CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Migration of Cloned Product Variants into an Annotative Software Product Line

An experimental case study on Android applications

Master's thesis in Software Engineering and Technology

SEBASTIAN NILSSON
JONAS ÅKESSON

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

Master's thesis 2019

Migration of Cloned Product Variants into an Annotative Software Product Line

An experimental case study on Android applications

SEBASTIAN NILSSON
JONAS ÅKESSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

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

Migration of Cloned Product Variants into an Annotative Software Product Line
An experimental case study on Android applications
SEBASTIAN NILSSON
JONAS ÅKESSON

Supervisor: Thorsten Berger, Department of Computer Science and Engineering
Examiner: Riccardo Scandariato, 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

Migration of Cloned Product Variants into an Annotative Software Product Line
An experimental case study on Android applications
SEBASTIAN NILSSON
JONAS ÅKESSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Software-product-line engineering (SPLE) is a systematic way of using software engineering methods, tools and techniques in order to create a collection of similar software-based systems. The idea of SPLE is that a product is defined based on the features of the product. The features are identified and structurally defined in the code. This is called software product line (SPL). A new variant of a product is created by selecting a certain set of features to fulfill the requirements of that product.

A common problem with SPLE is that companies often have trouble in deciding whether or not to perform an SPL migration, since they are unsure of the costs of doing so. Even if they decide to perform a migration, they do not know which approach is the best for them. These issues are the motivation for this thesis.

We have performed a migration on a set of Android games into an annotative SPL. An annotative SPL means that features are defined in the code in form of annotations and that the products are included in a common code base. The games were written by using a clone-and-own approach, meaning that products are cloned and modified to match some requirements. We migrated the games to an SPL by extracting the common elements of these games and annotating them as features in the code.

Each activity that we performed was logged with a measurement approach that we designed with another thesis group that did a migration with another approach called composition-based. This thesis presents what activities should be performed when doing a migration to an annotative SPL. It also presents the characteristics and costs of these activities, and there is also a comparison between using an annotation-based approach and a composition-based approach.

This study finds that there are several different activities that should be involved when performing a migration. The activities have different characteristics in terms of complexity, importance and timescale. By comparing to the composition-based approach, we found that the annotation-based approach is easier to start with when you are a novice to the SPL topic and that it also cost more to perform in terms of time.

Keywords: software product line, software-product-line engineering, annotation-based, migration

Acknowledgements

We would like to thank our supervisor, Thorsten Berger, for all the help during this project. We would also like to thank Jacob Krüger at Otto von Guericke University Magdeburg for all the help during this project.

Sebastian Nilsson, Gothenburg, June 2019
Jonas Åkesson, Gothenburg, June 2019

Contents

List of Figures

List of Tables

1
Introduction

Software-product-line engineering (SPLE) is a systematic way of using software engineering methods, tools and techniques in order to create a collection of similar software-based systems. The field first gained traction in the late 1960s, but it was not until the 1990s that it became more widely used. Some companies that have since adopted this strategy with great success are: Boeing, Bosch, Hewlett Packard, Toshiba and General Motors. Even though these companies are in widely different fields, they still have been getting much value out of this strategy [Apel et al., 2013].

The idea of SPLE is that a product is defined based on the features of the product. Features are identified and structurally defined in the code. This is called a *software product line* (SPL). A new product is created by selecting a certain set of features to satisfy the requirements of a new product.

The main motivation for creating SPLs are usually that companies create too many clones of their product which lead to maintenance issues. They realize they need to perform an SPL migration, but the costs of doing so are unclear. This study aims to research the activities and effort needed to transform several Android games into an annotative SPL. This will be done by performing an actual SPL migration and measuring the effort using different measures.

The subject systems that the migration will be performed on are a set of Android and Java games called the Apo-Games. They are developed by Dirk Aporius using the *clone-and-own* approach, which means that products are cloned and modified to match some given requirements [Krüger et al., 2018]. This thesis is the response to a challenge case from the Apo-Games case study [Krüger et al., 2018] and can be described as an empirical case study. The case study provides 20 Java games and 5 Android games, with five challenges to the research community. These challenges include: reverse engineering feature models, feature location, code smell analysis, architecture recovery and migrating the games to an SPL. Note that while an SPL migration will be performed, all of these challenges will not necessarily be done, the main focus is the actual migration.

There are mainly two approaches that are used in practice when migrating to an SPL, *annotation-based* and *composition-based* [Apel et al., 2013]. The difference between these are in the way that features are represented in the code, as well as how new products are created. In an annotation-based approach, the features are directly marked in the code in the form of annotations and the code from all prod-

ucts are included in a single code base. The traditional way to annotate the features are by using C preprocessor constants such as *#ifdef* [Kästner and Apel, 2008]. This means that the code that belongs to a certain feature is maintained under the *#ifdef* annotation. For example, *#ifdef FirstFeature* defines the code that belongs to FirstFeature. In this project, a simple Java preprocessor will be used to the same effect. Another thesis project will perform a similar study, however they will be using a composition-based approach to perform the migration. The results of these approaches will be compared to draw conclusions on the effectiveness of them.

1.1 Statement of the Problem

Krüger et al., state that while SPLs are far superior to clone-and-own practices, many companies still use clone-and-own practices. This is because they might fear the upfront cost of creating an SPL, or being uncertain of how their product might look like in the future. With an increasing amount of similar products, the effort of maintaining them increases as well. The solution can be to extract similar elements and merge them into an SPL [Krüger et al., 2018].

Creating SPLs are often risky and very costly [Apel et al., 2013]. With several different products, extracting similar features in each of the clones is a very complex task, as the dependencies and interactions between the features are often poorly documented. Locating these features is therefore time-consuming and challenging. This process is called feature location and most of time it has to be done manually, as automated tools are not as accurate [Krüger et al., 2018]. In addition to locating features, there is often a lack of understanding with regards to the other tasks that have to be performed. The costs of these tasks are often hard to estimate without real data. In addition, if a company decided to perform an SPL migration, there is still the problem of choosing which approach to use.

1.2 Purpose of the Study

The purpose of the study is to understand the activities and the costs needed to migrate existing products into an annotative SPL. Many organizations need the means to decide whether they should migrate to an SPL or not, but the costs are not clear. It is also not entirely clear what tasks should be performed to minimize costs while performing the best migration possible. Choosing the suitable approach to the migration is also a problem., This thesis will provide fine-grained cost data with regards to the migration tasks which should be valuable for organizations considering migrating to an annotative SPL. The Apo-Games are suitable for this task since they are somewhat complex, with several layers of abstraction to them. They all have complex application logic which is found in many real-world scenarios.

2
Annotation-based SPL

This chapter describes the fundamentals of software product product line engineering, with focus on annotation-based approaches. It also introduces and describes cost models, which will serve as a foundation for our measurement approach.

2.1 SPLE and Features

As mentioned in Chapter 1, an SPL is a way to structure code to promote the creation of new products with varying requirements. This means that similar product variations can be created with differences between them, depending on what the customer requests. SPLs are feature-based, meaning that a new product variant can be configured based on the features that are needed to satisfy the requirements of that product.

There are several reasons for migrating from a clone-and-own approach to an SPL. According to Apel et al. [2013], some of the biggest motivators are the following:

1. New variants can easily be created, tailor-made according to customer requirements. Instead of having a few standard versions of a product, SPLs support a bigger range of variations.
2. While the upfront cost of building the SPL is higher than building a single product, the cost of the SPL is far cheaper given enough time. This is due to new products not having to be designed and built from scratch.
3. Another important factor is the increased quality that comes from utilizing an SPL. Standardized parts are often controlled and tested thoroughly in comparison to code that is very specific and seldom used. In general, the more a part of code is used, the better quality it will have.
4. SPLs have the potential to reduce time-to-market significantly compared to traditional systems. If a customer's requirements are supported by existing features, a new variation of the product can be created very quickly. In addition, building new features can often be done easily in an SPL compared to building the product from scratch.

Managing several similar versions of a product can be challenging and costly if they have to be modified down the line [Antkiewicz et al., 2014]. For example, if a bug is encountered in the applications, the bug would have to be located and fixed in all of the cloned products if a clone-and-own approach has been used. In the case of an

SPL, the bug should be tied to a single feature and would only have to be fixed once.

Another motivator for migrating to an SPL is the branding aspect. Certain parts of cloned variants can be modified in the migration process to strengthen the company's brand and to make it obvious that all products are owned by the same company. An example of this is Angry Birds, with lots of similar but different products that have strong branding. When the company Danfoss [Jepsen and Beuche, 2009] migrated their cloned variants into an SPL, they integrated all of their menus etc., which can be seen as a kind of branding. The benefits of this can be that features become more unified and familiar to users, which is important to companies.

There are mainly three different ways in which an SPL migration can be performed: proactive, reactive and extractive [Apel et al., 2013]. The proactive approach involves an extensive domain analysis and focuses on features and variation from scratch. The reactive approach revolves around creating or updating an SPL when new products are created, due to new demands or requirements. On the opposite side of the spectrum, the extractive approach is when existing products are analysed to extract common and varying code into a single platform, which is the approach we will use for this migration. The extractive approach often involves activities such as refactoring old code and locating features that are scattered in many places. As the method relies on existing code rather than preplanned guidelines, it often leads to a less maintainable code base [Apel et al., 2013]. The approach requires extensive domain analysis and feature location efforts for these reasons.

A systematic mapping study was conducted by Assunção et al., with the purpose of providing an overview of the current research in the SPLE field [Assunção et al., 2017]. They concluded that there is a great interest in the field in the research community, despite a decrease in publications between 2014-2015. The study found that the most common inputs in an SPL migration are source code and requirements, while the most common outputs are feature discovery and refactored source code. For our migration, source code is the only input we have to work with.

A study on the topic of clone-based variability management showed that clone-based variability practices such as clone-and-own is widely used in the Android ecosystem, due to the fact that it does not offer any variability management [Businge et al., 2018]. On the other hand, platforms such as GitHub and BitBucket have enhanced the process of cloning software, by the use of forks and pull requests. The study also showed that most of the time, there is no form of code propagation performed within an app family, which means that there is no code sharing between the main variant of the app and the copies of the app. The result of this is poor code reusability, which often is a sign of wasted resources. Another interesting finding is that most of the variations of apps found are not developed by the same person. This can become an issue if the variability in the code is very different, since that will make an SPL migration harder.

Dubinsky et al. [2013] have observed through their study regarding cloning in in-

dustrial software product lines that people involved in cloned product lines favor cloning as a reuse approach. This is mainly because cloning has a low entrance barrier. Despite this, the long-term problems that are introduced by using cloning cannot be ignored. It is important to clearly identify the benefits of any approach trying to re-engineer cloned product lines into structured SPLE models since the engineers will have to abandon the current cloning practice.

Because of these issues of migrating to an SPL, a more flexible approach have been presented by Antkiewicz et al., a strategy called virtual platform [Antkiewicz et al., 2014]. The purpose of the virtual platform is to enable organizations to have the benefits of a fully-integrated platform while still having the benefits of a clone-based approach. The difference compared to an SPL, in which reusable assets are contained within a platform, is that a clone management approach is used to make assets reusable. There are many intermediate points between a clone-and-own approach and an SPL, which means that committing to an SPL is not always the best idea. The conference paper presents six different levels of software reusability, ranging from different types of clone-and-own to a fully integrated SPL.

