrate into the SPL. This is where the transformation began between two variants, until week 10. When the two variants were successfully migrated, the third variant with most commonality was the one planned to be migrated next.

4.4 Activity Efforts

Reengineering three out of five variants into a compositional SPL took 371,5 person hours. Figure 4.3 shows how the total amount of hours are spread across each activity.

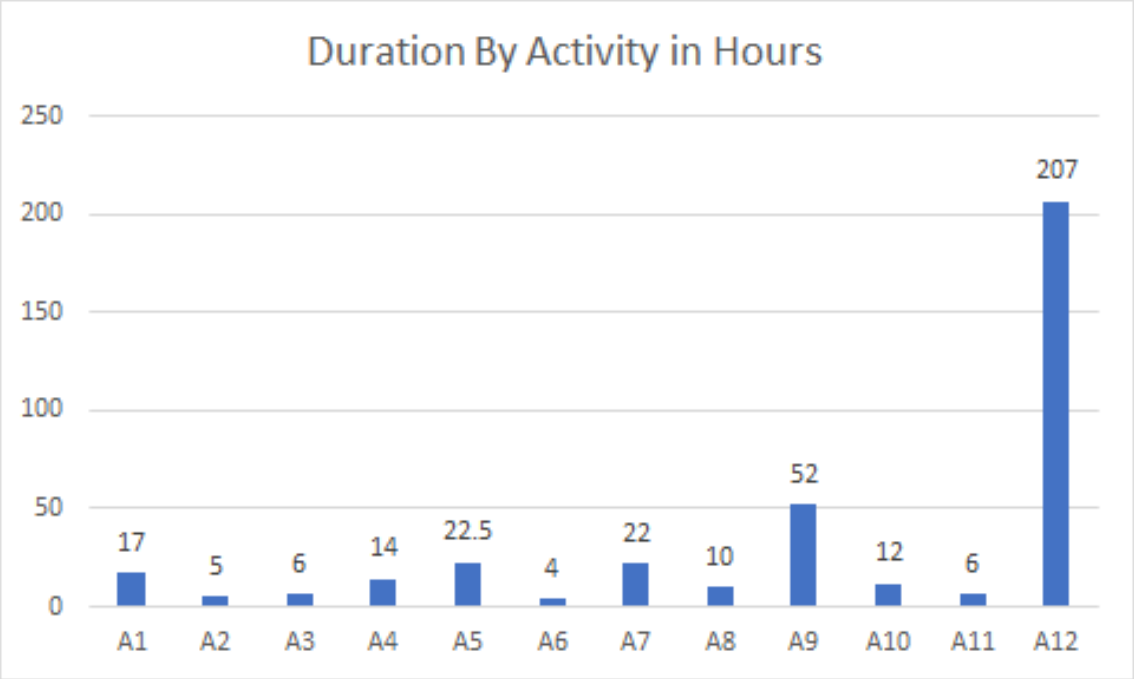

Figure 4.3: Duration of every activity in hours

In Figure 4.3, it is clear that the transformation was the most time-consuming (A12). Possible changes to A12's granularity are discussed in Section 5.1.4.

The following tables show what activities that were involved in the actual SPL integration. Activities that resulted with 0 in each field have been omitted.

Activity ID	LOC added	LOC removed	LOC modified
A5	0	0	3 365
A10	0	11 670	0
A12	17874	1492	N/A

Table 4.6: All activities and LOC added/modified/removed

As seen by Tables 4.6 and 4.7, most of the performed activities were not relevant for the actual SPL integration. It is actually only activity A12 that adds to the SPL. A5 and A10 changed the SPL LOCs but did not contribute to the migration of any feature, only the removal of dead code and translation of comments.

Activity ID	Files added	Files removed	Files modified
A5	0	0	128
A10	0	78	133
A12	184	80	4

Table 4.7: All activities and files added/modified/removed

Results seen in Tables 4.6 and 4.7 shows clearly that most of the activities during a reengineering process does not affect the code. This means that other areas has to be considered when measuring effort, see Section 5.1.5 for further discussion.

4.5 Thesis Comparison

This section lists activities from both the collaborating Master's Theses. Activities are sorted according to activity types and as aforementioned, only activities that are of the same type is compared. If an activity has more than one type, hours for that activity are divided by amount of types, for example: *two types => type x = activity hours/2 and type y = activity hours/2.* Table 4.8 compares amount of hours spent on every activity type.

Activity types	Annotative approach	Compositional approach
SPLE Training	90	16
Data cleansing	0	23,25
Domain analysis	82	18
Feature identification	22	22,25
Diffing	40	26
Architecture identification	5	2
Feature location	7	50
Feature modeling	10	7
Transformation	210	103,5
Quality assurance	30	103,5
Total amount of person hours:	**496**	**371,5**

Table 4.8: Comparison of total person hours per activity type

The data from Table 4.8 show that some activities have similar amount of person hours. The big differences can be seen for *SPLE Training* which might be because of different entry knowledge between the two groups that is the result of previously taken courses within the SPL field. Other larger differences, such as for *Transformation*, *Quality assurance* and *Domain analysis* can be seen as well. The comparison of hours is supplemented by Table 4.9 that shows what activities have been categorized with activity types. This data helps to comprehend the amount of person hours since relying on only the Table 4.8 can be misleading.

Activity types	Annotative approach	Compositional approach
SPLE Training	- Installing and learning FeatureIDE - Code diffing - General SPLE learning	- Reading about compositional SPL - FeatureHouse research - FeatureIDE research
Data cleansing		- Translating code comments - Removing unused code
Domain analysis	- Running and testing the games - Architecture analysis - Manually analysing code - Building and running games in Android Studio - Feature identification	- Running games - Mapping game features - Creating feature model
Feature identification	- Running and testing the games - Feature identification	- Running games - Mapping game features - Translating code comments
Diffing	- Code diffing	- Pairwise comparison of variants
Architecture identification	- Architecture analysis	- Reverse engineering of class diagrams
Feature location	- Feature identification	- Reverse engineering of class diagrams - Finding features in source code - Pairwise comparison of variants
Feature modeling	- Installing and learning FeatureIDE - Transformation: ApoSnake and ApoDice - Transformation: ApoSnake, ApoDice and ApoClock - Transformation: ApoSnake, ApoDice, ApoClock and ApoMono. - Transformation: All games.	- Creating a feature model
Transformation	- Transformation: ApoSnake and ApoDice - Transformation: ApoSnake, ApoDice and ApoClock - Transformation: ApoSnake, ApoDice, ApoClock and ApoMono - Branding: Font - Transformation: All games - Branding: Menu- Annotating features - Quality assurance	- Transforming source code to feature
Quality assurance	- Quality assurance	-Transforming source code to feature

Table 4.9: Comparison of performed activities per activity type

To grasp a more accurate picture of the two migration processes, one can examine the percentage of duration that each activity type takes in the entire migration process. Looking at the red bars in Figure 4.4 making up the compositional migration process, it is clear that testing the SPL (Quality Assurance) and transforming it (Transformation) occurred in parallel and together cover over 50% of the migration process. Another considerable activity type is Feature Location with 15.5%, with diffing also constituting 7%.

Looking at the blue bars in Figure 4.4, representing the annotative approach, Quality Assurance activities occurs only at the end of the migration process. Naturally, Transformation activities are making up the largest section of the migration process. However, with the annotative approach, we see a larger focus on the domain analysis and a smaller focus on feature location with only 1.4% as opposing to 13.5% to the compositional counter part.

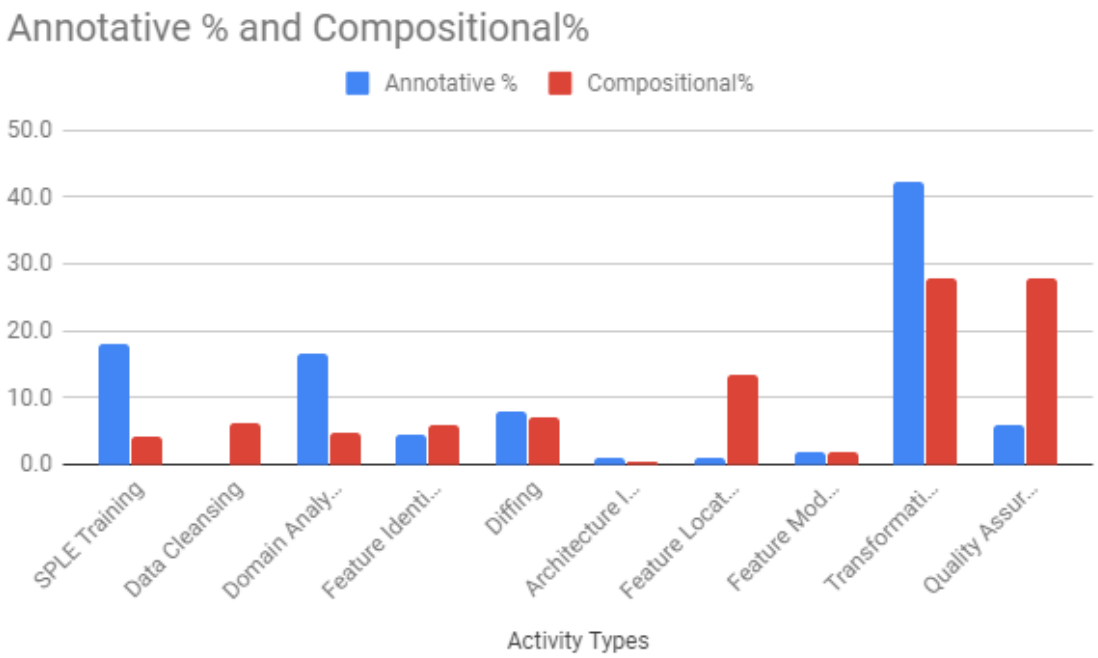

Figure 4.4: Comparison of percentage of each activity type in both migration process approaches

5
Discussion

The chapter discusses the results acquired from this study in relation to this thesis' research questions. It also presents different challenges that were discovered during this study.

5.1 Discussion

This study provided many interesting results but despite all empirical data on what it actually is that you do during a reengineering process, things could have been done differently. The following sections bring up different challenges and other discoveries that can produce even better results for a similar study.

5.1.1 Level of Completion

During the migration process, it was not possible to migrate all the five variants into one SPL, this was due to several challenges that were encountered (see Section 5.1.8). The three games that were migrated into the SPL have different degrees of configurability. The migration process began with reengineering ApoNotSoSimple (V2) and ApoSnake (v4) since the perception after running all games combined with results from the pairwise diffing showed that these two variants shared many commonalities. The last variant (ApoIcarus) that was migrated was not divided into as many features, however, the purpose of its inclusion is to demonstrate that it is possible to include games of different genre into the SPL (ApoSnake and ApoNotSoSimple are level-based games, while ApoIcarus is an endless-runner game). Ideally, the reengineering should extent to satisfy the feature model, meaning every feature is populated with its own respective code block. However, this is not always achievable for several reasons.

First, given the uniqueness of the code and the high degree of coupling between features within variants, it became quickly obvious that reengineering of all the variants to satisfy the feature model is too ambitious and not possible for the given time frame of the project. This is because each game, while architecturally similar, varies greatly in the game-play features such as enemies, world, game rules and so forth. Hence, even if time was spent satisfying every feature in the feature model, this would not increase the SPL's configurability. For instance, adding the ApoNotSoSimple enemy (an enemy which shifts all elements in the game world every time the player move), will not function in ApoSnake since the player and the levels will

not be winnable or playable even if one would have spent the time necessary to decouple these features. In other words, the inclusion of a game rule from variant A to variant B, usually breaks all the rules of variant B, which renders the game unplayable.

To summarize, the focus in this migration process was to showcase useful configurability where the gameplay remains functional. For example, a client may find it useful to be able to generate variants with and without the Game Editor (where the user can create their own level) portion of the game depending on the customer requirements.

5.1.2 RQ.1 Pros and Cons of Different Strategies

Examining the literature and studying the different strategies was a time consuming task given the lack of concrete standardized, or at least agreed-upon, strategy or approach to migrate an existing system to a software product line. While there are several authors that attempted to present their experiences in such reengineering, their papers either rely on different resources other than ours (for instance on commit history rather than source code), or that depend on tools that are not available. Additionally, some papers also report results of successful migrations but they do not provide any repositories with their developed SPL nor do they provide sources to the existing systems or any concrete list of activities on how they achieved it.