2.1.1 Annotation-based vs Composition-based

As mentioned before, annotation-based means that the code from all products is included in a single code base, and features are marked directly in the code in the form of annotations. These annotations are commonly based on C preprocessor directives, such as *#ifdef*. Features can then be included and excluded in a product using preprocessor constants such as *#define Feature1*. Annotation-based approaches are commonly used due to their ease of use, as well as many languages having support for preprocessor directives. However, the ease of use comes with a price, as the annotation-based approaches are often critized due to their potential complexity, lack of modularity as well as the reduced code readability and feature traceability [Ji et al., 2015].

On the other hand, composition-based approaches locate features and separate them into different locations, such as dedicated files, containers or modules. According to Apel et al. [2013], a classic example of a compositional approach is a framework that can be extended upon using plugins, preferably one feature per plugin, so that different products can be generated using different plugins. This approach leads to a more structured code base and SPL, compared to an annotation-based approach. The main drawback of using this approach is the challenge that comes with maintaining the relation between a feature and its implementation.

The approaches are often compared using a set of characteristics that best describe their strengths and weaknesses. Some of these characteristics are: feature traceability, language independence, granularity and SPL adoption [Krüger et al., 2016; Kästner and Apel, 2008].

Feature traceability refers to the ability to trace a feature from the feature model

to its implementation in the SPL. This can be especially important when a developer is trying to fix a bug related to a certain feature. Compositional approaches support feature traceability more than annotative approaches, since all of a feature's code is contained in a single place in the compositional approach, while the code might be scattered in the annotative one.

Language independence is the dependency of a certain language when performing an SPL migration. Since features are directly annotated in the annotative approach, it can often be language dependent. However, with simple tools such as Antenna, a Java preprocessor, this language dependency can easily be bypassed.

Granularity describes the level at which features are implemented. This can either be *coarse-grained* or *fine-grained*, where coarse-grained can be variability at file or folder-level, and fine-grained can be variability at line-level. Annotative approaches support fine-grained variability for a low effort, while compositional approaches generally only support coarse-grained variability.

SPL adoption simply refers to the effort necessary to migrate existing products into an SPL. Annotative approaches are by far easier to adopt, for several reasons. Preprocessors are a part of many programming languages and easy to use. Compositional approaches influence existing code and processes too much for many companies to adopt a migration of this kind. These factors make it more likely for a company to adopt the annotative approach in the early stages of a project, and then later refactor the code into separate units.

2.1.2 Feature Models and FeatureIDE

A feature model is a way to model all features in an SPL, which shows their dependencies and constraints. Perhaps most importantly it shows valid configurations and enables creation of configurations [Classen et al., 2011]. Feature models helps illustrate dependencies between features effectively, as can be seen in Figure 2.1. A popular tool to maintain feature models is FeatureIDE[1], which is an addon for Eclipse IDE [Kästner et al., 2009]. In FeatureIDE, feature models can be created easily and a feature's code can be written directly in the IDE. To further customize the feature model, there is a constraint wizard which handles constraints between features. A constraint can be that one feature has to be selected if another feature is selected, which would limit the amount of valid configurations.

Figure 2.1 displays a simple example of a feature model created in FeatureIDE. A feature can either be mandatory or optional, which is shown by having a grey respectively white circle as connection type. A feature can have several different ways of being implemented. Figure 2.1 illustrates how a payment can be done either by credit card, bank transfer or debit card. Features can also be alternative, meaning that only one of the features can be implemented at the same time. Note that payment and security are abstract, which implies that they do not have any

[1]FeatureIDE: http://featureide.com/

impact on implementation level. The opposite of an abstract feature is a concrete feature, which means that it is implemented in the code base.

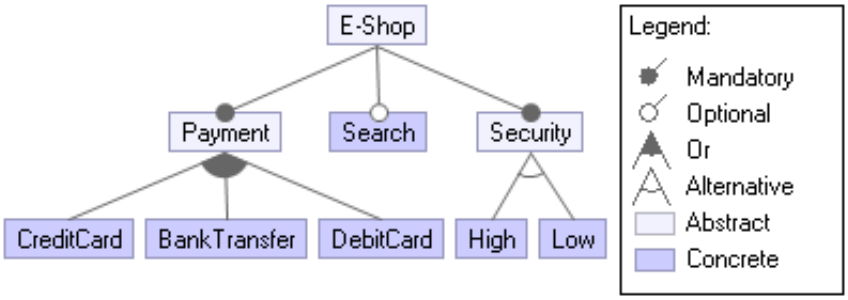

Figure 2.1: Example of a feature model

In addition to feature models, FeatureIDE supports several so-called composers, which are different ways of generating products. Antenna is one of these composers, which enables the use of annotations in Java by providing a simple Java preprocessor. With Antenna, features are defined using *if-else* statements in the code. After the feature model has been created and the corresponding code written, a configuration can be created by selecting which features the product shall include. Configuration files can be created that contain a predefined set of features which can be useful for creating base configurations of products. Antenna then composes the code and includes the code for all selected features.

2.1.3 Commonly Used Activities

This section presents some commonly used activities during the process of migrating to an SPL. These activities have been found by studying the literature on SPL migration strategies.

There are mainly two types of approaches used when it comes to performing a migration, the bottom-up approach and the top-down approach. The top-down approach is based on deriving features as a result of a domain analysis. The bottom-up approach focuses instead on deriving features as a result of code analysis techniques, such as clone detection. If someone is to perform an SPL migration on a system that they are unfamiliar with, they should probably always use a top-down approach to begin with. On the other hand, if they themselves have developed the system from scratch, there probably is no need to perform a domain analysis and they can use a bottom-up approach instead. To get a broader understanding of the features in a system, these approaches can be combined and used simultaneously by combining domain analysis and code analysis activities. Yinxing et al. calls this method the sandwich approach which can be useful where the domain knowledge is limited but not non-existent [Xue, 2011].

Whether the top-down or bottom-up approach is used, there are a number of activities commonly performed. Since features are often poorly documented, with regards to for example location, author and rationale, they need to be recovered

[Krüger et al., 2019]. This activity is called feature location and is one of the most common and most expensive activities involved in SPL migrations [Wang et al., 2013; Poshyvanyk et al., 2007; Rubin and Chechik, 2013]. This task often has to be done manually, since automated tools are often not accurate enough and need to be heavily adapted for each project. This makes the feature location process prone to errors in addition to being time-consuming [Krüger et al., 2019]. An experiment conducted by Wang et al. studied which actions are involved in the process of locating features. Among these actions, reading source code is the most common action [Wang et al., 2013]. Clone detection and diffing are two other powerful activities that aids the feature location process. Clone detection is the process of finding cloned code in source code. Diffing is the process of comparing files line by line to find similarities and differences. These processes provide a simple way of finding possible features in the cloned code.

In order to fully locate features, it is crucial to have an extensive understanding of the domain [Kang et al., 1990]. A domain analysis can consist of several activities, such as using the product, analysing UML diagrams and looking at the basic structure of the source code. In addition to providing useful data to the feature location process, the domain analysis will provide useful feature information. The goal of the domain analysis is to find out which features are relevant to have in the SPL and the result of the analysis is often documented in a feature model [Apel et al., 2013].

Analysing the requirements from the customer is often important in addition to performing the domain analysis. The purpose of this analysis is to map the customer's requirements to selections of features that were identified in the domain analysis [Assunção et al., 2017].

As mentioned earlier, features are generally maintained in a feature model [Lee et al., 2004]. When features have been defined in the feature model they have to be actually implemented, regardless if the SPL is built from scratch or code is extracted from existing products. As mentioned earlier, the features can be implemented using either annotations in the code or by using a compositional approach.

2.1.4 Cost Models

Cost models are helpful in order to predict efforts and savings when migrating to an SPL [Clements et al., 2005]. This section explains the well-established cost models SIMPLE, InCoME and COPLIMO. These cost models serve as a basis for our measurement approach, which is described in Section 3.4.

2.1.4.1 SIMPLE

The *Structured Intuitive Model for Product Line Economics* (SIMPLE) is one cost model used for SPL [Krüger et al., 2016; Clements et al., 2005]. It describes a set of functions and allows companies perform cost estimation for their SPL development.

There are four different basic cost functions in SIMPLE:

- Organizational [C_{org}] is the cost of introducing SPLE to an organization. Can for example include training and process improvement.
- Core Asset Base [C_{cab}] explains the cost of developing the core asset base. Can for example include learning different development environments, and analysis of commonality and variability.
- Unique [C_{unique}] describes the cost of implementing unique parts that are not part of the core asset base.
- Reuse [C_{reuse}] describes the cost of reusing features from the core asset base. This includes identification, integration and performing testing on assets.

These four basic cost functions can be used to describe the cost of building an SPL that contains n number of products (p). The equation of how to calculate the cost of building a product line, C_{SPL}, is shown below in Equation 2.1.

$$C_{SPL} = C_{org} + C_{cab} + \sum_{i}^{n}(C_{unique}(p_i) + C_{reuse}(p_i)) \tag{2.1}$$

With the use of this SPL cost equation, we can create another equation that shows the cost difference between using a product-line and stand-alone development. This helps to decide whether to use product-line or stand-alone development. SIMPLE defines the function C_{prod} which is the cost of developing a product without reuse. The equation for the savings, $C_{savings}$, is shown below in Equation 2.2.

$$C_{savings} = \sum_{i}^{n} C_{prod}(p_i) - C_{SPL} \tag{2.2}$$

To calculate the return of investment, ROI, the savings ($C_{savings}$) is divided with the sum of the investment cost. The investments is the organizational cost (C_{org}) and the cost of building the core asset base (C_{cab}). This calculation is shown below in Equation 2.3.

$$ROI = \frac{C_{savings}}{C_{org} + C_{cab}} \tag{2.3}$$

The cost for evolution, C_{evo}, describes the effort of releasing new versions which can include bug fixes or new functionality. The function C_{cabu} is the cost for updating the core asset base, which can occur when bug fixing or adapting to a product. The equation for calculating C_{evo} is shown below in Equation 2.4.

$$C_{evo} = \sum_{i}^{n}(C_{cabu}(p_i) + C_{unique}(p_i) + C_{reuse}(p_i)) \tag{2.4}$$

2.1.4.2 InCoME

The *Integrated Cost Model for Product Line Engineering* (InCoME) helps organizations decide if it is worth to invest in a product line, based on produced cost and benefits [Nóbrega et al., 2008]. The model consists of three layers. The cost

factors layer is the lowest level and provides a number of cost functions. The view point layer is the middle level which contains three viewpoints; product, domain and corporate engineering, which all includes a reuse scenario. The investment layer is the highest level and contains a number of economic functions. In Figure 2.2 is the structure of InCoME illustrated in a block diagram notation.