To summarize, we find the literature on SPL migration strategies to be fragmented and most if not all papers provide a high-level or a brief summary of the authors migration process. For this reason, we conclude that while strategies do have some advantages and disadvantages, the more important factor is that the selection of the most appropriate strategy, may not depend on pros and cons of the said strategy, but on three factors which are described below.

5.1.2.1 Data Available

When it comes to transforming an existing software, the data available varies greatly. And according to studies, in most industrial cases, the developers may not even be able to run the software at all[19]. Since most of the approaches to migration take data as an input to produce an output. Therefore, the availability of the data will filter out several approach and limits the developer to only few applicable strategies. However, one can produce more relevant data from already available data. For example in this study, we were able to reverse engineer class diagrams using existing tools with high accuracy which increases the number of applicable strategies. Although reverse engineering other high level documentation such as domain model may have reduced accuracy comparing to class diagrams.

5.1.2.2 Resources

Another factor that is a driver in choosing the appropriate strategy, is the resources available. Reengineering existing software into a compositional product line is ex-

tremely time consuming. One must estimate the time necessary to complete the migration before starting, especially since the compositional approach is slower and riskier than the annotative one [16].

Additionally, ideally, the team responsible for the migration should have at least one domain expert if they need to carry any Expert-driven strategy. Moreover, the team also need to allocate resources into SPL training and getting familiar with the necessary tools such as FeatureIDE.

5.1.2.3 Tools

Tools available is also a decisive factor in choosing the strategy. Several approaches, especially dynamic analysis approaches requires tools for feature location which aren't always readily available. Most of the tools were either outdated, commercial or non-existing (discontinued). Moreover, several tools requires training to use, for example search-based usually entails feeding data to statistical software such as R studio or requires the need to write searching algorithms that optimized the search for commonalities and variablities [36]. Finally, tools might provide unhelpful output, for example in our case But4Reuse.

5.1.3 RQ.2 Migration Effort Measurement

The designed template have different factors that are similar to some cost models. These factors help when trying to illustrate how much effort a certain activity takes. Even though it is about effort and not costs in terms of monetary values, one will get an estimation of what it might cost especially when comparing activities to each other. But despite how accurate cost models might be and the mapping of those metrics into our measurement design, the estimation might not correspond to the amount of effort invested.

Discussions in Section 5.1.5 bring up difficulties with non-coding activities. This is coming from missing measurements in the measurement design, which leads to questions about how to show effort for these kinds of activities other than the amount of hours spent. Another problem turned out to be edited LOC, where no tool to identify if a line had been modified was found. Going through thousands of LOC in git commits to identify what lines were edited instead of added/removed was not possible. This led to missing data for the important activity A12.

5.1.4 RQ.3 Activities in a Compositional Reegineering

Previous literature describes the migration process in a waterfall fashion, as one would start with the detection phase, followed by the analysis phase and finally the transformation phase [3]. In our research, we find that in the beginning of the reengineering process, the first two phases happen in parallel until enough information about source code and features is gathered. In other words, we go back and forth between detection and analysis phase. Once enough data is gathered, the transformation phase can begin, but even in this phase, we always did revisit the

two first phases. This is because as you transform certain features, one may discover other features or wants to separate a feature into two or combine two existing ones. For this reason, we say that developers will most likely go back and forth to different activities during the reengineering process.

Early in the process, activity granularity was discussed. After defining activities, it becomes clear that, throughout the reengineering process, the granularity is too vague and it could be even more fine-grained. An activity should be language independent and domain independent. It should also describe a performed activity more detailed than what previously been determined. This is so it can be easily transferable to other studies, as well as tested on migrations on different software systems. The retrieved list of activities does not give an accurate picture of the final phase, the transformation phase. Even though the granularity is at a similar level throughout the process, one does not really see what has been going on during the transformation. Activity A12 ends up being too holistic and lacks details, it should have been broken down into several smaller activities. If this would have been done, the transformation phase would have been easier to grasp, which is what previous literature have been missing as mentioned earlier. A list with more detailed activities from this study regarding transformation could consist of things such as:

- Make hard-coded parts dynamic
- Change data structures to allow for greater modularity
- Refactor similar methods into a generic method
- Unify duplicated code into general code

This list shows that these activities are out of scope in regards of the defined activity granularity, where we assumed the reader understands that these things happen during reengineering. But a more detailed activity definition would have provided a better overview of the process.

5.1.5 RQ.4 Different Efforts of Activities

Even by basing effort metrics on well-established cost models, we found it challenging to track the migration process. This is because one might switch between activities in a matter of seconds. For instance, pairwise comparison, feature location and transformation often occur at the same time. However, it is clear that transformation is by far the most time-consuming activity. The results also show that many performed activities have empty, or non available data, in the log remaining after mapping relevant efforts. This means that effort measuring can be even more difficult for non-coding activities.

Commit history helps quantifying efforts of the migration process by keeping track of lines of code migrated, files added and features implemented. However, tracking lines of code modified from the original variants is challenging as migration to SPL necessitate the developer to start a brand new SPL project, hence commit history does not provide changes between the variants and the SPL. Moreover, even with frequent commits with descriptive commit messages, we find it challenging to determine when a feature is fully migrated. This is because as the developer continue to

migrate the rest of the system, they may end up changing previous features (breaking down a feature or merging two features) and hence source code from previously migrated features need to be changed.

Furthermore, we identify additional factors that we learned (after the migration process) to be of significance when it comes to measuring efforts of the migration process. The quality of the code is a major factor in estimating the efforts. For instance, if the original variants does not contain any comments, the migration duration will increase as the developer need to spend additional resources on software comprehension. Additionally, we believe that studying internal software quality such as cyclomatic complexity and the amount of hard coded source code can be an indicator on how long it will take the transform the variants into an SPL.

5.1.6 Top-Down vs. Bottom-up

Code belonging to a feature might be scattered or tangled. This turned out to be problematic in different ways after trying both the top-down and bottom-up approaches during this study. A top-down approach gave a less accurate SPL result than a bottom-up, but the bottom-up approach ended with more bad features (in our case, feature that only works for a specific variant) compared to top-down. A good feature usually provide a distinct functionality to an SPL [54], where bottom-up could have features such as *Utils* compared to top-down that described functionality in a better way, e.g. *Player*. Because of this, a top-down approach gives a better overall understanding compared to bottom-up. But it takes a longer time to achieve the same level of configurability as for if a bottom-up approach would be used.

5.1.7 Thesis comparison

This section discusses the comparison of results found in the collaborating theses. As aforementioned, the other thesis uses an annotative approach as oppose to our compositional approach, and with a different dataset based on Android, yet the same domain.

Many of the big differences in hours spent per activity type between the two theses is most likely caused by the previously mentioned dividing of hours by amount of types per activity. An example that reflect this misleading result is hours for *Transformation* and *Quality assurance* in the Compositional approach. There is only one activity that is categorized with these two which means the hours are split in half over the two activity types. This misleading data also shows what have been discussed earlier: activities needs to be more fine-grained. That would separate activities between types more clearly, which in the end delivers a better result.

However, looking at the percentage of each activity type occupying both SPL migration approaches in Figure 4.4, we notice that all activity types represents similar sizes in terms of percentage in both migration approaches, with the exception of activities relating to Feature Location which are a much bigger part of the composi-

tional approach than the annotative one. This can be because in the compositional approach, there is a large focus on separating code blocks in each feature in the feature model while in the annotative one it suffices to annotate each feature using *IFDEF* techniques. In the same figure, we observe a difference between Transformation activity types and Quality Assurance, however, this can be misleading as previously described and because of different logging processes between the two theses. SPLE Training activities also differ considerably between the two migration processes, however, this is insignificant since the duration of these activities are solely dependent on the developers' previous experiences in SPLE.

The last difference is that the annotative approach spent more efforts in Domain Analysis as opposing to the compositional approach. As domain analysis and domain knowledge is important in any SPL migration process, we suspect that this is the result of poor or misleading logging of the other thesis when it comes to Domain analysis activities.

5.1.8 Challenges

Many of the concepts within SPLE are good in theory. When it comes to applying them in practice, and especially to a reengineering process, several challenges were faced that hinder the process. This section summarizes obstacles that were discovered during this Masters Thesis.

Challenge 1 - Features can be technical or domain-level: While the feature model is seen as a communication tool between client and supplier, it may be beneficial to add technical features in the feature model that may not mean anything to the client such as "input", "engine" and such. This is to increase the configurability, hence increase the number of possible variants to be generated from the SPL.

Challenge 2 - Amount of Resources to Allocate to Refactoring: We learned from the migration process that it is important to somehow quantify how long the reengineering of each feature is. From this, one can define how many features a developer can afford to migrate from the beginning of the project. From this information, the developer will choose to break down the existing system in a defined number of features in order to finish the project in time and within budget. In our case, we broke down the system in a way that we believed would be optimal to the client and end-user. In other words, the feature model contained enough configurability to satisfy any possible requirements given by the two types of stakeholders.

In addition, we find it equally important to identify feature dependencies prior the transformation. This step requires an extensive analysis of the source code. This is important because the number of dependencies of one feature will heavily impact the effort required to transform the feature. Ideally, one would measure and combine the size of the feature and its number of dependencies to control the scope of the migration process. Moreover, another metric that is important to quantify is the degree of entanglement and scattering a feature has. All of these measurements

combined can help the developer understand the effort needed to transform a feature, and perhaps even which feature the developer should migrate first.

Finally, another factor that is identified as important, is to study and compare the dependencies between different code artifacts (more specifically classes) and the dependencies between features. We find it easy to assume that these two dependencies are similar, if not identical. However, technically, this is not the case at source code level. For instance, in the beginning of the feature identification phase, we did not consider or assume dependencies between Actor and Menu Button until the source code was systematically read.
Challenge 3 - Quality of the Existing Source Code: Code clean-up extends beyond finding dead code and duplication, the developer does not always use descriptive naming and sometimes an artifact (e.g. variable or class name) with same purpose can be named differently in different variants. Also, some pieces of code can belong to two features, for instance, *isVisible()*,a function in all variants source code, is used to toggle the visibility of both buttons (UI elements) and Enemies, hence it will belong to two features (enemy and UI element). This is because source code dependencies are not the same as feature dependencies.

Code quality can be inconsistent and poor, for instance, most buttons in the game are hardcoded into an array buttons. Since each button can represent a feature (for example, Game Editor feature has a button in the Menu, and so does Options and Credits and so forth), one needs to change the data structure into a more dynamic one such as a HashMap to allow configurability (enabling and disabling) of these buttons without resulting into any kind of exception.

Additionally, other buttons are not even included in the array, instead they are scattered in different classes. This was discovered when migrating other features and required us to go back to fix already-migrated features. Moreover, not all buttons uses an image as asset, some are drawn using Java libraries, which makes the initialization of buttons different from one another. This further complicated the migration process and this can be avoided if all the buttons followed a consistent design choice.

Challenge 4 - Feature dependencies: Features depend on other features. In order to extract one feature you have to use a lot of code that is not part of that feature, which means it is difficult to test if a feature is fully functioning by itself. We find the identification of feature dependencies extends beyond domain knowledge, and one must also analyze what features depend on the source code of the feature that is being migrated.

Moreover, even though variants have been derived from the clone-and-own methodology, there is a high possibility that they are very different, especially in the way it connect to the rest of the source code. This leads to a lot of effort to reengineer and make “general” features. The few things that are commonalities could have many dependencies to variabilities (related to feature dependencies challenge and technical/domain challenge). E.g. “enemy-feature”.

Challenge 5 - Superimposition becomes complex for larger projects: Superimposition is great in theory and also works according to plan. When software grows and the amount of features gets larger as well, same goes for superimposition that might expand over new features. When the same class exists in several features, this means that a large class (from the original source code) that belongs to many features, say n number of features results in the division on the said class to n number of classes with identical name. The presence of several classes with identical names can get confusing and the maintenance of these classes becomes a complex task..

Challenge 6 - Refactor complications: During refactoring different data structures may be modified. This leads to redundant workload when different features need the same refactoring. E.g. array[] to ArrayList<>.