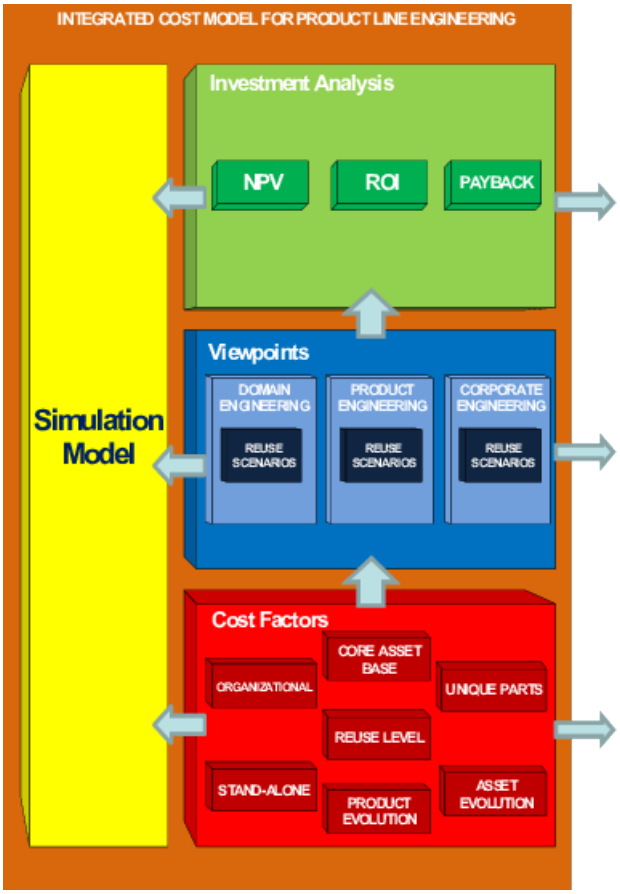

Figure 2.2: Structure of InCoME in a block diagram notation [Nóbrega et al., 2008]

The cost factors layer includes seven different cost functions that are used to provide the upper layers with cost estimations. These cost functions are the same that are used in the SIMPLE model. InCoME introduces a new function called stand-alone (C_{ind}), which is the cost of building a product independently of the product line.

The viewpoint layer uses the values that are calculated in cost factors layer. The values are used to calculate the functions for the reuse scenarios. The functions explain the cost savings that are targeted by a specific stakeholder. The viewpoints includes three different SPLE cycles; product, domain and corporate engineering cycle.

The values that are produced by all of the scenarios and built by a specific viewpoint are sent to the investment layer. This layer produces different economic calculations for the different viewpoints. The functions that are calculated in this layer are Net

Present Value (NPV), Return of Investment (ROI) and Payback Value (PV). NPV returns the value of a predicted cash flow. ROI is calculated by dividing the NPV with the investment cost. The investment cost is the sum of the organizational cost and the core asset base cost ($C_{org} + C_{cab}$). The PV function returns the smallest period of time when the NPV return a positive value.

The equation for calculating ROI is shown below in Equation 2.5. The NPV of viewpoint (v) is divided with the investment cost.

$$ROI = \frac{NPV(v)}{C_{org} + C_{cab}} \tag{2.5}$$

The equation for calculating NPV for viewpoint v is shown below in Equation 2.6. The investment cycle is denoted by Y and describes which time period the investments will be analyzed. This is often set in years and is often calculated from a start date, which is denoted SD. Discount rated (d) is a quantity that expresses the time value of money.

$$NPV(v) = \sum_{z=1}^{Y} \frac{B_v(SD + z)}{(1 + d)^z} \tag{2.6}$$

2.1.4.3 COPLIMO

The *Constructive Product Line Investment Model* (COPLIMO) is based on and extends the well-established model COCOMO II [Boehm et al., 2004]. COPLIMO consists of two different components: a cost model for the development cost of a software product line, as well as an annualized post-development life cycle model.

Cost models such as COPLIMO must cover two main sources of investments or savings, being the *Relative Cost of Writing for Reuse* (RCWR), as well as the *Relative Cost of Reuse* (RCR) [Pfleeger, 1997]. RCWR is the added cost of writing reusable code to be used in a SPL, compared to the cost of writing code to be used in a single application. RCR is the cost of actually reusing the code in a new SPL product, compared to the cost of writing new code for that product [Boehm et al., 2004].

2.1.5 Tools

There are several existing tools to aid in all stages of an SPL migration process, including tools to discover features and other tools that increase feature traceability. As mentioned earlier, FeatureIDE is an important tool to maintain feature models, as well as to configure variants and see valid configurations. Other tools include CIDE, AHEAD, FeatureHouse, FLOrIDA, BUT4Reuse [Kästner, 2007; Batory, 2006; Andam et al., 2017; Martinez et al., 2017]. CIDE originally stood for Colored Integration Development Environment, which is a tool that supports colored code annotations to make the code belonging to one feature appear with a certain background color, among other views and navigation support. AHEAD (Algebraic Hierarchical Equations for Application Design) is an architectural model for

feature-oriented programming (FOP) with focus on supporting large-scale compositional SPLs. FeatureHouse is a tool which allows composition of software artifacts, which needs to be used in compositional SPLs. FeatureHouse can be used alongside FeatureIDE. FLOrIDA (Feature LOcatIon DAshboard for Extracting and Visualizing Feature Traces) is a tool that allows increased feature location and feature traceability by providing different views such as a feature tree and other visualizations. Finally, BUT4Reuse is a tool that provides several functionalities for extracting software artefact variants, including commonality and variability analysis, feature identification and location and other features.

3
Methods

This chapter describes the approach for performing the migration. The chapter explains the research questions, the general methodology and the Android applications used in the migration. The measurement approach used for logging and the migration strategy is also presented in this chapter.

3.1 Research Questions

RQ1: *What activities should be performed when migrating cloned variants into an annotative product line?*

It is important to find out what activities that should be performed and documented when migrating to an annotative SPL. There is no predefined list of what activities that should be included in a migration. This thesis provides a number of different activities performed during the migration process together with a detailed log of each activity. This is answered by literature study as well as our experience.

RQ2: *What are the characteristics and cost of the migration activities?*

The characteristics and cost of the activities are logged with our own designed measurement approach. This logging approach helps us to keep track of each activity that was performed during the migration.

RQ3: *How does the migration to an annotative product line differ from the migration to a compositional product line?*

By comparing the results from RQ2 to the results from a compositional product line migration, the difference between these approaches can be presented. This will be done by comparing our result to the result of the other group that is performing a compositional product line migration.

3.2 General Methodology

This section provides an overview of everything that we performed during this project. The first thing we did was to conduct a study on the literature in the subject. Once we had a decent understanding of the topic in general, as well as on cost models, we designed a common measurement approach for logging activities,

in collaboration with the other thesis group. It is important to have a common measurement approach in order to make a reliable and accurate comparison of the two different approaches. Figure 3.1 shows the workflow of the general methodology.

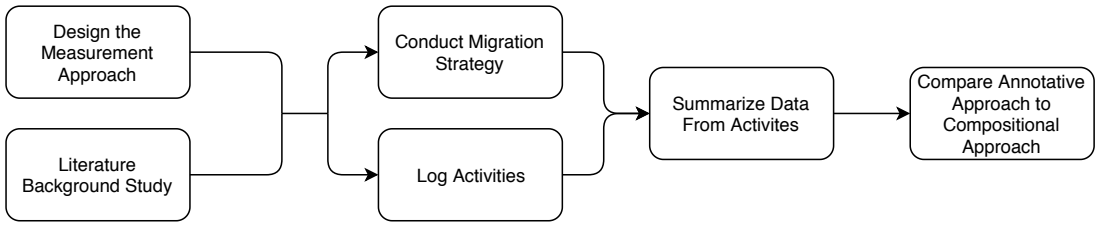

Figure 3.1: Overview of the workflow

After we had created a strategy for measuring our efforts, we created a general strategy for how to perform the actual migration. This strategy was based on the literature review, mainly the literature on commonly used activities. The purpose of this strategy was to have some sort of structure when performing the migration.

The following text briefly explains each step that we performed during the migration. To start off the actual migration, we performed a domain analysis to get a better understanding of the different games. We wanted to find what features they have in common and what features that differ. As part of the domain analysis, all games were played and class diagrams were created from the source code. Class diagrams are important to better grasp the structure of the games and to see which classes have been cloned from game to game. In addition to simply playing the games, we had to build the code to make sure that the code worked.

By manually analysing the code after performing architecture analysis, more features were discovered, in addition to their location in the code. To analyse the code further, activities such as clone detection and pairwise diffing are important to locate more features. These two activities proved to be very useful in the migration process.

Once most features had been found, a feature model was created using FeatureIDE to visualize all feature and to be able to create new configurations. Antenna was used as a composer in FeatureIDE to be able to use Java preprocessor statements to annotate features in the code. The games were migrated into a common code base by annotating the features from the feature model in the code. Once finished, the cost of the migration is measured using our activity logs and the results are then compared to the results of the composition-based approach.

During the migration process, it was crucial to continuously ensure that the quality of the products was intact. To do this, all games were built and thoroughly tested after each addition to the SPL, which makes sure that no functionality has been lost in the process. Unit testing might have been suitable for this purpose, but it was not something we wanted to spend time on, due to the already high complexity of configuring and building the games.

3.3 Subject Systems

The Apo-Games includes several different Android games. Five of these games are included in this report and are part of the Apo-Games case study [Krüger et al., 2018], with the source code being available on BitBucket[1]. The games are written in Java and range from around 3500 LOC to 8000 LOC in size. The games were developed from 2012 to 2014 and uses different versions of BitsEngine created by Marc Wiedenhoeft. Table 3.1 lists the LOC and number of files in each game.

Game	**LOC**	**Files**
ApoDice	3479	19
ApoSnake	3975	19
ApoClock	4944	28
ApoMono	7854	24
MyTreasure	7156	27
Total	27408	117

Table 3.1: LOC and number of files in each of the games and the total LOC and number of files

ApoClock consists of two different game modes, arcade and puzzle. The mission in the arcade mode is to survive as long as possible by sending a ball, which is spawned at the start of the game, between different clocks before the clock timeouts. The game continues until you either miss a clock with the ball or a clock timeouts before you send the ball to another clock. In arcade mode you get a score at the end of the round based on how long you managed to stay alive.

The mission in the puzzle mode is to shoot the ball to all the clocks in a level to beat it. A clock disappears when the ball leaves that specific clock and is shot to another clock. As in arcade mode, you fail the level if the ball misses a clock or the clock timeouts before the ball is sent to another clock.

ApoDice is a puzzle game in which there are several dice with different numbers on them, indicating how many times they can be moved. The goal is to move each dice to a black square on the board. If you run out of turns, you need to restart the level. There are 30 levels to choose from, excluding user-created levels.

ApoMono is a puzzle game where the goal is to move an avatar from the starting position to the goal. There are several different obstacles that need to be passed in order to reach the goal, such as high walls and missing floor. The player is allowed to move specific stones in order to surpass these different obstacles and reach the goal.

ApoSnake is a game mildly similar to the classic snake game, in which the player controls a snake and eats the candy placed on the board. Whenever candy is eaten,

[1] ApoGames source code: https://bitbucket.org/Jacob_Krueger/apogamessrc/src/master/

the length of the snake is increased. The candies are colored differently and the snake becomes the color of a candy when eating it. The snake can move through walls that are colored differently from itself.

MyTreasure is another puzzle game where the character is placed within a 2D maze. Inside the maze there is a golden coin which the player shall reach. This is done by rotating the maze so that the character moves in the maze using gravity. There are also yellow blocks that are moving when rotating the maze.

All of the games have an editor, where you have the ability to create your own custom level of that specific game. The custom level is then stored among all other levels created by other users, and all these levels can be retrieved and played from the userlevels button in the menu. Note that this button crashes the game in all games except ApoMono and MyTreasure due to a bug. An ingame overview screenshot of each of the games can be found in Appendix B to get a more clear understanding of the different games.

3.4 Design of the Measurement Approach

The measurement approach was designed in collaboration with the other thesis group by having three joint meetings and discussing how to log the activities. The approach explains in detail how the logging of activities was done. It explains what qualitative and quantitative information that will be recorded and decide how it should be documented.