Challenge 7 - Feature model constraints: When trying to implement new features one continuously recognizes new constraints for the feature model. It is difficult to add these to the model during implementation since if the constraint is there, the feature and other features that are involved in the constraint must be fully functioning. This complicates the implementation process and possibility of missing constraints. It raises the question as to when is the optimal time to add the constraint to the feature model.

Challenge 8 - Difficulties to track evolution: Because of scattered and tangled features, commit history will never tell the real story of what you actually did in that commit. It might be that feature X and Y is edited in order to make feature Z working.

Challenge 9 - Test-driven development: There might not exist a possibility to test an implemented feature even if the feature has been successfully integrated. The software might not be possible to build and test, despite a feature that is working. This relates to the challenge of feature dependency, since the migrated feature may depend on other feature and hence, even after successful migration, once can not test the feature.

Challenge 10 - Involvement of Stakeholders: During the migration process, we needed to make important decisions with low confidence given the absence of important stakeholders such as the original developer, clients and end-users. Most questions evolve around features and data available from the start (in our case the source code). There were decisions such as:

- Is characteristic X of the system important enough to be a feature?
- Is the difference between two code blocks caused by evolution or is it feature specific?
- Should this feature be broken down into more features to increase configurability?

These types of questions are important because the migration process can become more time-consuming depending on the answer. For instance, if feature *a*, *b* and *c* are not important to the client, the developer can save time by not separating the concerning code blocks. Hence, to increase the quality and efficiency of a migration, several stakeholders must be involved.

Challenge 11 - Poor Readability: The very nature of compositional SPL reengineering is to break down the existing software system's source code into code units where each unit represent a certain feature. In practice, this results most of the time in every feature sharing classes with the same names, that will later be superimposed when a variant is generated. For example, a feature *Option*, responsible for letting the user access and change different options in the game (e.g. turn music on and off), has a GUI part, logic part, buttons and rendering methods. In order to implement this feature in an SPL, each part of the system (GUI, logic) will be implemented in the Java class that is also shared by dozens of other features. For instance, *ApoGamePanel.java* is the class that contains most of the GUI elements and handles the button listeners and more. Hence, different feature folders will have files named *ApoGamePanel.java*, each having a code fragment that defines a specific feature. In fact, the SPL project contained 15 classes with the name *ApoGamePanel* after the last commit. The challenge is illustrated in Figure 5.1.

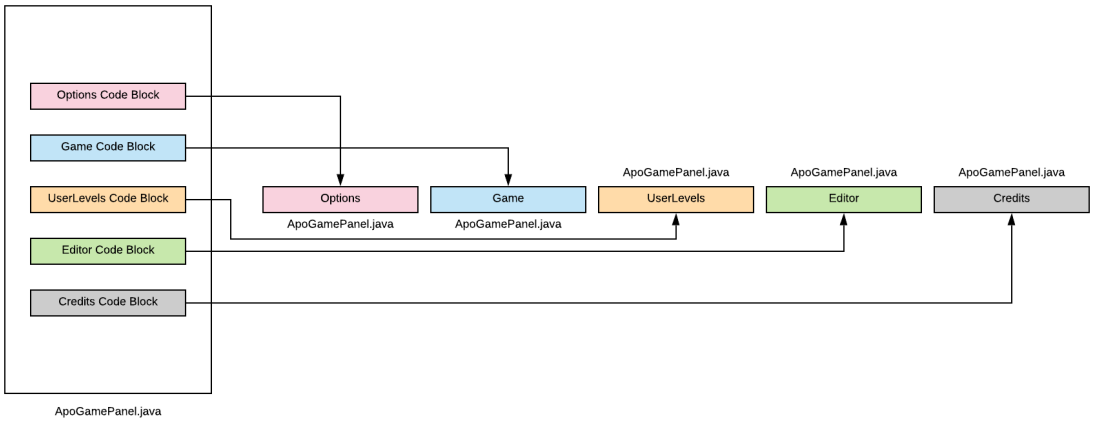

Figure 5.1: Illustration of the Result of Distributing Code Blocks Between Features

When the system contains a lot features, the developer can lose track of how many files in total that makes up the entire class. One possible solution is to look at the generated code of a specific variant once all the code is superimposed in one file, but it is not always possible to choose all features that represent a class since that configuration might be invalid (because of feature constraints). However, even readability there is difficult as each fragment is implemented as its own method. This means that if a function, such as *init()*, contains code blocks from 10 features, the function *init()* will be scattered in 10 different functions in the generated class. This can be seen in Figure 5.2.

```
 private void  init__wrappee__WhiteMenu  () {
    super.init();
    super.setShowFPS(false);

    if (this.buttons == null) {
        this.buttons = new ApoGameButtons(this);
        this.buttons.init();
    }
    if (this.menu == null) {
        this.menu = new ApoGameMenu(this);
    }
    this.setMenu();
}

 private void  init__wrappee__World  () {
    init__wrappee__WhiteMenu();
    if (this.game == null) {
        this.game = new ApoGameGame(this);
    }
}

 private void  init__wrappee__UserLevels  () {
    init__wrappee__World();
    this.game.getUserLevels().loadHighscore();
}

 private void  init__wrappee__LevelChooser  () {
    init__wrappee__UserLevels();
    if (this.levelChooser == null) {
        this.levelChooser = new ApoGameLevelChooser(this);
    }
        if (ApoGameConstants.B_APPLET) {
            String load;
            try {
                load = ApoHelp.loadData(new URL(ApoGameConstants.PROGRAM_URL), ApoGameConstants.COOKIE_NAME);
                int level = Integer.valueOf(load);
                this.levelChooser.setMaxLevel(level);
            } catch (MalformedURLException e) {
                e.printStackTrace();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
}
```

Figure 5.2: Example of Poor Readability in the Generated Java Files

6
Conclusion

The chapter present the final conclusion of the research and also recognizes threats to validity and what can be done as future work.

6.1 Migration

The reengineering process for a compositional SPL was more time consuming than anticipated. In some situations where an annotative approach can be fairly simple, the compositional approach on the other hand needs plenty of overhead and takes longer to implement [16]. It might be more efficient to start from scratch with SPL if a system is very large, instead of trying to reengineer the current system. Another solution could be to use both approaches, where annotative is used in the beginning since it usually goes faster and later cross over to a compositional approach. It is also possible to improve superimposition by implementing annotation within the compositional approach, as suggested by [16]. But all this is of course time consuming and will further increase required efforts.

No matter what was tried during this study, we ended up with some bad features. It could be because of the domain (games) that even though some variants might seem to have many commonalities, they have extreme variability on source code level. This leads to that much of the code base ends up in variant-specific features. This decreases the overall configurability of the SPL.

Throughout the process, difficulties were encountered when trying to migrate one feature at a time. It might be that the migration process can work more fluently, with continuous working builds, if one applies "reverse featuring". With this approach, one complete variant is implemented in one feature. Afterwards, one feature at a time is getting extracted from the complete variant into a separate feature. This means you keep having a working game every time a new feature has been extracted.

6.2 Threats to Validity

This section brings up recognized internal threats and external threats to validity. It also presents how the study tries to mitigate these threats.

6.2.1 Internal Validity

This study decides on a migration strategy after a literature review about existing strategies and their differences. There are limitations when choosing a specific strategy, activities involved can differ depending on what strategy you choose. The domain can matter as well, which means that the answers for RQ.2 might not be applicable in general.

The qualitative data with activities involved has a risk of being incomplete, where something that should be done is missed. Another internal threat is the consistency and accuracy of the logging. To try and avoid these threats we thoroughly document every step of our migration process and will conduct weekly meetings to discuss the logging and its accuracy, and amend any inaccuracy found. In addition, to reduce the risk to collect subjective data and to perform subjective tasks, all our measurement and tasks are be based on processes that has been previously used in past research of reengineering software systems into product lines. There is a possibility that the measuring of time spent for activities are wrong. It can be that there are inconsistencies when it comes to logging hours spent.

Another threat regarding activities is that it can be difficult to keep track of what activities that has been performed. This is since activities can sometimes be performed in parallel to each other. These threats are mitigated by continuously adding hours to reduce mistakes if numbers were to be added at later stages. If any number is wrong they most likely present less hours than what has really been spent on that specific activity.

Moreover, there is always a risk that we can misunderstand features of the software. To mitigate this, all the features are discussed with the supervisor. Another risk is that the research is be conducted by students and not industry practitioners, hence the applicability of our findings in an industrial context may be reduced. To mitigate this risk, our measurements and reengineering strategy are be based on well-established literature and common practices of SPL engineering. This means that if an industry practitioner would perform a reengineering process, they would apply the same methodology as in this case study.

6.2.2 External Validity

The dataset studied in this research is a collection of Java games with an average of 6000 LOC per product. This can limit the applicability of our results (migration strategy and efforts) to this domain and size of the software. The limitation is reduced since the games have typical layers that are shared with other programs, such as user interface and application logic such as game rules and game mechanics. But in terms of scalability, we cannot generalize beyond the size of the software.

6.3 Future Work

This study could be produced again with the same dataset and strategy to try and identify further reengineering activities, by applying the previously discussed revised definition of activity granularity. A replication could also finish migrating all five variants in order to have a more vigorous quantitative data on efforts needed, since this study only implemented three out of five variants before time ran out.

We encountered several difficulties during this study. These are described to pinpoint what different thresholds that exists for an easy migration process. Researchers could further investigate the challenges this study mentions. By doing this, there is a possibility to evolve SPLE so that it becomes less complex in practice. This could in the end be of high value within the industry where SPLs could be the next step, and companies become more confident in migrating their software systems using the compositional SPL approach.

Bibliography

[1] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*, 2013. [Online]. Available: http://link.springer.com/10.1007/978-3-642-37521-7

[2] S. Apel, C. Kastner, and J. Liebig. Featurehouse: Language-independent, automated software composition. Accessed: 2019-02-26. [Online]. Available: http://www.fosd.de/featurehouse

[3] W. Assunção, R. Lopez-Herrejon, L. Linsbauer, S. Vergilio, and A. Egyed, "Reengineering legacy applications into software product lines: a systematic mapping," *Empirical Software Engineering*, vol. 22, no. 6, dec 2017. [Online]. Available: https://www.engineeringvillage.com/share/document.url?mid=inspec{_}3e5bef5d161be2c280dM5ec41017816339{&}database=ins

[4] J. Bosch, "Software product line engineering," *Systems and Software Variability Management: Concepts, Tools and Experiences*, pp. 3–24, 2013.

[5] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Bottom-up adoption of software product lines: A generic and extensible approach," in *Proceedings of the 19th International Conference on Software Product Line*, ser. SPLC '15. New York, NY, USA: ACM, 2015, pp. 101–110. [Online]. Available: http://doi.acm.org/10.1145/2791060.2791086

[6] M. Lillack, W. Hedman, and T. Berger, "Intention-Based Integration of Software Variants." [Online]. Available: http://www.cse.chalmers.se/~bergert/paper/2019-icse-incline.pdf

[7] J. Krüger, W. Fenske, T. Thüm, D. Aporius, G. Saake, and T. Leich, "Apo-games: a case study for reverse engineering variability from cloned Java variants," *Proceeedings of the 22nd International Systems and Software Product Line Conference - Volume 1, {SPLC} 2018, Gothenburg, Sweden, September 10-14, 2018*, pp. 251–256, 2018. [Online]. Available: https://doi.org/10.1145/3233027.3236403

[8] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, ser. SEI series in software engineering. Addison-Wesley, 2002.

[9] F. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action*, 2007, vol. 41, no. 6193.

[10] M. Asadi, E. Bagheri, B. Mohabbati, and D. Gašević, "Requirements engineering in feature oriented software product lines," vol. II, p. 36, 2012.

[11] A. Classen, P. Heymans, and P. Y. Schobbens, "What's in a feature: A requirements engineering perspective," 03 2008, pp. 16–30.