In order to estimate the reengineering efforts, the migration process must be measured. The measurement approach is defined for how to thoroughly log everything. The metrics that are used in the log is based on the cost models SIMPLE, InCoME and COPLIMO, and their respective metrics which are described in Section 2.1.4. More specifically, the metrics used in the measurement approach are based on the different metrics from the cost models.

As aforementioned, we designed the measurement approach in collaboration with the other thesis group, so that a conclusion could be drawn based on the comparison of the results. The actual strategy will be different, which means that activities involved will differ as well. To circumvent this, we compared the effort that was spent on each activity type, rather than the activities themselves.

A log entry contains information about the activity. Each activity has a unique ID and a name in order to be able to identify it. An activity is also assigned to one or several activity types, which enabled us to measure what efforts were spent in different parts of the migration. These types were created in collaboration with the other thesis group and our supervisor, since we wanted to have common types of activities when comparing the cost of the two different SPL approaches. The types are defined and explained below.

- **SPLE training** - Any activity that involves researching specific literature of SPLE, including different approaches of SPLE, such as strategies to apply annotative or compositional approach to transform an existing software system into an SPL. Also includes learning different software programs that are used to be used for the product line migration.
- **Domain analysis** - Identifying commonalities within variants and map it to the domain level.
- **Feature identification** - Identifying functionality that could be classed as a feature, i.e. Enemy.
- **Diffing** - Activity that revolves around finding differences in implementation between clones.
- **Architecture identification** - Any activity that revolves around identifying the architecture, i.e creating class diagrams.
- **Feature location** - Activities that relate to locating identified features in the code.
- **Feature modeling** - Mapping all identified features into a feature model.
- **Transformation** - Any activity that has to do with code modification, for example, annotating features in the code.
- **Quality assurance** - Activities such as running and testing games and game-functionalities after each iteration are classified as quality assurance activities.

There is a date for when the activity was started and a date for when it was ended. There is also a description of the activity, to get further understanding and information about the activity, and also understand why it was performed.

COPLIMO and InCoME estimates effort in terms of person month [Boehm et al., 2004; Nóbrega et al., 2008]. The logging of activities will therefore be made each person day at least, due to our limited time. 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 is incremented and other fields are filled in when appropriate. This logging of duration is also based on InCoME where it takes into consideration the start and end date of the investment, as well as the value of time and money, in its calculation of SPL migration costs [Nóbrega et al., 2008].

Other data in a log entry consists of number of LOC and files that have been added, removed or edited. This is based on COPLIMO that uses parameters such as percent of code that has been modified and portion of software that must be modified to work well [Boehm et al., 2004]. Metrics such as number of LOC can help in trying to describe how large an activity is, as well as its complexity.

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

Lastly, each activity should have a description. In the description, qualitative data about the complexity and importance of the activity is documented. In addition, any dependencies to other activities are noted here. The final template for logging activities along with an example is displayed below in Table 3.2.

INFORMATION
- **Activity**:
- **Activity ID**:
- **Activity type**:
- **Start date**:
- **End date**:
- **Description**:

DATA
- **Person hours spent**:
- **Number of commits**:
- **LOC added**:
- **LOC removed**:
- **LOC modified**:
- **Number of files added**:
- **Number of files removed**:
- **Number of files modified**:

ARTIFACTS
- **Input**:
- **Output**:
- **Tools used**:

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

Table 3.2: The logging template for the migration strategy activities

The logging template is divided into four types of data that are collected. The information section contains metadata about the activity such as activity ID that can be used to reference activities while studying their dependencies. It contains a data section where we input quantitative data such as total hours spent and number of LOC added. This section may differ from activity to activity as some activities may not have any LOC modified. The artifacts type entails the input and output of the activity, such as source code as input and feature model as an output, while tools used could be FeatureIDE. Lastly, we provide the complexity and importance of the activity along with its dependencies on other activities under the activity description section.

Table 3.3 lists different cost models factors that was used to define the measurements included in our measurement approach.

Cost Model	Cost Model Factor	Measurement
SIMPLE	CSWdev - full cost of developing SPL	Hours spent
InCoME	SD - start date	Start/End date
InCoME	Y - investment cycle	Start/End date
InCoME	PM - person month	Logging every day
COPLIMO	PSIZE - effort and SPL development cost	Nr of Commits, Hours spent
COPLIMO	SU - understandability of software	Hours spent
COPLIMO	CM - percent of code modified	LOC and Files added, removed and modified
COPLIMO	DM - percent of design modified	LOC and Files added, removed and modified

Table 3.3: Table of cost model factors from the cost models linked together with the measurements from our measurement approach

3.5 Migration Strategy

In this section, we outline our strategy for migrating the variants to an SPL. This strategy is based on the commonly used activities described in Section 2.1.3.

3.5.1 Domain Analysis

A domain analysis is a crucial part of an SPL migration, especially when there is no previous knowledge about the products, since it provides an overview of how the products work and how they are structured [Kang et al., 1990].

The domain analysis was important in the early stage of the migration since each of the domain analysis activities increased our understanding of the games and their domain. A better view and understanding of the domain helped us perform the other activities in the migration, since an understanding of the domain often is crucial in order to perform an SPL migration.

3.5.1.1 Running and Testing the Games

As part of the domain analysis, each game was installed from Google Play and played by both group members on an Android smartphone. This was done to gain a first understanding of the functionalities of the games and to find some initial features. The features that were found were documented and can be found in Section 4.1.1.

3.5.1.2 Building the Source Code

In addition to simply playing the games, an important step is to actually build the source code to make sure that the games even work. Knowing how to build the

source code enabled us to verify the quality of the SPL during the process, as well as after. We first tried to build the games in Eclipse, but after struggling with Android development in Eclipse, we switched to Android Studio, and played the games on an Android emulator. This proved to be a very time consuming task, since we had very little previous experience in running old projects in Android Studio and Eclipse. It was also quite tedious because all games uses a different version of the engine that they run on, since they use different features. This results in us having to change the library (.jar) files each time a different game is built.

3.5.1.3 Architecture Analysis

An important part of the domain analysis is to study the architecture of the games. Using IntelliJ, we created a Java class diagram for each of the five games. The class diagrams provide an overview of the structure of the games as well as how different parts interact. As a complementary activity to the architecture analysis, we performed a simple code analysis where we manually analysed the classes found in the class diagrams. The code analysis provided further understanding of the structure and architecture of the code. We studied the code for each game and paid extra attention to classes that were unique to a certain game. This activity gave us a good grasp of how the games had been implemented.

3.5.2 Locating Features

From the domain analysis we gained a broad understanding of the features of the games and their structure. To gain a more intricate understanding, we had to go deeper and analyze the code further by performing clone detection and pairwise diffing between the games to find cloned code and features.

3.5.2.1 Clone Detection and Diffing

In addition to manually analysing code, there are several tools that can be used to find cloned code. Both clone detection and diffing were used to identify and locate features in the cloned code. Clone detection was done using CPD[2] (Copy/Paste Detector), which is a simple command-line tool to perform clone detection. We selected the whole Apo-Games folder with all games was selected to showcase cloned code between the games. We only did this once and documented where we found much cloned code, in addition to describing what the code implemented. We also documented which games had the most similarities to each other, to help with the transformation process.

Diffing was performed in a well-established tool called Meld[3]. Meld enabled us to compare files with the same name, line by line, which makes it easy to spot variations. Since Meld and similar tools compares file by file, we were forced to rename most files in our source code so that Meld could compare them. An example is ApoSnakeModel.java and ApoDiceModel.java which both have to be renamed to

[2]Finding duplicated code with CPD: https://pmd.github.io/latest/pmd_userdocs_cpd.html
[3]Meld: http://meldmerge.org/

the same thing, such as ApoGameModel.java. We compared every game's files to all other games' files and documented each case of similarity. We checked for cases where all the code was similar, as well as cases where the code was dissimilar but it contained the same methods and overall structure. We decided to include these cases and annotate the differences in child features of the main feature. This process made made it much more clear to us which games use similar assets and which ones that do not.

In Figure 3.2 a snapshot can be seen showing a comparison between the menu class files in ApoSnake and ApoDice in Meld. The code of ApoSnake is shown in the upper half and the code of ApoDice is shown in the lower half. Code marked with blue means that the code is nearly the same on these lines in both files, and the difference between them is marked with a darker blue color. Code marked with green implies that this line of code is missing in the other file. White, unmarked code means that it is exactly the same code in both files.

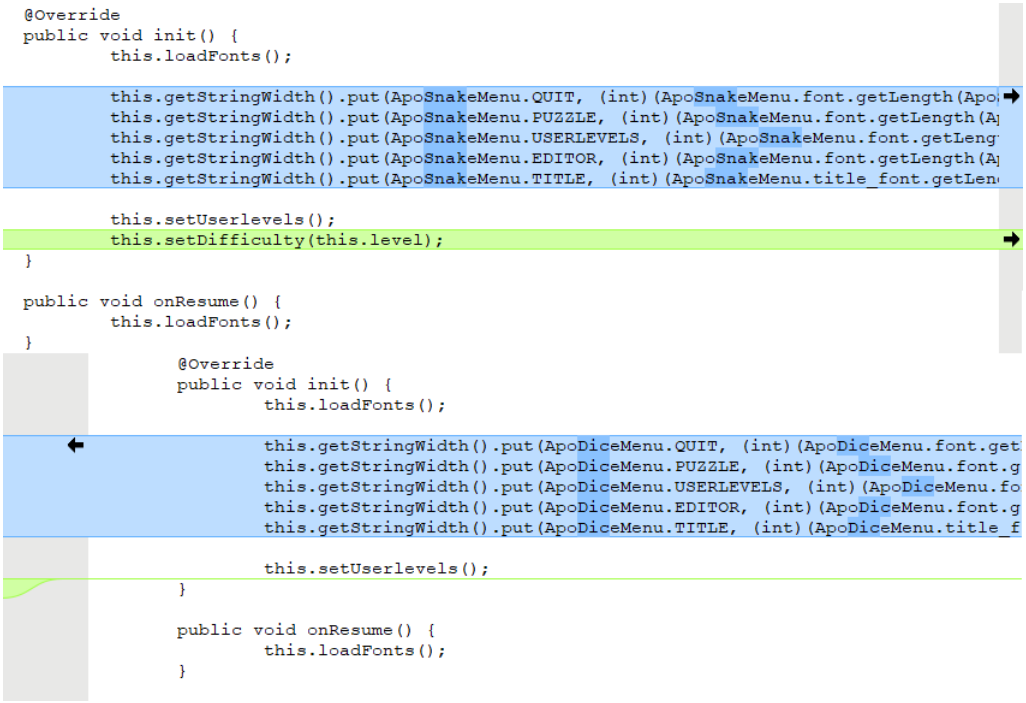

Figure 3.2: Comparison of ApoSnake and ApoDice in Meld

3.5.3 Feature Modeling and Antenna

The FeatureIDE plugin available in Eclipse was used to create a feature model. We used Antenna as our composer in FeatureIDE to be able to create annotations in the code by using #if-else directives.