[12] W. Fenske, J. Meinicke, S. Schulze, S. Schulze, and G. Saake, "Variant-preserving refactorings for migrating cloned products to a product

line," Piscataway, NJ, USA, 2017//, pp. 316 – 26, code identification;clone detection;feature-oriented SPL;semiautomated process;step-wise process;commonality extraction;commonality identification;SPLs;software product lines;product variants;bug fixes;evolution;maintenance;clone-and-own;code reusing;cloned product migration;variant-preserving refactorings;. [Online]. Available: http://dx.doi.org/10.1109/SANER.2017.7884632

[13] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Measuring clone based reengineering opportunities," in *Proceedings Sixth International Software Metrics Symposium (Cat. No.PR00403)*. IEEE Comput. Soc., Los Alamitos, CA, USA, 1999, pp. 292–303. [Online]. Available: https://www.engineeringvillage.com/share/document.url?mid=inspec{_}base906441778{&}database=inshttp://ieeexplore.ieee.org/document/809750/

[14] G. Zhang, L. Shen, X. Peng, Z. Xing, and W. Zhao, "Incremental and iterative reengineering towards software product line: An industrial case study," in *IEEE International Conference on Software Maintenance, ICSM*. IEEE, Piscataway, NJ, USA, 2011, pp. 418–427. [Online]. Available: https://www.engineeringvillage.com/share/document.url?mid=inspec{_}10655dd1345c62be6fM621b2061377553{&}database=ins

[15] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," 05 2008.

[16] C. Kästner and S. Apel, "Integrating compositional and annotative approaches for product line engineering," 10 2008.

[17] S. Apel and C. Kastner, "Language-independent and automated software composition: The featurehouse experience," *Software Engineering, IEEE Transactions on*, vol. 39, 01 2011.

[18] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 311–320. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368131

[19] V. Anwikar, R. Naik, A. Contractor, and H. Makkapati, "Domain-driven technique for functionality identification in source code," *ACM Sigsoft Software Engineering Notes*, pp. 1–8, 05 2012.

[20] Y. Tang and H. Leung, "Top-down feature mining framework for software product line," vol. 2, 01 2015, pp. 71–81.

[21] J. Rubin, K. Czarnecki, and M. Chechik, "Managing Cloned Variants: A Framework and Experience," *Proceedings of the 17th International Software Product Line Conference - SPLC '13*, p. 101, 2013. [Online]. Available: http://doi.acm.org/10.1145/2491627.2491644{%}5Cnhttp://dl.acm.org/citation.cfm?doid=2491627.2491644{%}5Cnhttp://dl.acm.org/citation.cfm?doid=2491627.2491644

[22] O. LLC. The objectaid uml explorer. Accessed: 2019-02-03. [Online]. Available: https://www.objectaid.com/home

[23] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, Sep. 2009.

[24] T. Eisenbarth, R. Koschke, and D. Simon, "Derivation of feature component maps by means of concept analysis," in *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, March 2001, pp. 176–179.

[25] P. Frenzel, R. Koschke, A. P. J. Breu, and K. Angstmann, "Extending the reflexion method for consolidating software variants into product lines," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, Oct 2007, pp. 160–169.

[26] D. Romero, S. Urli, C. Quinton, M. Blay-Fornarino, P. Collet, L. Duchien, and S. Mosser, "SPLEMMA: A Generic Framework for Controlled-Evolution of Software Product Lines," *Proceedings of the 17th International Software Product Line Conference co-located workshops (SPLC)*, vol. 2013, p. 59, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2499777.2500709

[27] H. Prahofer, F. Angerer, R. Ramler, H. Lacheiner, and F. Grillenberger, "Opportunities and challenges of static code analysis of iec 61131-3 programs," 09 2012, pp. 1–8.

[28] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic Notes in Theoretical Computer Science*, vol. 217, pp. 5 – 21, 2008, proceedings of the 3rd International Workshop on Systems Software Verication (SSV 2008). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066108003824

[29] Z. Zhioua, S. Short, and Y. Roudier, "Static code analysis for software security verification: Problems and approaches," in *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, July 2014, pp. 102–109.

[30] L. Hu and K. Sartipi, "Dynamic analysis and design pattern detection in java programs," Skokie, IL, USA, 2008//, pp. 842 – 6.

[31] H. Safyallah and K. Sartipi, "Dynamic analysis of software systems using execution pattern mining," vol. 2006, 07 2006, pp. 84– 88.

[32] A. Santos, F. Nunes Gaia, E. Figueiredo, P. Neto, and J. Araújo, "Test-based spl extraction: An exploratory study," 03 2013.

[33] H. Koziolek, T. Goldschmidt, T. de Gooijer, D. Domis, and S. Sehestedt, "Experiences from identifying software reuse opportunities by domain analysis," 08 2013, pp. 208–217.

[34] U. Ryssel, J. Ploennigs, and K. Kabitzsch, "Extraction of feature models from formal contexts," in *ACM International Conference Proceeding Series*, 2011.

[35] M. Eaddy, A. Aho, G. Antoniol, and Y.-G. Guéhéneuc, "Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," 07 2008, pp. 53–62.

[36] R. E. Lopez-Herrejon, L. Linsbauer, and A. Egyed, "A systematic mapping study of search-based software engineering for software product lines," *Information and Software Technology*, vol. 61, pp. 33–51, 2015.

[37] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques.* Berlin, Heidelberg: Springer-Verlag, 2005.

[38] K. Pohl and A. Metzger, *Software Product Lines*, 06 2018, pp. 185–201.

[39] P. Clements, J. McGregor, and S. Cohen, "The structured intuitive model for product line economics (SIMPLE)," *Technical Report CMU/SEI-2005-TR-003*, no. February, 2005. [Online]. Available: http://oai.dtic.mil/oai/oai?verb=getRecord{&}metadataPrefix=html{&}identifier=ADA441881

[40] B. Boehm, A. Brown, R. Madachy, and Ye Yang, "A software product line life cycle cost estimation model," *2004 International Symposium on Empirical Software Engineering*, 2004. [Online]. Available: https://www.engineeringvillage.com/share/document.url?mid=inspec{_}480457101c9d09f2eM69b919255120119{&}database=ins

[41] J. Nobrega, E. Santana de Almeida, and S. Meira, "Income: Integrated cost model for product line engineering," vol. 0, 09 2008, pp. 27–34.

[42] I. Sommerville, "28. Software Re-engineering," pp. 1–18, 2000. [Online]. Available: https://ifs.host.cs.st-andrews.ac.uk/Resources/Notes/Evolution/SWReeng.pdf

[43] A. Dreiling and C. Kästner. Variability mining with leadt. Accessed: 2019-02-10. [Online]. Available: https://github.com/ckaestne/LEADT

[44] C. Kästner, S. Apel, M. Rosenthal, and A. Dreiling. Cide: Virtual separation of concerns (preprocessor 2.0). Accessed: 2019-04-17. [Online]. Available: http://ckaestne.github.io/CIDE/#intro

[45] B. S. Inc. Biglever software gears product line engineering tool lifecycle framework. Accessed: 2019-02-19. [Online]. Available: https://biglever.com/solution/gears/

[46] S. U. of Luxembourg. But4reuse. Accessed: 2019-04-17. [Online]. Available: https://but4reuse.github.io/

[47] Y. Xue, "Reengineering legacy software products into software product line based on automatic variability analysis," Piscataway, NJ, USA, 2011//, pp. 1114 – 17. [Online]. Available: http://dx.doi.org/10.1145/1985793.1986009

[48] fournova Software GmbH. Version control best practices. Accessed: 2019-02-08. [Online]. Available: https://www.git-tower.com/learn/git/ebook/en/command-line/appendix/best-practices

[49] Angular. Developing angularjs, git commit guidelines. Accessed: 2019-02-08. [Online]. Available: https://github.com/angular/angular.js/blob/master/DEVELOPERS.md#-git-commit-guidelines

[50] M. GmbH. Featureide. Accessed: 2019-03-15. [Online]. Available: http://www.featureide.com/

[51] V. Paradigm. Visual paradigm. Accessed: 2019-03-15. [Online]. Available: https://www.visual-paradigm.com/

[52] I. Eclipse Foundation. Desktop ides. Accessed: 2019-04-26. [Online]. Available: https://www.eclipse.org/ide/

[53] P. America, H. Obbink, R. van Ommering, and F. van der Linden, *CoPAM: A Component-Oriented Platform Architecting Method Family for Product Family Engineering.* Boston, MA: Springer US, 2000, pp. 167–180. [Online]. Available: https://doi.org/10.1007/978-1-4615-4339-8_9

[54] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a feature?: A qualitative study of features in industrial software product lines," in *Proceedings of*

the 19th International Conference on Software Product Line, ser. SPLC '15. New York, NY, USA: ACM, 2015, pp. 16–25. [Online]. Available: http://doi.acm.org/10.1145/2791060.2791108

A

Appendix 1

A.1 The Logging Template for Reengineering Activities

INFORMATION
- **Activity type**:
- **Activity**:
- **ActivityID**:
- **VariantID**:
- **Start Date**:
- **End Date**:
- **Description**:

DATA
- **Total Hours spent**:
- **Number of commits**:
- **LOC added**:
- **LOC removed**:
- **LOC modified**:
- **Number of files added**:
- **Number of files removed**:
- **Number of files modified**:

ARTEFACTS
- **Input**:
- **Output**:
- **Tools Used**:

ACTIVITY DESCRIPTION
- **Complexity**:
- **Importance**:
- **Dependencies on other activities**:

A.2 An Example of the Logging Artifact for each Activity

INFORMATION

- **Activity type**: Domain analysis
- **Activity**: Playing game variants
- **ActivityID**: 1
- **Start Date**: 2xxx-xx-xx
- **End Date**: 2xxx-xx-xx
- **Description**: Understanding of the application domain. Running the games to find similar concepts, such as the world, player, enemies and such. This helps in identifying features.

DATA

- **Total Hours spent**: 100
  - **IDE Tracker**: WakaTime (70 hours)
  - **Extracting jar files**: 2 hours
- **Number of commits**: 5 (#16, #17, #18, #19, #20)
- **LOC added**: N/A
- **LOC removed**: N/A
- **LOC modified**: N/A
- **Number of files added**: N/A
- **Number of files removed**: N/A
- **Number of files modified**: N/A

ARTEFACTS

- **Input**: Source Code
- **Output**: N/A
- **Tools Used**: Eclipse IDE

ACTIVITY DESCRIPTION

- **Complexity**: Time consuming but not difficult.
- **Importance**: A domain analysis is crucial to gain knowledge on the domain that the games are in. To be able to find commonalities and features, there has to be extensive domain knowledge.
- **Dependencies on other activities**: None

A.3 Performed activities

INFORMATION

- **Activity type**: Domain analysis, Feature identification
- **Activity**: Running games
- **ActivityID**: A1
- **VariantID**: V1, V2, V3, V4, V5
- **Start Date**: 2019-02-22
- **End Date**: 2019-02-25
- **Description**: Running all the game variants in order to get an overview of implemented functionality and features.

DATA

- **Total Hours spent**: 17
- **Number of commits**: 0
- **LOC added**: N/A
- **LOC removed**: N/A
- **LOC modified**: N/A
- **Number of files added**: N/A
- **Number of files removed**: N/A
- **Number of files modified**: N/A

ARTEFACTS

- **Input**: Source code
- **Output**: Runnable Java application
- **Tools Used**: Eclipse IDE

ACTIVITY DESCRIPTION

- **Complexity**: This activity is considered of low complexity since the source code was either already executable, or we only needed to extract the source code from a jar file.
- **Importance**: This activity is considered highly important for the domain engineering. This activity gave us a high level look at what features exist at runtime.
- **Dependencies on other activities**:

INFORMATION

- **Activity type**: Domain analysis, Feature identification
- **Activity**: Mapping game features
- **ActivityID**: A2
- **VariantID**: V1, V2, V3, V4, V5
- **Start Date**: 2019-02-25
- **End Date**: 2019-02-25
- **Description**: Comparing features from all game variants in order to find commonalities. These features can then be called domain level features.

DATA

- **Total Hours spent**: 5
- **Number of commits**: 0

- **LOC added**: N/A
- **LOC removed**: N/A
- **LOC modified**: N/A
- **Number of files added**: N/A
- **Number of files removed**: N/A
- **Number of files modified**: N/A

ARTEFACTS

- **Input**: List of features retrieved during A1
- **Output**: List of common features
- **Tools Used**: Google docs

ACTIVITY DESCRIPTION

- **Complexity**: This activity is considered of moderate complexity as it can be difficult to decide whether two features from two different variants of large similarities can be unified or not. For instance, two variants may have the exact same menu but with different colors.
- **Importance**: This activity is very important because deciding on the commonalities and variabilities will affect the the entire migration process and can be difficult to change mid-transformation.
- **Dependencies on other activities**: Domain analysis must be finished before conducting this activity.

INFORMATION

- **Activity type**: SPL training
- **Activity**: Reading about compositional SPL
- **ActivityID**: A3
- **VariantID**: N/A
- **Start Date**: 2019-02-26
- **End Date**: 2019-03-05
- **Description**: In order to create a compositional SPL it is necessary to know what is specific for a compositional approach.

DATA

- **Total Hours spent**: 6
- **Number of commits**: 0
- **LOC added**: N/A
- **LOC removed**: N/A
- **LOC modified**: N/A
- **Number of files added**: N/A
- **Number of files removed**: N/A
- **Number of files modified**: N/A

ARTEFACTS

- **Input**: Articles, web pages
- **Output**: Knowledge
- **Tools Used**: N/A

ACTIVITY DESCRIPTION

- **Complexity**: The material available online about compositional SPL is very high level and while there are few examples of compositional SPL, we found

that the most effective way to learn was through trial and error using FeatureHouse in FeatureIDE.
- **Importance**: It is fundamental to understand superimposition and how tools such as FeatureHouse work.
- **Dependencies on other activities**: N/A

INFORMATION
- **Activity type**: Domain analysis, Feature modeling
- **Activity**: Creating a feature model
- **ActivityID**: A4
- **VariantID**: N/A
- **Start Date**: 2019-02-26
- **End Date**: 2019-04-09
- **Description**: In order to create a compositional SPL it is necessary to know what is specific for a compositional approach.

DATA
- **Total Hours spent**: 14
- **Number of commits**:
- **LOC added**: N/A
- **LOC removed**: N/A
- **LOC modified**: N/A
- **Number of files added**: N/A
- **Number of files removed**: N/A
- **Number of files modified**: N/A

ARTEFACTS
- **Input**: List of features from A1 and A2
- **Output**: .xml file of the feature model
- **Tools Used**: Eclipse IDE, FeatureIDE

ACTIVITY DESCRIPTION
- **Complexity**: Once the features has been identified, one only needs to know how to use FeatureIDE to create the feature model.
- **Importance**: Creating the feature model is important because once the transformation begins, changes made to the feature model can cause some feature to lose its implementation.
- **Dependencies on other activities**: A1, A2, A3.

INFORMATION
- **Activity type**: Code cleansing, Feature identification
- **Activity**: Translating code comments to English
- **ActivityID**: A5
- **VariantID**: V1, V2, V3, V4, V5
- **Start Date**: 2019-02-26
- **End Date**: 2019-03-01
- **Description**: Translating comments in the code from German to English with the help of google translate is a first step to understanding the code better.

DATA

- **Total Hours spent**: 22.5
- **Number of commits**: 5
  - 62f531371afd8b7e6d6b4cc9aefd46fcebabff47
  - 28fd5758ce133f7cd0690b35e49de276a5d32b44
  - 620a1955bbcb4ff568b6084e22734d43a883c82b
  - 817f8c3182b3c7eb1266b66e9b087cdbc60a9add
  - a0154f2c383a55205516cc317c3b4b2f1ad68090
- **LOC added**: 0
- **LOC removed**: 0
- **LOC modified**: 3 365
- **Number of files added**: 0
- **Number of files removed**: 0
- **Number of files modified**: 128

ARTEFACTS

- **Input**: Source code (comments in German)
- **Output**: Source code (comments in English)
- **Tools Used**: Eclipse IDE, Google translate

ACTIVITY DESCRIPTION

- **Complexity**: While this activity is simple, with the help of Google Translate, it is time consuming to manually scan all the files. Additionally, once the comments were translated, many of them proved not helpful.
- **Importance**: Since there were no documentation provided with the source code, it was of great important to translate the comments to maximize software comprehension as much as possible.
- **Dependencies on other activities**: N/A.

INFORMATION

- **Activity type**: Feature location, Architecture identification
- **Activity**: Reverse-engineering class diagrams
- **ActivityID**: A6
- **VariantID**: V1, V2, V3, V4, V5
- **Start Date**: 2019-02-28
- **End Date**: 2019-03-01
- **Description**:

DATA

- **Total Hours spent**: 4
- **Number of commits**: 0
- **LOC added**: N/A
- **LOC removed**: N/A
- **LOC modified**: N/A
- **Number of files added**: N/A
- **Number of files removed**: N/A
- **Number of files modified**: N/A

ARTEFACTS

- **Input**: Source code
- **Output**: UML diagrams

- **Tools Used**: Visual Paradigm

ACTIVITY DESCRIPTION

- **Complexity**: This activity is of low complexity as this is automated by Visual Paradigm.
- **Importance**: Analyzing class diagram helped understand and find commonalities between the variants' architecture. Finding variability in the architectural level impacts the transformation phase, hence this activity is important.
- **Dependencies on other activities**: N/A

INFORMATION

- **Activity type**: Feature location
- **Activity**: Finding features in source code
- **ActivityID**: A7
- **VariantID**: V1, V2, V3, V4, V5
- **Start Date**: 2019-03-04
- **End Date**: 2019-03-20
- **Description**: Locating features in the variants' source code with the help of the extracted class diagrams from A6 and feature model from A4.

DATA

- **Total Hours spent**: 22
- **Number of commits**: 0
- **LOC added**: N/A
- **LOC removed**: N/A
- **LOC modified**: N/A
- **Number of files added**: N/A
- **Number of files removed**: N/A
- **Number of files modified**: N/A

ARTEFACTS

- **Input**: Source code, class diagrams, feature model
- **Output**: Excel sheets
- **Tools Used**: Eclipse IDE, Google sheets

ACTIVITY DESCRIPTION

- **Complexity**: This activity is considered of moderate complexity, as it requires careful usage of breakpoints and careful tracking of variables in order to detect where a feature starts and ends. This is further complicated by feature scattering and tangling.
- **Importance**: This activity is mandatory before the transformation.
- **Dependencies on other activities**: A2

INFORMATION

- **Activity type**: SPLE Training
- **Activity**: FeatureHouse research
- **ActivityID**: A8
- **VariantID**: N/A
- **Start Date**: 2019-03-05
- **End Date**: 2019-03-13

- **Description**: There are tools to use when creating feature compositions. In order to be able to make a compositional SPL we need to know about this recommended framework.

DATA

- **Total Hours spent**: 10
- **Number of commits**: 0
- **LOC added**: N/A
- **LOC removed**: N/A
- **LOC modified**: N/A
- **Number of files added**: N/A
- **Number of files removed**: N/A
- **Number of files modified**: N/A

ARTEFACTS

- **Input**: Articles, web pages
- **Output**: Knowledge
- **Tools Used**: N/A

ACTIVITY DESCRIPTION

- **Complexity**: This activity is of moderate complexity. This is mostly because of the instability of FeatureHouse. The existing example projects are usually too simple to showcase complex superimposition hence this tool was mostly learned through trial and error. However, the concept of superimposition are well-explained in literature.
- **Importance**: This activity is of high importance as it is crucial to understand FeatureHouse and how to generate variants and how to test them.
- **Dependencies on other activities**: A4

INFORMATION

- **Activity type**: Diffing, Feature location
- **Activity**: Pairwise comparison of variants
- **ActivityID**: A9
- **VariantID**: V1, V2, V3, V4, V5
- **Start Date**: 2019-03-05
- **End Date**: 2019-05-02
- **Description**: Pairwise comparison of all variants using Code Compare tool to calculate the commonality between the variants. The tool highlights identical, same-class-name-but-different-content, and unique classes of variant A comparing to Variant B.

DATA

- **Total Hours spent**: 52
- **Number of commits**: 0
- **LOC added**: N/A
- **LOC removed**: N/A
- **LOC modified**: N/A
- **Number of files added**: N/A
- **Number of files removed**: N/A
- **Number of files modified**: N/A

ARTEFACTS

- **Input**: Source code
- **Output**: Excel sheet and documents describing the comparison
- **Tools Used**: Code Compare by DevArt, FileMerge, IntelliJ, Eclipse

ACTIVITY DESCRIPTION

- **Complexity**: This activity is of high complexity as there are no tools powerful enough to detect commonalities in the source code. Tools such as CodeCompare only flags classes with identical filenames. Otherwise, we must manually detect which classes contain similarities and then CodeCompare highlights identical code in the file, and if the identical methods are not in the same lines of code in each respective source code files, they will not be flagged by the software. This becomes more complicated when 2 methods in 2 variants are written differently but serves the same purpose. NOw the developer must know which implementation to use.
- **Importance**: This activity is of crucial importance as the better it is done, the most accurate the common code base and variability will be. Understanding commonality in early stages also facilitates the transformation.
- **Dependencies on other activities**: N/A

INFORMATION

- **Activity type**: Code cleansing
- **Activity**: Removing unused code
- **ActivityID**: A10
- **VariantID**: V2, V3, V4, V5
- **Start Date**: 2019-03-11
- **End Date**: 2019-03-12
- **Description**: Identifying code that is not used for the variants. Unused code is then removed to make it easier to analyze variants.

DATA

- **Total Hours spent**: 12
- **Number of commits**: 7
  - 35351f7035e22907d30828cd82a475d6fd012d75
  - 1397a2c35632c474e361da003d7c8027f3d659e7
  - a450d3e51c183344f4c33b27553d2eb5a58e1e40
  - 18211d909a74260df11eacfe6fbec6a00c860ddb
  - 02fef29fcb029135a1fccf0dcad4762a4a64d9b7
  - 5012572ac268c26736bd60a1a0bfb7559d545ae5
  - 3f5c63e01bd8f826a0c6f820c79fe3fd369851e2
- **LOC added**: 0
- **LOC removed**: 11670
- **LOC modified**: 0
- **Number of files added**: 0
- **Number of files removed**: 78
- **Number of files modified**: 133

ARTEFACTS

- **Input**: Source code

- **Output**: Refactored source code
- **Tools Used**: Eclipse, UCDetector, IntelliJ

ACTIVITY DESCRIPTION

- **Complexity**: This activity is of relatively low complexity thanks to the available tools.
- **Importance**: This activity is very important because failure to detect unused code means the developer will spend time transformation source code that is never used.
- **Dependencies on other activities**: N/A

INFORMATION

- **Activity type**: SPLE Training
- **Activity**: FeatureIDE research
- **ActivityID**: A11
- **VariantID**: N/A
- **Start Date**: 2019-03-12
- **End Date**: 2019-03-13
- **Description**: FeatureIDE is a tool to use when creating a SPL, hence it is good to read about FeatureIDE. Both students involved have previously worked with FeatureIDE.

DATA

- **Total Hours spent**: 6
- **Number of commits**: 0
- **LOC added**: N/A
- **LOC removed**: N/A
- **LOC modified**: N/A
- **Number of files added**: N/A
- **Number of files removed**: N/A
- **Number of files modified**: N/A

ARTEFACTS

- **Input**: Book: Mastering Software Variability with FeatureIDE
- **Output**: Knowledge
- **Tools Used**: N/A

ACTIVITY DESCRIPTION

- **Complexity**: Similar to FeatureHouse, the best way to learn FeatureIDE was through trial and error however, FeatureIDE is better documented than FeatureHouse, however the scope of FeatureIDE goes beyond SPL compositional reengineering hence this activity is of low complexity.
- **Importance**: This has the same importance as A8.
- **Dependencies on other activities**:

INFORMATION

- **Activity type**: Transformation, Quality assurance
- **Activity**: Transforming source code to feature
- **ActivityID**: A12

- **VariantID**: V2,V3,V4
- **Start Date**: 2019-03-14
- **End Date**: 2019-05-02
- **Description**: With feature-oriented programming, from the feature model and compositional SPL, we transform parts of the source code into different features from the feature model. Every features functionality is manually tested. continuously.