To get familiar with how Antenna uses preprocessor directives to compose code, we performed a straightforward migration of ApoDice and ApoSnake into a common codebase with some feature annotations. We mainly annotated the logic algorithms and the menus of the games, so that one of the games could be chosen with one of

the menus. Since much of the code is similar for these games, they were easy to integrate into a common codebase. Since some parts differed a lot, such as game logic, we did not spend time annotating extracting code from there. The code snippet below explains how Antenna uses the #if directive to include and exclude code depending on feature configurations, in this case the Dice feature.

```
//#if Dice
<Dice code here>

//#elif Snake
//@<Snake code here>

//#endif
```

Figure 3.3: Antenna feature example

Once we were familiar with the tools, the other features from the feature model were to be annotated. To do this, we had to go back and look at the results from previous activities, which served as a detailed summary of all features identified so far.

3.5.4 Transformation

The overall goal of the transformation was to have all the games in a common code base and to compile and run them. To do this, the transformation was performed in several activities. The migration started with combining ApoDice and ApoSnake into a common codebase using Meld and FeatureIDE. The goal of the initial migration was simply to get the games to run, so we did not focus on annotating many features in this iteration. Once we were able to run both the games and configure the menus, we started annotating more features in the code. These were features such as the controls of the games and the option to enable or disable certain parts of the games, e.g. the level editor. We created configuration files so that we could quickly select all features of a game to build and run it.

When we were mostly finished with the migration of these two games, it was time to migrate another game into the platform. ApoClock was the game that was most similar to the already migrated games, so it was the next game to be integrated. Worth noting is that even though the games look very similar, ApoClock uses a different rendering engine compared to the other two in the SPL, which complicated the migration process. This was evident by many methods being different in the games. Meld was used once again, to compare the migrated code base with the code of ApoClock, and the initial goal was simply to get all the games to run from the common code base. When ApoClock could be compiled and run from the SPL, we started annotating more features in the code, similarly to the first migration.

At this point, three out of five games had been migrated into the SPL. ApoMono and MyTreasure, the games that were least similar to the other ones, were remain-

ing. We chose to start with ApoMono, and migrated it into the SPL, using the same method as before. When that iteration was finished, we migrated MyTreasure, the final game, into the SPL. These two games could not be integrated as cleanly as the other games could, due to the fact that they were quite different from the other games. However, they share some similarities to each other, which meant that some redundancy could be reduced.

3.5.5 Quality Assurance

During the migration of the games, a quality assurance process had to be undertaken to maintain the quality of the games. The goal of this process is to make sure that each of the games will still compile and run without any loss of functionality. This can be done by using configuration files in FeatureIDE, in which the features of a certain product are predefined. We created configurations for all games which configured the base games, so that we easily could switch between the games. We tested more configurations than the standard one, for all games. After each major iteration of the transformation process, such as when a new game was integrated in the SPL, or when a number of features have been annotated, the base configuration of each game was selected so that they could be tested to maintain the quality of the games.

3.5.6 Branding

To perform branding, certain parts of the games are supposed to be made more similar, to make it obvious to a user that the games belong to the same brand. We created a separate activity for branding, since we wanted to draw a conclusion of its value to the transformation.

The branding was supposed to mainly be performed in the menus of the games. The menus of ApoDice, ApoSnake and ApoClock were very similar from the start, which meant that these menus could be replaced by one universal menu. This makes it clear for a user that the games have something in common, and reduces redundancy in the code. Other branding that we performed was to statically include font selection in the games, so that all games can use any font from the other games.

We wanted to perform more substantial branding, specifically we wanted to enable an options menu in all games, where for example music could be dynamically turned on or off. We spent a whole lot of time on this process and finally decided to not continue, as we struggled with the many differences in the implementation of the games.

4
Results

This chapter is divided into three different sections; the activities performed during the migration, the cost and characteristics of the migration activities and lastly the comparison of the cost between our annotative approach and the compositional approach. Each section relates to one of the research questions presented in Section 3.1.

4.1 Activities

This section relates to research question RQ1 - *What activities should be performed when migrating cloned variants into annotative product line?* As described in Section 3.5, several activities were performed to migrate the clones into an SPL. It has been described how the activities were performed and why they should be a part of the migration, while this section explains the result of each activity and how it contributed to the SPL migration. The activities that were performed during the migration are listed below, with activity name, activity ID and reference to the log in Appendix A.

- A.1 - Running and testing the games
- A.2 - Architecture analysis
- A.3 - Code analysis
- A.4 - Building and running games in Android Studio
- A.5 - Installing and learning FeatureIDE
- A.6 - Code diffing
- A.7 - Transformation: ApoSnake and ApoDice
- A.8 - Feature identification
- A.9 - Transformation: ApoSnake, ApoDice and ApoClock
- A.10 - Transformation: ApoSnake, ApoDice, ApoClock and ApoMono
- A.11 - Font
- A.12 - Transformation: All games
- A.13 - Branding - Menu
- A.14 - General SPLE learning
- A.15 - Annotating features
- A.16 - Quality assurance

List 4.1: List of performed activities

4.1.1 Running and Testing the Games

While playing the games, we found many similarities between them and it was obvious that assets had been copied from game to game. The goal of this activity was to discover an initial set of features that were recognizable from playing the games. In other words, we wanted to perform feature identification using a top-down approach. This worked well and we found features in all games which are described and listed below.

In particular, we found that ApoClock, ApoDice and ApoSnake were similar to each other, in addition to ApoMono and MyTreasure being similar to each other. The most striking similarities were found in the menus, where most of the games had the same look and feel, with only a color and style change difference. Another similar feature is the Editor option where the player can create their own level of the game and add it to the levels that are created by players. The editor itself differs between the games since the elements used in the games are different. All of the levels created by players can be found in another similar feature called Userlevels. The user has the possibility to play all levels created by other players. The games have different control systems. ApoClock and MyTreasure are controlled by pressing, while ApoDice is controlled by swiping. ApoSnake and ApoMono have special move buttons for navigating. The games have different score systems. ApoClock's puzzle mode, ApoSnake and ApoDice and ApoMono are similar in that they have a binary score system, in which the user either completes a level or not. ApoClock's arcade mode has a score counter that increases as the user survives. MyTreasure is more unique system, where the user can get gold, silver or bronze on levels, depending on how many moves they finished the level in. The first bullet list is a list of features that are common to some games, and the other bullet list shows features that are specific to certain games.

All games
- Menu
- Editor
- Level chooser menu
- Userlevels
- Theme for menu and level chooser menu
- Controls for character
- Scoring

ApoClock - ApoDice - ApoSnake
- Same menu style
- Same grey canvas
- Editor
- Options - specific for ApoClock

ApoMono - MyTreasure
- Sound
- Music

MyTreasure
- Language option - specific for ApoMono

List 4.2: List of features we found by playing the games

4.1.2 Architecture Analysis

The goal of this activity was mainly to begin a feature location process, to see if we could locate some of the features that we had already identified. In addition, we wanted to gain a better understanding of how each game was structured. We feel that this was successful overall but we were a bit misled by the similarities of the class diagrams.

From the class diagrams, it is clear that the games have been developed using a clone-and-own strategy, which is true according to the Apo-Games case study [Krüger et al., 2018]. Analysing them enabled us to easier see where certain features are located and how they have been implemented. All the games have a model which is an abstract class that contains most aspects of the games, including menus and the actual game logic itself. The classes that are not connected to the model are things such as buttons and other entities that are game specific and used by the model classes.

As part of the code analysis performed along with the architecture analysis, it is evident that the developer focused on reusing code as much as possible. An example of this is that each game has buttons with a special style, but all buttons are created using the general ApoButton class. Furthermore, ApoButton is an extension of ApoEntity that provides basic functions for buttons.

The class diagram of ApoDice is shown below in Figure 4.1 as an example. The abstract class ApoDiceModel has four subclasses connected to it. As mentioned above, the number of classes connected to the model varies between the games, but

all games have menu, game logic and editor connected to it.

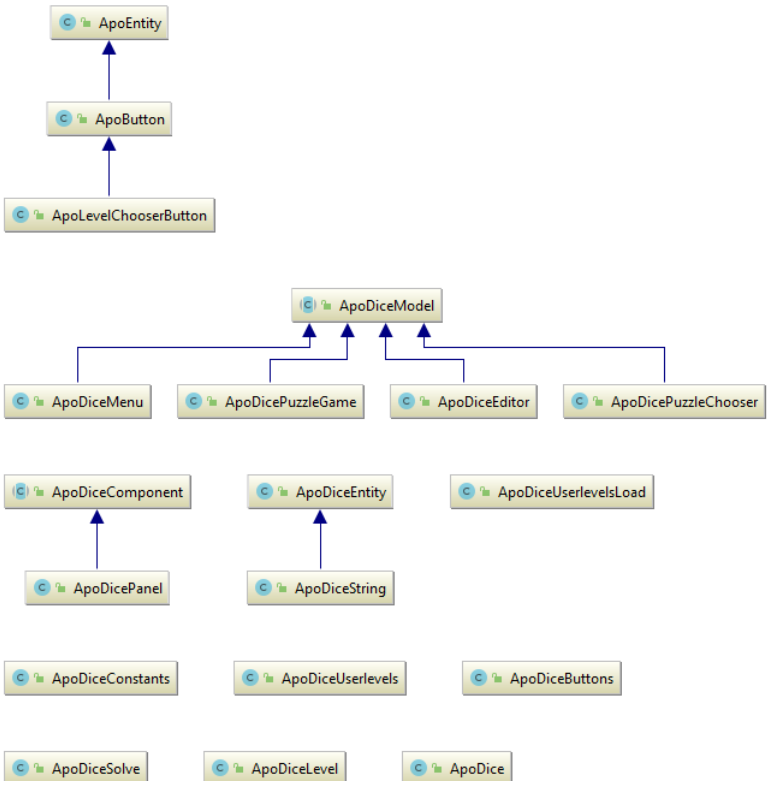

Figure 4.1: Class diagram of ApoDice

4.1.3 Building the Games

All of the games were built in Android Studio and played on an Android emulator, except for ApoSnake that could not be run for some unknown reason. We spent several hours trying to find and solve the issue, but without success. ApoSnake could never be tested in our SPL for this reason. An attempt was made to contact the developer of Apo-Games to solve the issue, without any response.

4.1.4 Clone Detection and Diffing

Clone detection proved to be one of the more important bottom-up activities, since we got detailed information about exactly which code had been cloned between the games. Perhaps most importantly, it provided us information about where features are located in the code. This activity was the starting point of the first transformation we performed, since we were able to clearly see which games were the most similar and therefore the easiest to integrate. Since the games mostly share the same structure, it was easy to compare one class in one game to the corresponding class in another game.

4.1.5 Feature Modeling

The feature model can be seen as the result of all previous activities in which features were identified. It was defined based on the discovered features that we have listed in the previous activities. In total, we identified 59 features, including abstract features that are not annotated in the code. Excluding the abstract features, there are 42 implementable features.

The feature model contains features from all the games with each of the possible variations to a feature. There are a few mandatory features in the feature model. These are GameLogic, Menu (Theme), LevelGrid and Font. The reason for this is that these will enable the core features of a game. The GameLogic feature is the most robust feature, which contains most of the intricate algorithms that define the games. Other features include the Editor, Userlevels and Options, which gives the ability to remove these completely from the games. The Controls feature has three different types of controls that are used in the games. The Music and Sound features below Options can enable and disable these in the games that support it, i.e. ApoMono and MyTreasure. There are some constraints defined in the feature model. These are created to prevent configurations that will not work. An example is that if the TreasureGameLogic feature is selected, the TreasureMenu needs to be selected as well. Some features are hidden in the feature model, as shown by the yellow triangle. These are features that we did not annotate, but we decided to keep them in the feature model. All features are described in more detail in Appendix C. The feature model can be found in Figures 4.2, 4.3 and 4.4. All features included in each game are listed below.

ApoClock

- ApoMenu
- Editor
- Userlevels
- LevelGrid
- NextLevel
- Clock
- Grey
- Press
- Counter
- Nickname
- ClockLogic
- ApoFont

List 4.3: Features included in ApoClock

ApoDice
- Editor
- Userlevels
- ApoMenu
- LevelGrid
- NextLevel
- Dice
- Grey
- Drag
- Moves
- DiceLogic
- ApoFont

List 4.4: Features included in ApoDice

ApoMono
- Editor
- Userlevels
- MonoMenu
- LevelGrid
- NextLevel
- MonoAvatar
- MonoObjects
- MonoCanvas
- MoveButtons
- MonoEffect
- MonoMusic
- German
- English
- MonoLogic
- MonoFont
- White
- Green

List 4.5: Features included in ApoMono

ApoSnake
- Editor
- Userlevels
- ApoMenu
- LevelGrid
- NextLevel
- SnakeAvatar
- Grey
- MoveButtons
- Moves
- SnakeLogic
- ApoFont

List 4.6: Features included in ApoSnake

MyTreasure
- Editor
- Userlevels
- TreasureMenu
- LevelGrid
- NextLevel
- TreasureAvatar
- TreasureBox
- TreasureKey
- MonoCanvas
- Press
- Moves
- TreasureEffect
- TreasureMusic
- TreasureLogic
- TreasureFont

List 4.7: Features included in MyTreasure

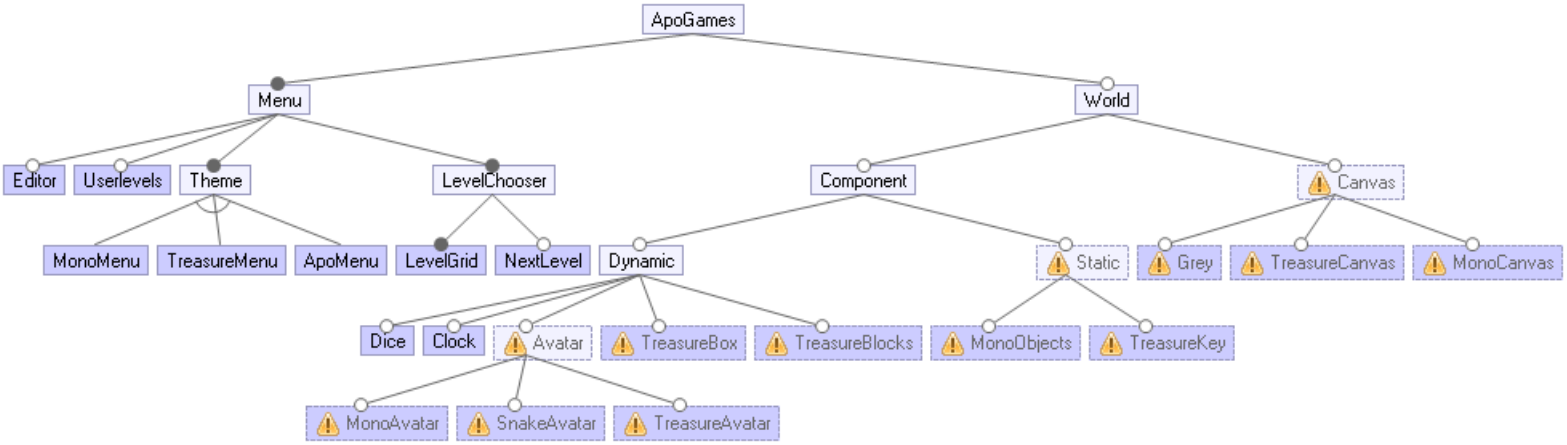

Figure 4.2: First part of feature model of Apo-Games

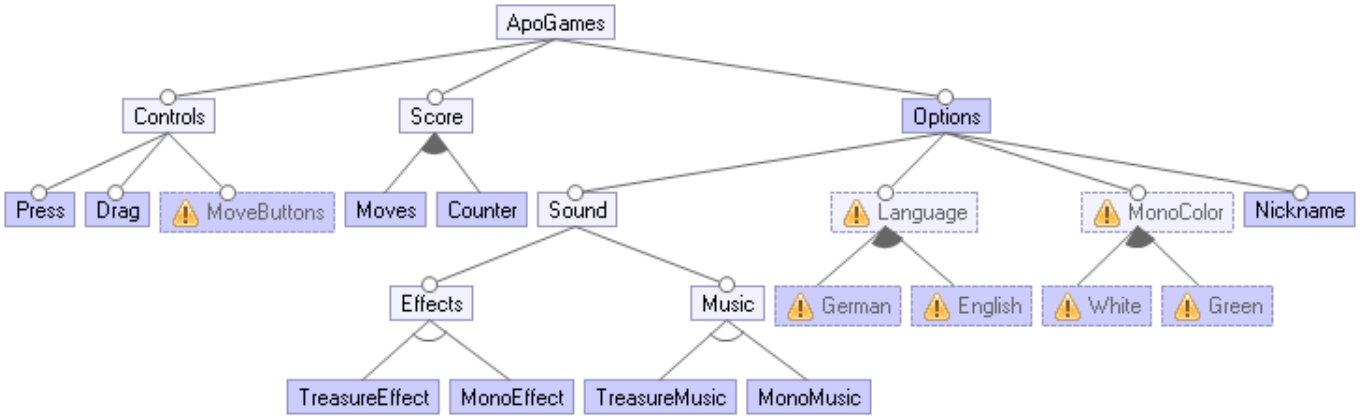

Figure 4.3: Second part of feature model of Apo-Games

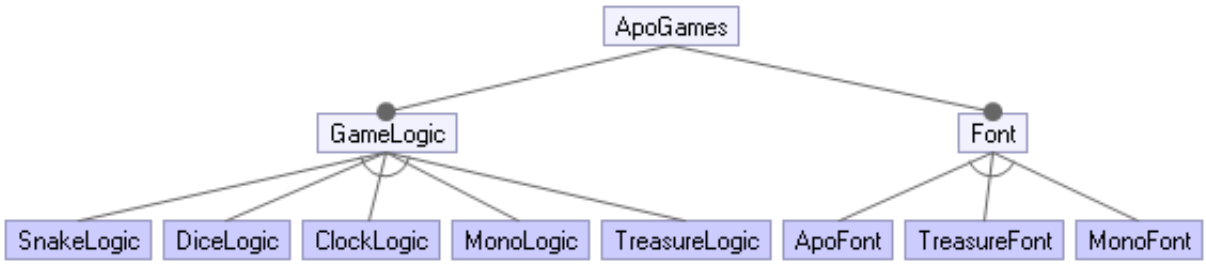

Figure 4.4: Third part of feature model of Apo-Games

4.1.6 Transformation

The first transformation where we integrated ApoDice and ApoSnake reduced the total LOC by 30.3% compared to the regular two games. We were able to successfully build both games in Android Studios and ApoDice could be played as before the transformation.

The second transformation where we integrated ApoDice, ApoSnake and Apoclock reduced the total LOC by 43.9% compared to the regular three games. All of the games were successfully built in Android Studio and could be run and played as before the transformation.

The third transformation reduced the total LOC by 38.4% compared to the regular code base of ApoDice, ApoSnake, ApoClock and ApoMono. As in the second transformation, all games were successfully built and playable as before the transformation.

The last transformation, which resulted in a common code base for all of the games, reduced the total LOC by 32.7%. There was no problem with building, playing and running the games after the final transformation. As mentioned, the total LOC were reduced by 32.7% and the number of files were reduced by 74 files, down from 117 files to 43 files. The difference in LOC and number of files is showcased in Table 4.1.

Continuously throughout the transformation process, we annotated features from the feature model into the SPL. In total, we annotated 27 features out of the 42 in total.

The quality assurance process was performed after each iteration during the transformation. It was a crucial activity to perform since it made sure that all games in the SPL were in a playable state at all times.

Game	**LOC**	**Files**
Before (All games)	27408	117
After (Common code base, ApoHybrid)	18438	43
Difference	*8970*	*74*

Table 4.1: LOC and number of files before and after the transformation and the difference between before and after the transformation

4.1.7 Summary of Activities

To answer RQ1, the following list summarizes the activities that are needed to perform an SPL migration:

- Studying the subject
- Domain analysis to understand the domain and find initial features
  - Using products
  - Architecture analysis
- Clone detection and diffing to find more features and their location
- Manual code analysis to locate features in code
- Creating a feature model
- Annotating the features as part of transformation
- Quality assurance

List 4.8: Activities needed to perform an SPL migration

4.2 Characteristics and Costs of Activities

This section relates to RQ2 - *What are the characteristics and costs of the migration activities?* The total cost of the migration is calculated by summarizing the amount of person hours (ph) spent on each activity. By summarizing activity A1 to A16, we get the total cost of 495ph. The total amount of person hours spent on each activity type are listed below, together with all the activities included in each activity type. Note that if an activity has several activity types, the time spent on that activity is divided among the activity types.

The characteristics of the activities are basically a summary of the qualitative data from the activity logs. General SPLE learning was a time-consuming but very important activity. Reading was not the hard part, but rather finding relevant literature on the subject. Running and testing the games was easy, fun and valuable for us, but it did take some time to try them thoroughly and to identify potential features.

The architecture analysis was easy to perform, but it kind of tricked us into thinking the games were more similar than they actually were, so we did not get too much value from it except for general knowledge about the structure of the games' code. The code analysis we performed along with the architecture analysis was challenging but very valuable. We analysed the classes we identified in the class diagrams and learned more about how the games were built.

Building the games in Android Studio was at first very challenging, until we learned how to configure the games correctly and how to include the BitsEngine library. This activity was very important for us since we realized that ApoSnake was not working at all. We wasted much time during this activity by trying to configure Android development in Eclipse. Installing and learning FeatureIDE took some time before we realized we needed an older version of Eclipse. Learning FeatureIDE was crucial so that we could create a feature model and configure the games.

- **SPLE training**: 90ph
  - A5 - Installing and learning FeatureIDE. 10ph
  - A14 - General SPLE learning. 80ph
- **Domain analysis**: 82ph
  - A1 - Running and testing the games. 15ph
  - A2 - Architecture analysis. 5ph
  - A3 - Manually analysing code. 15ph
  - A4 - Building and running games in Android Studio. 40ph
  - A8 - Feature identification. 7ph
- **Feature identification**: 22ph
  - A1 - Running and testing the games. 15ph
  - A8 - Feature identification. 7ph
- **Diffing**: 40ph
  - A6 - Code diffing. 40ph
- **Architecture identification**: 5ph
  - A2 - Architecture analysis. 5ph
- **Feature location**: 7ph
  - A8 - Feature identification. 7ph
- **Feature modeling**: 10ph
  - A5 - Installing and learning FeatureIDE. 10ph
- **Transformation**: 210ph
  - A7 - Transformation: ApoSnake and ApoDice. 25ph
  - A9 - Transformation: ApoSnake, ApoDice and ApoClock. 15ph
  - A10 - Transformation: ApoSnake, ApoDice, ApoClock and ApoMono. 20ph
  - A11 - Branding: Font. 15ph
  - A12 - Transformation: All games. 25ph
  - A13 - Branding: Menu. 40ph
  - A15 - Annotating features. 40ph
  - A16 - Quality assurance. 30ph
- **Quality assurance**: 30ph
  - A16 - Quality assurance. 30ph

List 4.9: List of all the activities and their cost in terms of person hours

Code diffing was one of the most important activities we performed, since it helped us perform the actual migration. It was time-consuming to compare all games to each other, but very necessary. Since the tools provided a straightforward interface, they were not particularly hard to use. Many of the files had varying complexity, since some files differed a lot from game to game.