DATA

- **Total Hours spent**: 207
- **Number of commits**: 10
  - 47cc01014a3b20506cb40a60da46f77cf6f1b2d2
  - 53361f57464568c09c2d1754b8248104c39b28c5
  - 319aa8090ba75df16e28824ebf4deea250d6b712
  - f2418fa6b2fe1e2d7c900d49ff3065f43deb981f
  - 29d7f267084446b746a1261cbe12b28363973f89
  - 8062c850e2bdc0d1c7ac72046342650d120e2b76
  - 4d4beaecfe695dd2cbf3ae2ef22b1f85c5e8516f
  - e468b3ace02f8dcb0cd1456f741e93939da4bdbe
  - 4dfa44677907424c036390dbe7bc248ef998e648
  - 1f83f2b640b34717aaa938489d737d8035c6c33c
- **LOC added**: 17874
- **LOC removed**: 1492
- **LOC modified**: N/A
- **Number of files added**: 184
- **Number of files removed**: 4
- **Number of files modified**: 80

ARTEFACTS

- **Input**: Source code
- **Output**: Separated features
- **Tools Used**: Eclipse, FeatureIDE

ACTIVITY DESCRIPTION

- **Complexity**: This activity is of high complexity and time consuming. This resulted in the detection of several challenges in the migration process that are listed in the discussion.
- **Importance**: This activity is mandatory and the the migration process cannot procede without it.
- **Dependencies on other activities**: A4, A7, A9.

A.4 Notes After Running Games

Notes were taken during the process of running games to identify features. The notes can be seen below.

A.4.0.0.1 ApoCheating Description:

2D game A student in a classroom full of students and teacher(s), where the player (student) can move around the classroom to cheat (increase test score) avoiding the teachers cone of vision.

Features

- Menu
  - Description Area (how to play) overlayed on classroom
  - Choose level with arrows (14 levels)
  - Start option
  - Load option
  - Stop option
  - Score board
    - Cheated %
    - Detected %
    - Number of coins in the level
  - Random Option
    - Randomize location of students that you can cheat from
    - Enable Detect highlights students you can cheat from
  - Quit option
- In-game Menu (after finishing a round)
  - Replay option
  - Next option
  - Scoreboard
- World
  - Classroom full of tables
  - Characters are presented in circles
    - Yellow circle (student cannot cheat from)
    - Green circle (student can cheat from)
    - Grey circle (student masked whether or not you can cheat from)
    - Red circle (teacher) enemy
    - Blue circle (main character)
- Enemies
  - Has green cone of vision (detection zone) that moves and changes in size
  - Can move
  - Speed of cone of vision changes
  - Can be many teachers in one room
  - Detection zone changes color if it is placed on player expect if player is in his designated seat
- Player
  - Can move using all 4 keyboard arrows

  - Press and Hold space to cheat (when nearby a student you can cheat from)
  - Cheating process slows down on harder level and detection rate goes up
- End game score
  - Losing or winning level gives you a scoresheet in form of exam and description (looser, very good, perfect and numerical score (increases the more u are detected)

A.4.0.0.2 ApoIcarus Description:

Character jumping on clouds. Automatically jumping, just movement with arrow keys. Missing a cloud and fall out of screen leads to game ends.

Features

- Menu
  - Start
  - Achievements
  - Help
  - Highscore
  - Options
  - Quit
- World
  - Clouds
    - white/black/yellow
    - White: normal
    - Black: disappear after 1 jump
    - Yellow: invisible after a while (timer)
  - Clouds can move
  - Trampoline to give extra jump
  - Feather to fly (timer)
  - Wings to fly (timer)
- Enemies
  - Standing still / moving left&right
  - Kills you if touching character
  - Kill enemy by landing on top of it with character / shoot it with arrow
- Player
  - Move left&right (a&d OR left&right arrow)
  - Mouse1 to shoot arrows
  - Aim arrow with mouse alignment
- Counting points (score), highscore list
- Counting elapsed time

A.4.0.0.3 ApoNotSoSimple Description:

Moving a character through different worlds/levels with the goal to reach a certain point. The world consists of obstacles that if touched resets character to start position.

Features

- Menu

  - Play
    - Redirects to Level chooser
  - Userlevels
    - Step through different levels / sort by solution, username, levelname
    - Right side of screen choose levels
    - Left side of screen (most part) shows level
  - Editor
    - Step through views a level can have
    - Choose name / description / username
    - Right side of screen choose view, choose figures to put in level
    - Left side of screen (most part) shows level and name / description
  - Options
    - Checkboxes for sounds
      - Music
      - Effects
  - Level chooser
    - Rows of circles with number inside to pick level
- Player
  - Move with arrowkeys
- World
  - "Goal/finish"
  - Obstacles
    - Circles (if stepped into makes character reset)
    - Arrows (Moves vertically when you press up/down, if stepped into character reset)
- Enemies
  - See obstacles in World
- Counting moves/steps (NO HIGHSCORE)
- GUI
  - Menu has game name in center and
  - boxes vertically with e.g. options Level chooser / options has game name and "location" in center
  - Game has big box on left side, step counter + level indicator on right side

A.4.0.0.4 ApoSnake Description:

Similar to classic snake, but several level has more than one snake to control. Additionally, the bits that the snake consume are colored differently and the snake is colored like the last bit consumed. Finally, the snake can traverse walls that are of different color than the snake.

Features

- Menu
  - Puzzle option
    - Display a grid of different levels (level chooser)
      - Levels are locked until the previous one is completed.
- Userlevels

  - Display user created level (this can crash the game, more specifically when going through the level it crashes with string index out of range 1)
- Editor
  - Editor offer a customizable grid where the user can add the following components:
    * Red snake
    * Blue snake
    * Green snake
    * Red block
    * Blue block
    * Green block
    * Red bit
    * Blue bit
    * Green bit
    * Increase height of grid
    * Decrease height of grid
    * Increase width of grid
    * Decrease width of grid
    * Clear cell (reverts it into empty cell)
    * Test option (to test level)
    * Back option
- World
  - Upper part of screen has:
    * Level number
    * Name of the game
    * Number of moves
  - Middle part has:
    * Box where game occurs
    * Snake
    * Empty cells
    * Bits
    * Blocks
    * Box introducing new game mechanics (not always present)
  - Lower part has:
    * Previous level button
    * Restart button
    * Next level button
    * Back button

A.4.0.0.5 ApoStarz Description:

A “star” should be moved from point A to point B, by rotating the “world”.

Features

- Menu
  - Turorial
  - Normal
  - Time trial

  - Highscore
- Player
  - A “star” that is moved by gravity when turning the “world” with arrow keys
- World
  - Goal/finish
  - Obstacles blocking character
  - The world can move by using arrow keys “left/right”
- Counting steps/moves
- Highscore
- Levels

A.5 Early Bottom-up Feature Model

Figure A.1 shows what the Feature Model looked like after the final iteration.

A.6 Finalized Feature Model

Figure A.2 shows what the Feature Model looked like after the final iteration.

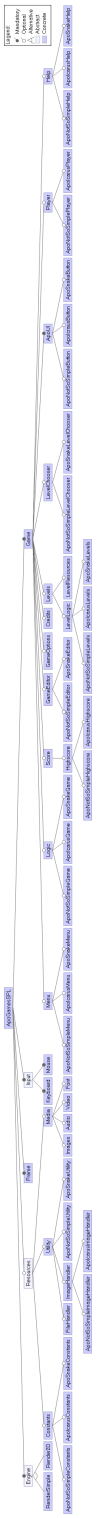

Figure A.1: Feature Model Extracted From Bottom-up Approach

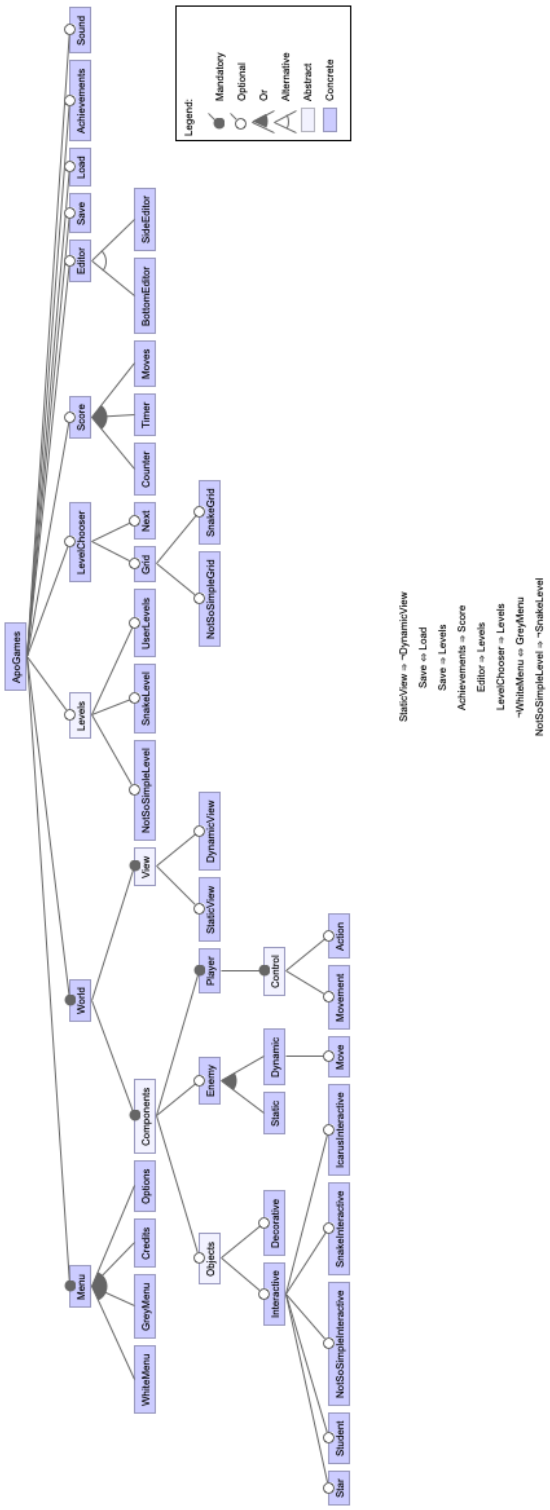

Figure A.2: End-result of the Feature Model

1. **ApoGames:** This feature represent the common-base of all the variants. In other words, this feature contain all the code shared by every single variant, and none of the variants can be generated if this feature is disabled. This feature contains code fragments responsible for things such as the game panel, game engine and the main method that is the starting point for every variant. This feature is mandatory.
2. **Menu:** This feature contains the common code that is responsible for rendering the menu. This includes the common buttons such as Play, Quit and so forth.
3. **World:** This feature instantiate the game level and logic depending on the configuration selected.
4. **Levels:** This is an abstract feature, parent of game levels and user levels.
5. **Level Chooser:** This feature instantiate the level chooser depending on the configuration (whether its grid of level of an endless runner game).
6. **Score:** This feature loads high score from an URL.
7. **Editor:** This contain the common code responsible for building the game editor.
8. **Save:** This feature handles all the persistent saves done by the user.
9. **Load:** This feature handles all the loading of files done by the user.
10. **Achievements:** This loads achievements accomplished by the player.
11. **Sound:** This feature activates and deactivates the sound in games.
12. **WhiteMenu:** To establish a sense of branding in the SPL, we limited the menu to white and grey menu. This feature is responsible on displaying menu content in white colors.
13. **GreyMenu:** This feature is responsible on displaying menu content in grey colors.
14. **Credits:** This enables the option to see the credits of the game.
15. **Options:** This toggles the Options of the games.
16. **Components:** This is an abstract feature that is a parent to objects, player and enemies, this is because all the three entities share similarities in the code level.
17. **View:** This is an abstract feature, parent of Dynamic and Static view.
18. **NotSoSimpleLevel:** This feature loads the level and level logic of the ApoNotSoSimple game.
19. **SnakeLevel:** This feature loads the level and level logic of the ApoSnake game.
20. **UserLevels:** This feature loads the level created by the users.
21. **Grid:** This feature presents all the level in a grid.
22. **Next:** This feature only shows the next level (with a next button).
23. **Counter:** This feature sets a counter than the player must beat to win the game.
24. **Timer:** This feature counts the game time spent by the player.
25. **Moves:** This feature tracks the number of moves done by the player.
26. **BottomEditor:** This loads the game editor panel at the bottom of the screen.
27. **SideEditor:** This loads the game editor panel on the side of the screen.

28. **Objects:** This is an abstract feature that is a parent to decorative and interactive objects.
29. **Enemy:** All actions and logic of enemies are represented in this feature.
30. **Player:** The players action and resources are represented in this feature.
31. **StaticView:** Static view is the feature responsible for showing the world in a
32. **DynamicView:** This feature is responsible for changing the background (world) for endless runner types of games.
33. **NotSoSimpleGrid:** This dispalys the level grid in white style (ApoNoto-Simple theme).
34. **SnakeGrid:** This dispalys the level grid in grey style (Snake theme).
35. **Interactive:** Interactive objects are objects that the player can interactive with, for instance an enemy or a power-up.
36. **Decorative:** This feature displays object that the player can not interact with (for instance a window).
37. **Static:** This enables enemies that do not move nor shoot any projectiles but kills the player once they come in contact with them.
38. **Dynamic:** This enables enemies that can move and shoot projectiles (such as arrows)
39. **Control:** This is an abstract feature parent for movement of the player and player's action
40. **Star:** This is a power-up star that the user can pick up.
41. **Student:** This feature represent a student that a student can interact with
42. **NotSoSimpleInteractive:** This feature loads all objects that a player can interact with such as power-ups taken from ApoNotSoSimple variant.
43. **SnakeInteractive:** This feature loads all objects that a player can interact with from ApoNotSoSimple variant.
44. **IcarusInteractive:** This feature loads all objects that a player can interact with from ApoNotSoSimple variant.
45. **Move:** This feature represent the movement of an enemy.
46. **Movement:** This feature represent the implementation of the player movement.
47. **Action:** This feature represent different interaction the player can have with the world (eg. opening a door).