The transformations were all quite challenging to perform, though the difficulty varied from game to game. The first transformation was super easy compared to the last ones, due to the similarities between ApoDice and ApoSnake. To make the migration process easier, we needed to synchronize the code in Android Studio and

Eclipse, a task which was not easy.

The branding activities were really challenging overall, except for the font branding. The code for loading fonts was not too complex and there were only small differences in how they were loaded in each game. The menu was hard and time-consuming to perform, which is why we settled on combining the menu of only three games: ApoDice, ApoSnake and ApoClock. It was hard because of the big difference between the standard menu and the menus used in ApoMono and MyTreasure. These two menus had a complete different look to the other three and used a different version of the BitsEngine library. The options menu was really hard to integrate into the games that did not have it for the same reasons, which is why we decided to not proceed with this task.

Annotating features is a crucial activity since the features form the basis of an SPL. The activity was complex because the code that belongs to a feature was often scattered in many places in the code. It was a time consuming activity for many reasons, including the issue of annotation placement in a file. Another thing that made it more time consuming was the messy and non-structured code we created.

The quality assurance activity was not overly complex, but it was very time consuming since we usually broke something in the code each time a transformation was undertaken or after a few features were annotated. Building each game and playing them to make sure everything worked as it should is crucial for this reason.

4.3 Comparison of Annotative and Compositional SPL

This section relates to RQ3 - *How does the migration to an annotative product line differ from the migration to a compositional product line?* Table 4.2 shows the cost of the activities types for both the annotative approach and the compositional approach. The total cost of each approach is shown at the bottom row of the table. Note that if an activity has several activity types, the time spent on that activity is equally divided among the activity types. The differences between the approaches are discussed in Section 5.1.3.

Activity type	**Annotative (ph)**	**Compositional (ph)**
SPLE training	90	16
Data cleansing	-	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 (ph):	**495**	**371,5**

Table 4.2: Cost of annotative and compositional approach

Table 4.3 shows which activities that were performed in each activity type, both for the annotative approach and the compositional approach.

Activity types	Annotative	Compositional
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	-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 4.3: Activities performed in each activity type for both the annotative approach and the compostional approach

5
Discussion

This chapter discusses the research questions, challenges we faced during the thesis, different threats to validity and limitations.

5.1 Research Questions

Each research question is discussed in its own section. The discussion focuses on what was good and bad, easy and difficult and also what we learned from this. We also discuss what could be improved and the reason of the outcome.

5.1.1 RQ1 - Activities

The bullet list in Section 4.1.7 describes a summary of the activities that we feel are necessary to perform an annotative SPL migration. Overall, most activities worked well and were beneficial to the migration, and we feel that our strategy of transforming one game at a time into the SPL was successful. Our process resulted in a simple SPL with sufficient functionality. An alternative process could have been to locate a feature in all games, then annotate it directly based on all games and include it into the SPL. We decided not to pursue this process as we thought it would be too complex with our experience and considering our time frame.

Our process would have worked even better if the games were more similar to each other. We learned during the process that annotating features has to be done carefully in order to not make the code a mess. During our initial transformation where we focused mainly on integrating all games into the SPL, we found that the code got messy quickly. This was something that we tackled continuously during the process, trying to increase readability and structure of the code.

In our case, the architecture analysis was probably the least beneficial out of the listed activities, because it made us think that the games would be more similar than they were. In addition, we never really analysed the class diagrams more than once because we did not feel the need. The branding activities we performed were not completely necessary to the actual migration, and we felt that we spent too much time on them. In hindsight, we should have just focused on integrating the three similar menus and then stopping. Integrating the three menus into the same one made sense for a transformation as we managed to reduce redundancy by a lot. However, the time we spent on other could have been spent on annotating other fea-

tures. Another activity that we spent an unnecessary amount of time on was when we tried building the games, since we first tried to get them working in Eclipse. The only things that could have helped this activity would be to have more previous experience in either Android Studio or Eclipse, or getting help from the developer of the games. Something else that we found challenging, mainly in the beginning of the project, was that we had a hard time understanding exactly what a feature is. This made the feature identification process challenging, but we found that we got better at it during the migration.

The activity that was of most use during the migration was the code diffing, since it enabled us to see differences in the code line for line. It was a time consuming activity since we had to go through a lot of different files. Learning how to use FeatureIDE was very important for the overall success of the migration. We created the feature model and used the preprocessor, Antenna, in FeatureIDE which both were important for the migration. The basics of the tools were easy to learn and created a lot of value for the migration. Another activity that contributed a lot to the success of the thesis was the background study on the SPLE field. This was a crucial part in the beginning of the thesis in order to get a good understanding of the subject and to help us get started with the process.

5.1.2 RQ2 - Cost and Characteristics of Activities

We spent a lot of time studying the SPLE literature, which was time we could have spent on other activities if we had more experience. This stands true for all activities, since we had very little experience using most of the tools and techniques that we applied. SPL migrations are often performed by people who have been a part of the development of the products. If we had developed these games, we would have immensely more knowledge on the features in these games, in addition to having great knowledge of the code. We would have more domain knowledge which would have reduced the time spent on the domain analysis.

We found that it was very hard to classify everything we did as an activity. An example of this is the feature location activity type, which only has 7 hours. This is not an easy task to perform and we definitely spent more than 7 hours in total on it. The issue is that we performed feature location activities during our other activities, such as when we manually analysed the code and when we performed clone detection and diffing. These activities were categorized to their own activity types. The overall issue was therefore that the activities blended into each other too much, which made them hard to log separately.

Lack of planning is another factor which increased the time it took to perform certain activities, mainly the annotation of features in the code. We did not really have any plan at first on how we were to perform this, so we tackled it without any real structure in mind. The outcome of this was poorly annotated features in terms of structure and readability. This leads to a less manageable SPL which could increase maintainability efforts.

As for the cost models, we only used the person hour metric to measure the cost of the SPL migration. When we designed the measurement approach, we thought that we would have more use of the other metrics, but we found it hard to quantify a cost based on them. An interesting metric in our measurement approach is the code changed metric, which we would have gotten more use out of if we did not have the issue with git, where we had a hard time tracking our changes. For these reasons, the cost models were not as useful to us as we had hoped from the beginning. We also did more thorough background research of cost models than necessary. We thought that the equations and metrics would be used more in our cost estimation than they were.

5.1.3 RQ3 - Comparison of Annotative and Compositional Approach

The comparison of the two approaches can be found in Table 4.2. There are some big differences that we aim to explain and discuss in this section. The biggest differences are found in the activity types SPLE training, Data cleansing, Domain analysis, Feature location, Transformation and finally Quality assurance.

We met with the other group and discussed what could have caused the discrepancies between the results. Since we did not know how they had worked and logged their work, it was crucial to meet and get an understanding of the differences between our work. As seen in Table 4.2, our approach took almost 125 person hours more to perform compared to the compositional approach. This was apparently caused by several different reasons. In some cases, it did not make sense to categorize an activity as only one activity type and for this reason it was not evident how much time we spent on each activity type. An example being the activity where we ran the games, which was part of the domain analysis activity type but also the feature identification type. In these cases we split the time we spent on the activity evenly between the two (or more) types.

The difference in the SPLE training activity type is caused by the fact that the compositional group did not log their time spent on reading literature before performing the transformation, as they did not see it as an activity that belonged to the migration. The activity type Data cleansing was not used in our annotative approach, since it was not a prioritization for us to for example translate German code comments. We spent a lot more time performing domain analysis, where the biggest difference was caused by the issues we had with Eclipse, Android Studio and ApoSnake. We spent a lot of time trying to get Android development working in Eclipse, as we did not want to use both Eclipse and Android Studio. After some time we gave up on using Eclipse for that purpose but we still had a lot of issues configuring and building the games in Android Studio. We could get all games to run except for ApoSnake, which we spent a lot of time trying to fix. In total, we spent several days on these things, which explains the difference between our logs. In Section 5.1.2 we discussed why we had only logged 7 hours on the Feature location

activity type. The reason was that feature location was tied to a lot of the other activities we performed, such as the clone detection and diffing activities. This was not only an issue with the Feature location type, but it was most evident on that one.

One big difference was the time we logged on Transformation. We discussed this difference during our meeting and found several reasons for this difference. One reason is that they did not specify a quality assurance activity, instead they categorized their transformation activity as both Transformation and Quality assurance. This means that they split the total cost of their transformation on these two activity types. While they did spend a lot of time performing quality assurance, they actually spent more time performing the actual transformation than on quality assurance. The time could have been more accurately divided if they had made more activities separating transformation and quality assurance. In addition, they explained that they did not log as much time as they should have for many of the activities, because their intention was to create more sub-activities and log the time more accurately, but they never did. Some of the time we spent on Branding was not important to the actual transformation, which explains some of the difference. In addition, since we performed our quality assurance throughout the transformation, we categorized quality assurance as Transformation as well, which added 30 hours to that activity type.

There was also a big difference in the Quality assurance activity type. As mentioned earlier, we categorized our quality assurance as two types, which meant that we divided the time spent between the types. They did something similar, as explained in the last paragraph, where they did not have a separate quality assurance activity. This meant that half of the time they spent on the transformation is categorized as Quality assurance. However, these logging issues are not the only reasons, they did say that they had a hard time performing quality assurance, due to the complexity of the compositional approach. The overall conclusion is that they spent more time than us on quality assurance, but not as much as the table suggests.

The comparison table suggests that our approach is much more time consuming, but as discussed above this is not necessarily true. Ideally, we could have used more metrics than time when comparing the two approaches. We implemented a similar amount of features during our migrations. We initially wanted to compare the code changed metrics, but this was not feasible due to our issues with git.

Overall, we think that the slightly better approach is the annotative approach, especially with limited SPLE experience. This is mainly because of the lower cost of entry and the reduced effort spent on quality assurance. While we had some issues with code readability, we think that they had more issues with the complexity of the composition-based approach. Once again, this would probably be mitigated with more experience. If there were more games to integrate into the SPL, the composition-based approach might be better suited for the task. The biggest reason for this is the improved structure and maintainability of the compositional approach in the long run. At the very least, we would recommend starting out with an anno-

tative approach and later transition into a compositional approach if more variants are needed or if maintainability becomes an issue.

5.2 Challenges

This section presents some challenges that we faced during the migration process. The challenges are not listed in any particular order.

One challenge we faced during the migration was that our activities often blended into each other, which made it hard to accurately log them. We faced the same issue when trying to categorize the activities into activity types, as many activities fall under different types. This in turn made it challenging to compare our results to the compositional approach.

An interesting challenge was that when we analysed the architecture of the games, we thought that the games would be more similar than they were. Even though two games used pretty much exactly the same classes and many of the same methods in each class, the implementation could differ quite a lot. The reason for the different implementations was mainly that the games used different versions of the same game engine. As an example, ApoDice and ApoClock had the exact same look and feel, but they used different rendering techniques, which hindered variability.