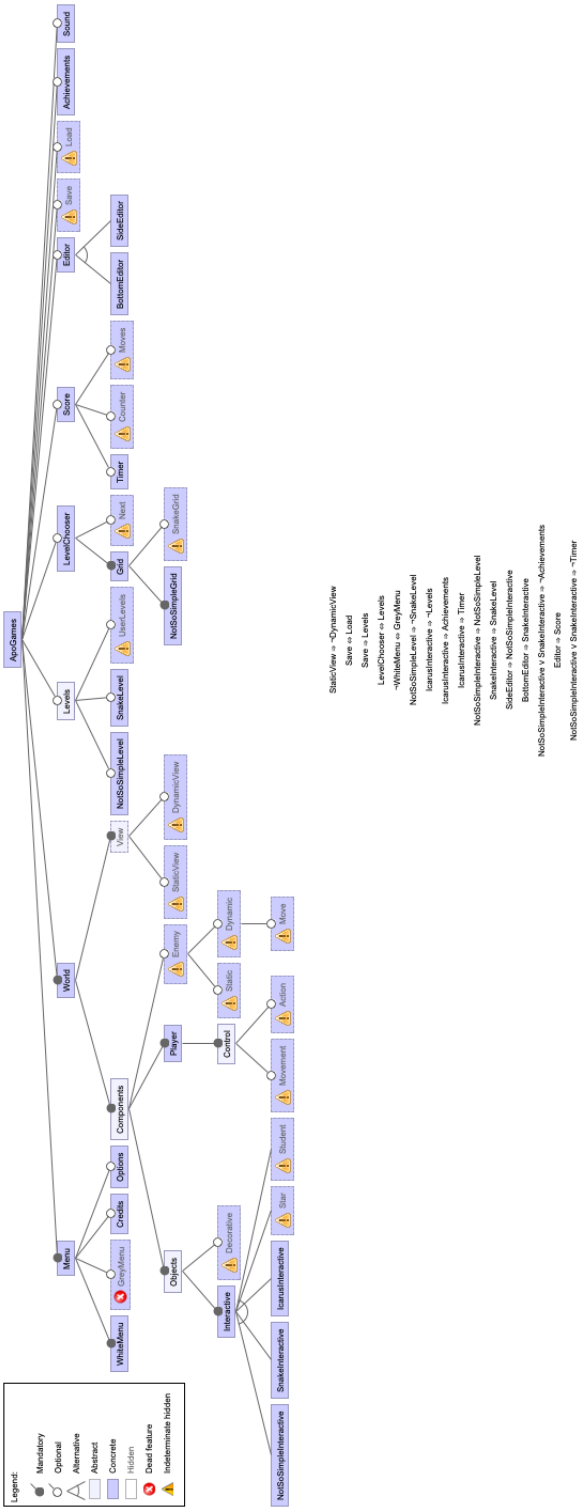

Figure A.3: Modified final Feature Model to generate 56 products

A.7 Class Diagrams of Java variants

A.7.1 Class Diagram for ApoCheating

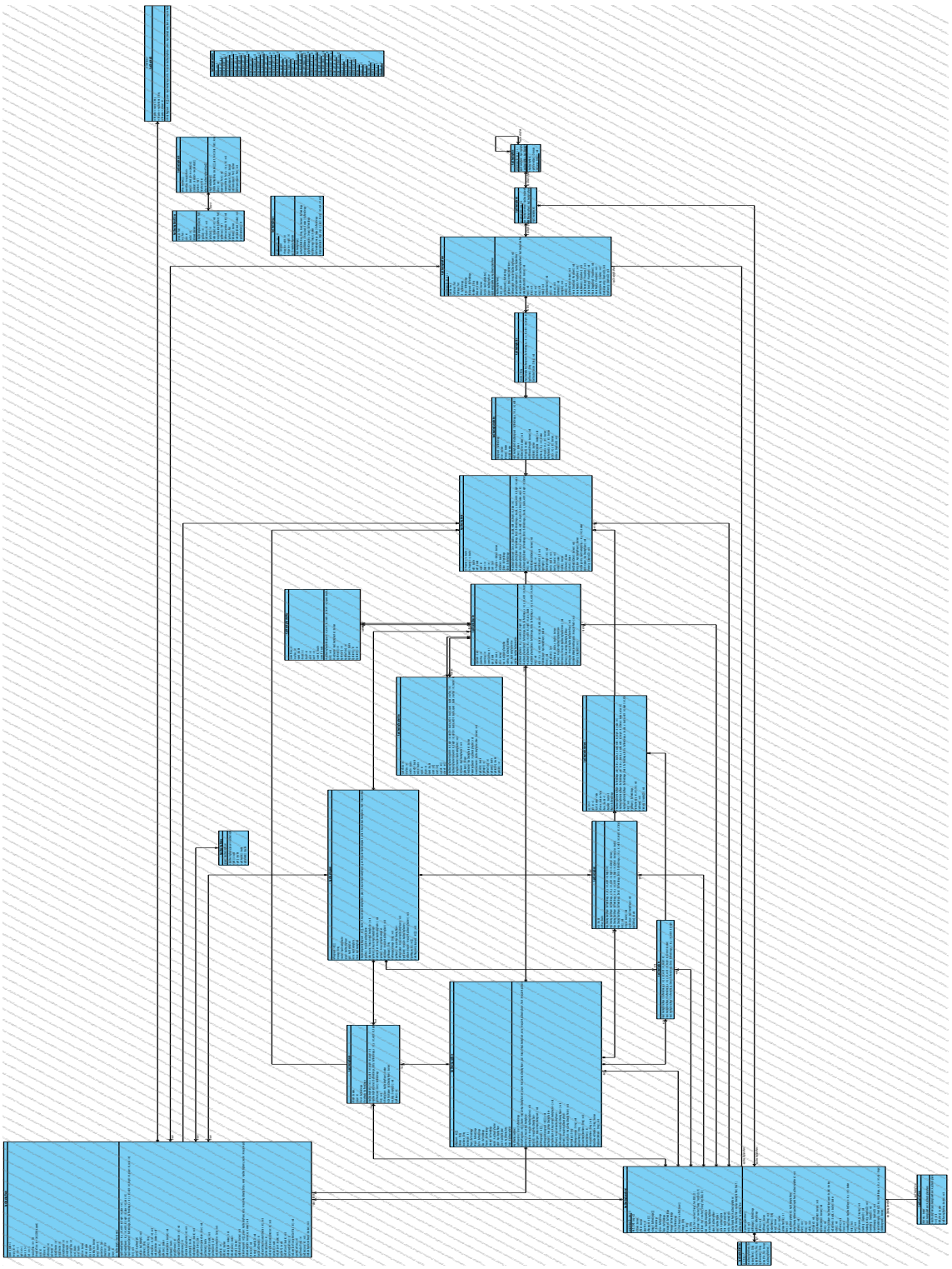

Figure A.4: ApoCheating Class Diagram

A.7.2 Class Diagram for ApoIcarus

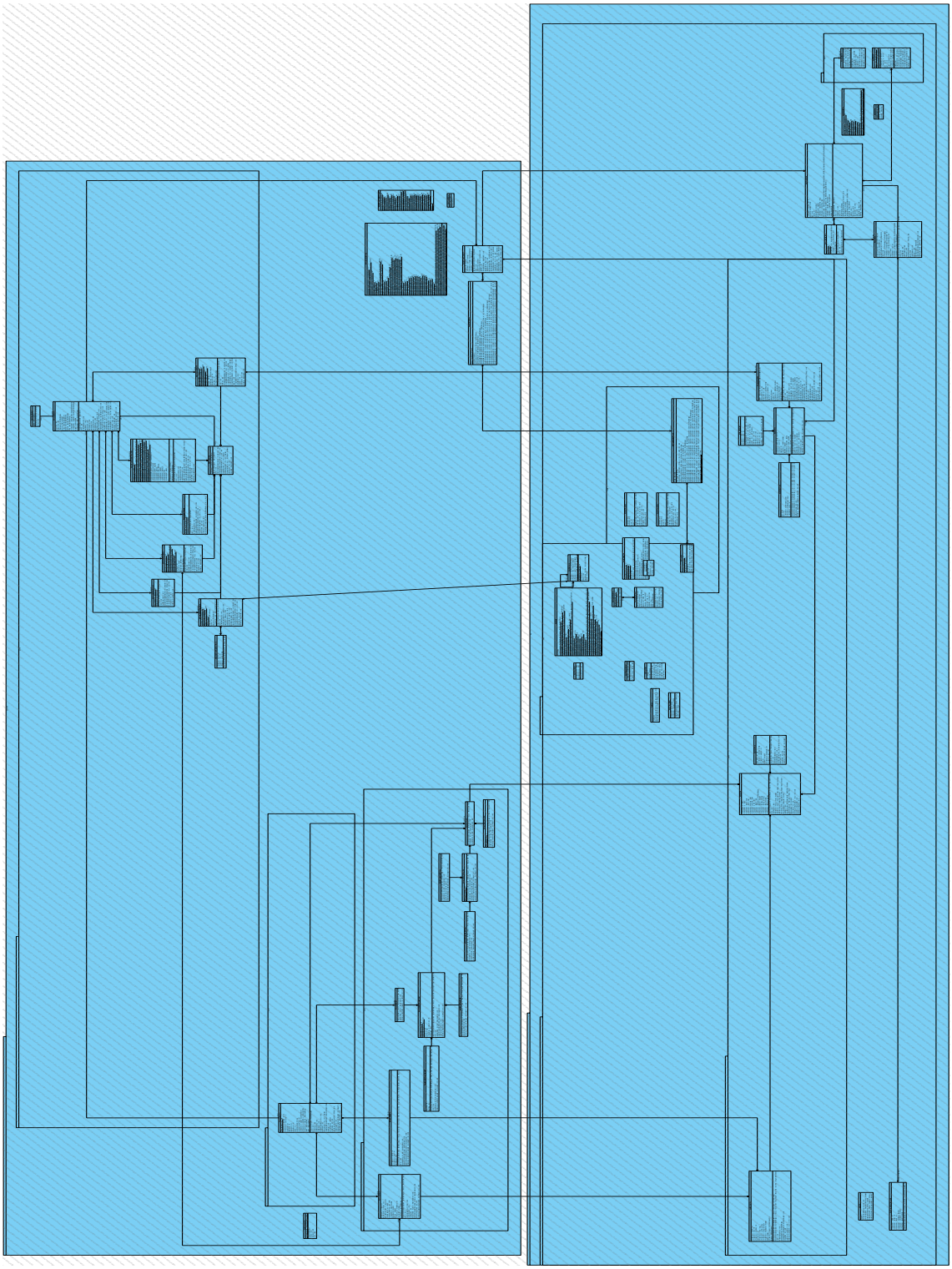

Figure A.5: ApoIcarus Class Diagram

A.7.3 Class Diagram for ApoNotSoSimple

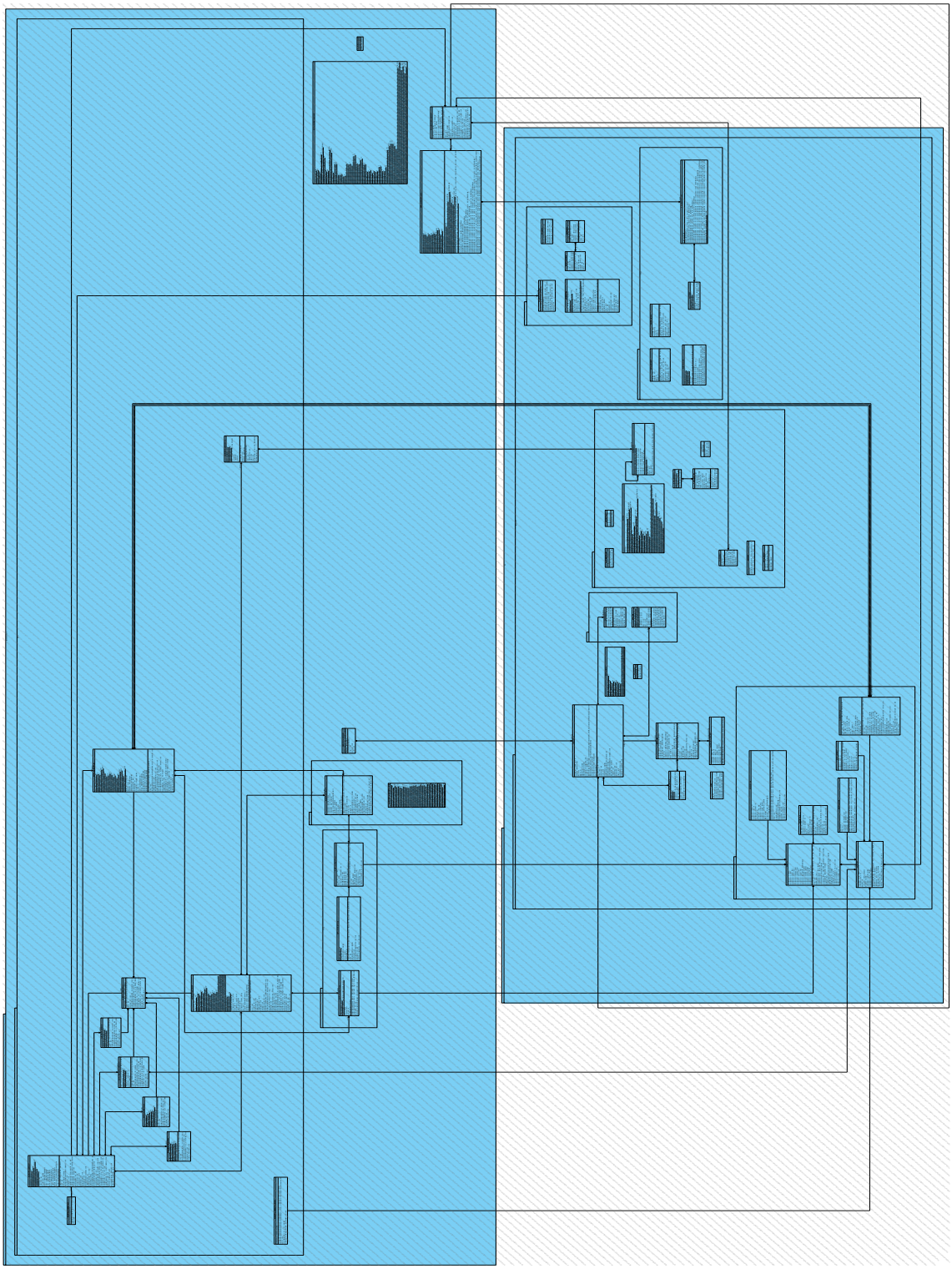

Figure A.6: ApoNotSoSimple Class Diagram

A.7.4 Class Diagram for ApoSnake

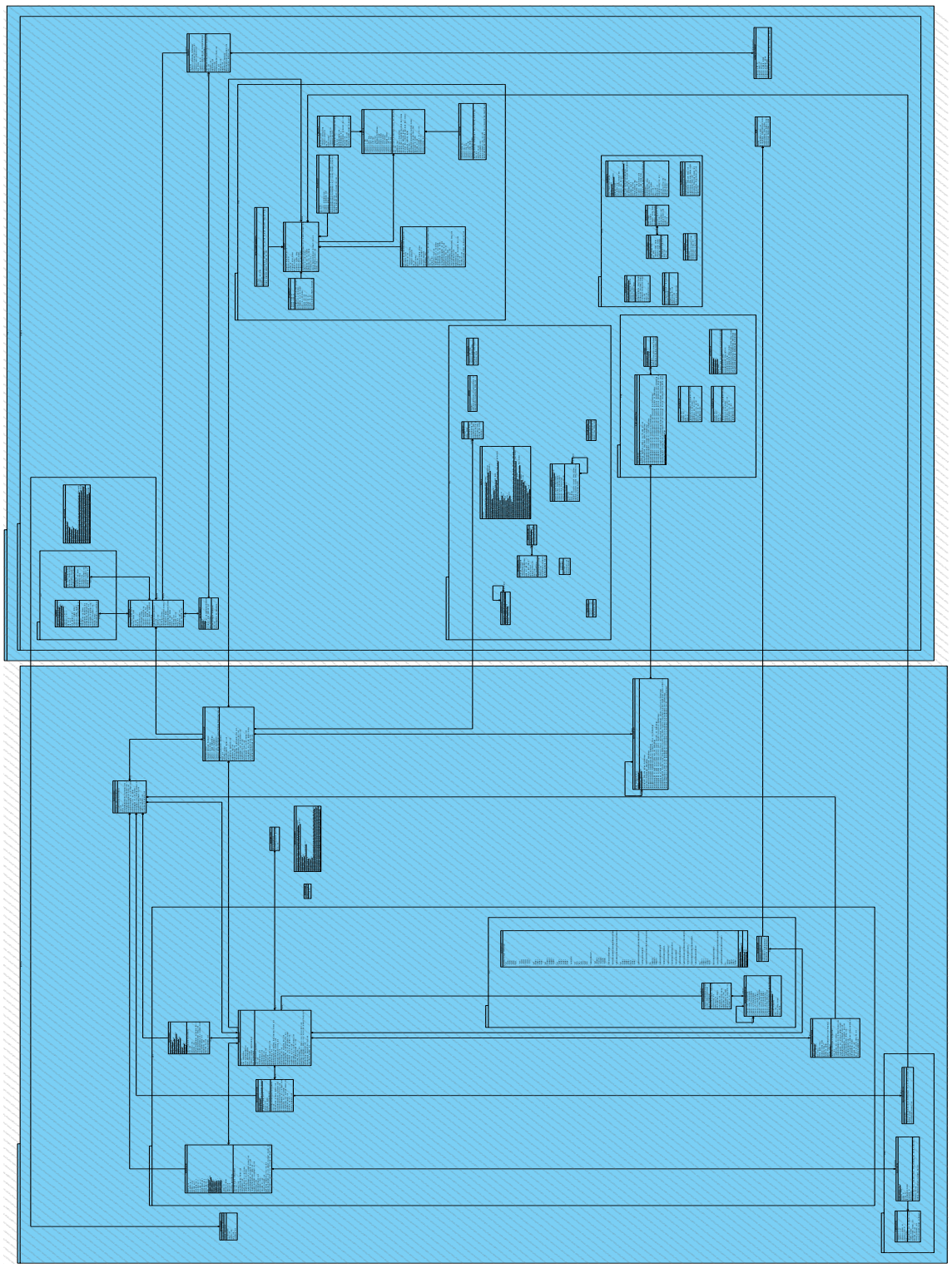

Figure A.7: ApoSnake Class Diagram

A.7.5 Class Diagram for ApoStarz

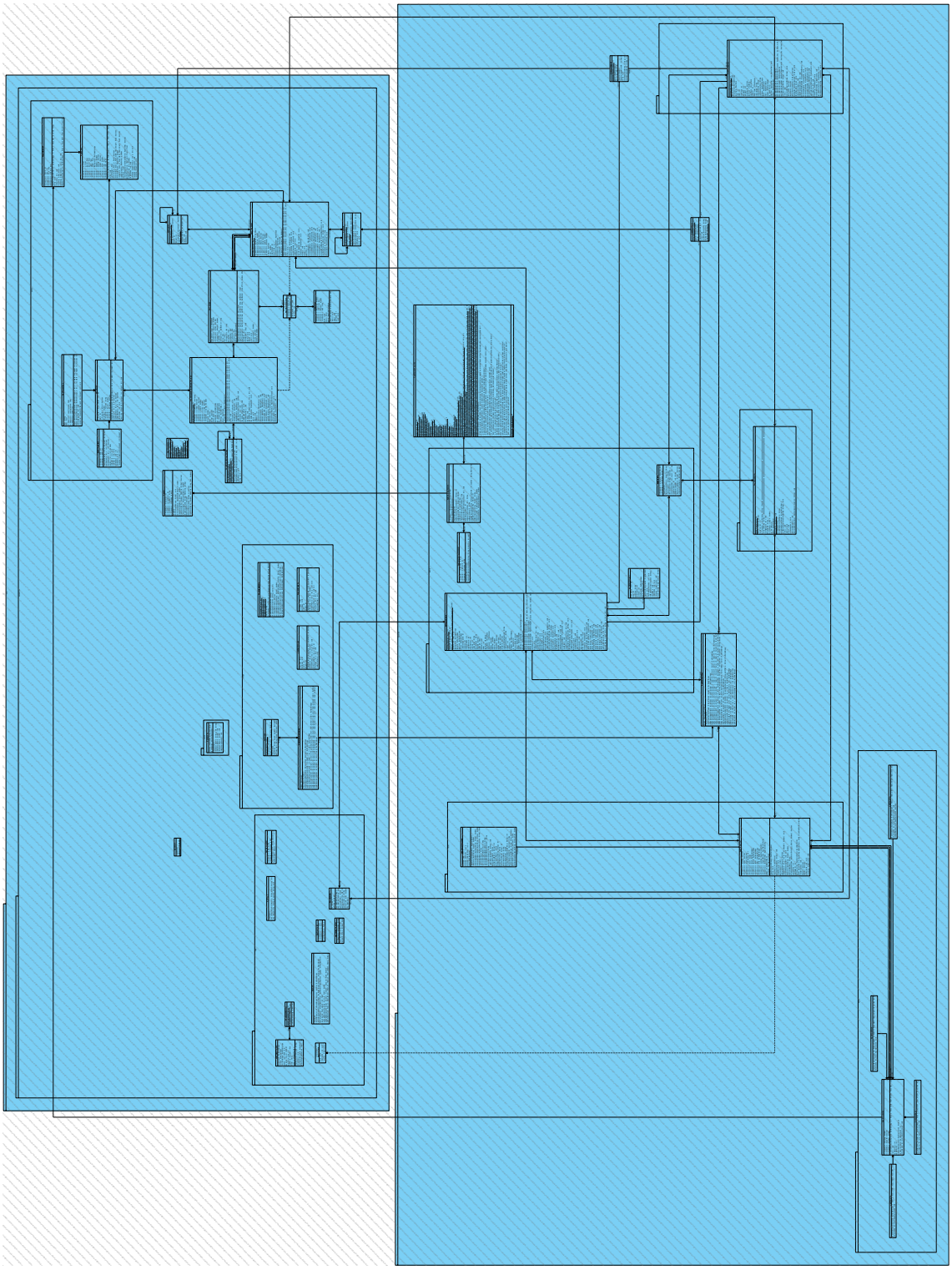

Figure A.8: ApoStarz Class Diagram