Another major challenge we faced was the branding of the games. Initially, we wanted to create a universal options menu for all the games in addition to a universal menu. This proved to be much harder than we initially thought, since the games were fundamentally different in their implementation of these views. This meant that we had to write a lot of code from scratch, since there was not much we could reuse from the other games. In the end, what we did with regards to branding was only to combine the menus from the three games where the menus were alike, i.e., ApoSnake, ApoDice and ApoClock. This reduced a lot of redundancy in the code, but ideally we would have wanted to perform additional branding. Due to time constraints we decided to not continue the branding activity and instead focus on adding new or improving existing feature annotations. Had we spent less time on branding, we think that the overall quality of the SPL would be higher.

This point builds upon the last point made. It was often very hard to know whether it was worth trying to extract a certain feature from several games, if the implementation of this feature was widely different in the different games. Does it really make sense to build a common code base if the code is drastically different? We were not sure about this. For it to be worth migrating to an SPL, we feel that the products need to be more similar to each other than they were with these games.

An issue we faced regarding git commits was that whenever we committed code, we had most likely changed configuration file in FeatureIDE. Sometimes this meant that thousands of lines of code had been changed since the last commit, which cluttered the code that we had actually changed. This made it hard to track the

changes that had been made in a specific commit. We fixed this issue after a while by creating a configuration file specifically for committing code. This meant that only the code that we actually changed was recorded with git.

The readability became worse when we merged the the files into one code base. Many methods in the code became long and messy after the merge, due to the need of keeping functionality from the different games within the same method. Some methods had several annotations from the different games, which made the method hard to read. This issue could have been avoided by keeping the whole method from each game, but that would lead to a lot of duplicated code in the common code base.

The SPLE tool we found most useful during the migration was FeatureIDE, which has been described in detail. We tried using BUT4Reuse during the feature location process, but we did not find the commonality analysis any better than the clone detection we performed using CPD. We wanted to try other tools that we had studied in the literature, but we decided not to spend time on trying to learn these tools, as much literature pointed to flaws in the automatic tools. If we had other artefacts in addition to source code, such as requirements or other documents, we could have gotten more use out of the SPLE tools. Overall, we did not get much use out of the classic SPLE tools, and it was hard for us to know when it would be worth trying to apply them to an activity.

Without a clear stakeholder in the project, it was hard for us to know which features to prioritize during the migration. In a real migration, requirements from a customer would be clear and it would be more obvious which features to implement. In our case, the goal was mostly to reduce the redundancy but we had a hard time knowing what to spend time on.

5.3 Threat to Validity

One threat to internal validity is the fact that all logging was done by ourselves, which can make it hard to ensure that the logging is performed correctly. To negate this threat, we made sure to perform logging each day to keep ourselves honest. In addition to the activity logs in Appendix A, we used a spreadsheet to log day-to-day which activities we worked on and on which games. The purpose of this was to ensure that we continuously logged our time. We looked into using a tool for logging the time spent in an IDE, such as WakaTime[1]. By using this tool we could log the time spent in Eclipse and Android Studio. Since several activities were performed within these programs, there was no way to fairly distribute the time spent on each activity. Therefore, we did not see any value in using a tool like this.

A lot of the theory and tools that we used, such as SPLE and FeatureIDE, were new to us. We had no previous experience working with some of the tools and little or no knowledge about some of the theory subjects, which increased the time it took

[1]WakaTime: https://wakatime.com/

to understand the subject as well as the tools. It is an important part of the thesis to correctly understand the theory and the tools that were used, in order to perform a successful migration.

It is hard to classify everything that is done as an activity. As an example, we spent a few days worth of time trying to get Android development working in Eclipse, so that we did not have to use Android Studio as well. We did not classify this as an activity since it was not part of the actual migration, but it would have been nice to see exactly how our time was spent.

An external threat was the fact that the source code for the games had more variation than we first thought, as mentioned earlier. This was obviously out of our control but we tried to do our best with what we had. Ideally, the SPL would consist only of the three games ApoSnake, ApoDice and ApoMono. However, if there only were three games in total it might have not made any sense to migrate them to an SPL.

Another external threat was the limited size of the applications. With a larger code base our findings might have been different, which can make it harder to generalize our findings. We do believe that most of the activities that we performed would be used in a migration of larger applications. However, the transformation and domain analysis activities would take more time to perform with a larger code base. In addition, we believe that many of the challenges we faced would be faced in a larger migration as well.

5.4 Limitations

The biggest limiting factor for this thesis was the lack of experience in the SPL area, as well as with the Apo-Games. With more knowledge on the subject, we would have spent less time on researching background and more time on the migration process. If we had previous experience with the games, the whole migration process would have been easier to perform.

The fact that the thesis was executed during a time span of about 20 weeks makes time a limitation. With more time to use, more activities and a more thorough analysis could have been performed. As mentioned earlier, having more experience on the subject in general would allow us to achieve more during the time frame.

6
Conclusion

Annotative SPLs are good for migrating smaller cloned variants with limited experience. We were able to get started with the migration fairly quickly and got at least the basics working without encountering too many issues. It does not necessarily lead to a clean and structured code base, but it does work sufficiently.

We feel that our migration was decently successful, as it most importantly retained all functionality that was in the games from the start. In addition, some interesting variations of the games could be created using our SPL. The main functionality that can be modified is the different activities that are present in the games, such as Editor, Userlevels and Options. These can be enabled or disabled according to specification. In total, we annotated 28 features which is not as many as we hoped from the beginning. One thing worth noting is that we had more features annotated at one point that we removed as we performed the branding, with the purpose of reducing redundancy. One thing that we feel are missing from our SPL is the variation in game logic. We wanted to do more with the game logic but with our limited knowledge and the big differences between the games, we did not find it worth to spend time on that.

Most of the activities that we performed during the migration contributed positively to the result of the SPL, some more than others. Note that the activities that we performed are not the only way to perform an extractive SPL migration, but rather a list of activities that worked well for us. Every SPL migration will be different with regards to the domain, previous experience and other factors which influences the impact of a given activity. This implies that the challenges we faced during the different activities also will differ in other migrations.

Most of the activities that we performed during the migration were manageable. We could, even without any previous experience with SPLs, perform the activities with an acceptable result. The activities had different levels of impact on the SPL and the final result. It was hard for us to know in advance what impact an activity would have on the SPL, which made it necessary for us to perform an activity if we believed that it would have any positive impact on the SPL and the result. The transformation activities were the most time-consuming and most error-prone activities, which is why we relied heavily on the quality assurance activity to ensure that the games we generated retained their functionality.

The conclusion that we can draw between our approach and the compositional ap-

proach is that many of the same activities are performed. Both approaches found code diffing to be the most powerful activity to find variation between the games. The biggest difference was the amount of time spent assuring the quality of the games. The biggest reason for this was the challenge of logging the activities. Both approaches could have logged activities in a more fine-grained manner to see exactly where time was spent, but that could potentially make it harder to compare the results. To gain more out of the comparison, we discussed our findings between groups which was of great value to us. Overall, both groups agreed that an annotative approach might be easier better to start with, mostly due to the lower cost of entry. With enough planning, structure and experience, it might make sense to transition into a compositional SPL which should be easier to maintain compared to an annotation-based one.

With regards to future work, this case study could be repeated again with similar results by using the same games to perform a migration on. The results of our SPL can be improved upon by spending more time on identifying features in the games' logic algorithms. By doing this, maybe the game logic of the games could be merged better and decrease the amount of code that are used for the game logic of all the games.

More time can be spent on branding activities which can improve the overall quality of the games. A lot of things could be branded, for example having language and sound options in all the games. There could also be branding to have a common theme in the menus for all the games. It would feel more like the games are part of the same SPL if more branding was done, since the games would then look more similar to each other.

Bibliography

Andam, B., Burger, A., Berger, T., and Chaudron, M. (2017). Florida: Feature location dashboard for extracting and visualizing feature traces. In *VaMoS*.

Antkiewicz, M., Ji, W., Berger, T., Czarnecki, K., Schmorleiz, T., Lämmel, R., Stanciulescu, S., Wasowski, A., and Schäfer, I. (2014). Flexible product line engineering with a virtual platform. In *ICSE*.

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013). *Feature- Oriented Software Product Lines*. Springer.

Assunção, W., Lopez-Herrejon, R., Linsbauer, L., Vergilio, S., and Egyed, A. (2017). Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering*, 22(6).

Batory, D. (2006). *A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Boehm, B., Brown, A., Madachy, R., and Ye Yang (2004). A software product line life cycle cost estimation model. *2004 International Symposium on Empirical Software Engineering*.

Businge, J., Moses, O., Nadi, S., Bainomugisha, E., and Berger, T. (2018). Clone-based variability management in the android ecosystem. In *ICSME*.

Classen, A., Boucher, Q., and Heymans, P. (2011). A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, 76(12):1130 – 1143.

Clements, P., McGregor, J., and Cohen, S. (2005). The structured intuitive model for product line economics (SIMPLE). *Technical Report CMU/SEI-2005-TR-003*, (February).

Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., and Czarnecki, K. (2013). An exploratory study of cloning in industrial software product lines. In *CSMR*.

Jepsen, H. P. and Beuche, D. (2009). Running a software product line: standing still is going backwards. In *SPLC*.

Ji, W., Berger, T., Antkiewicz, M., and Czarnecki, K. (2015). Maintaining Feature Traceability with Embedded Annotations. In *SPLC*.

Kang, K., Cohen, S., Hess, J., Nowak, W., and Peterson, S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep., SEI, CMU.

Kästner, C. and Apel, S. (2008). Integrating compositional and annotative approaches for product line engineering. In *GPLE*.

Kästner, C., Thum, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., and Apel, S. (2009). FeatureIDE: A tool framework for feature-oriented software development. In *ICSE*.

Krüger, J., Fenske, W., Meinicke, J., Leich, T., and Saake, G. (2016). Extracting software product lines: a cost estimation perspective. In *Proceedings of the 20th International Systems and Software Product Line Conference, SPLC 2016, Beijing, China, September 16-23, 2016*, pages 354–361. ACM.

Krüger, J., Schröter, I., Kenner, A., Kruczek, C., and Leich, T. (2016). Featurecopp: Compositional annotations. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*, FOSD 2016, pages 74–84, New York, NY, USA. ACM.

Krüger, J., Fenske, W., Thüm, T., Aporius, D., Saake, G., and Leich, T. (2018). Apo-games - a case study for reverse engineering variability from cloned java variants. In *SPLC*.

Krüger, J., Mukelabai, M., Gu, W., Shen, H., Hebig, R., and Berger, T. (2019). Where is my feature and what is it about? a case study on recovering feature facets. *Journal of Systems and Software*.

Kästner, C. (2007). Cide: Decomposing legacy applications into features. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings. Second Volume (Workshops)*, pages 149–150. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan.

Lee, K., Kang, K. C., and Lee, J. (2004). Concepts and Guidelines of Feature Modeling for Product Line Software Engineering.

Martinez, J., Ziadi, T., Bissyandé, T. F., Klein, J., and Traon, Y. L. (2017). Bottom-up technologies for reuse: automated extractive adoption of software product lines. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 67–70. IEEE Computer Society.

Nóbrega, J. P., de Almeida, E. S., and Meira, S. R. L. (2008). Income: Integrated cost model for product line engineering. In *2008 34th Euromicro Conference Software Engineering and Advanced Applications*. IEEE.

Pfleeger, S. L. (1997). Measuring software reuse: Principles, practices, and economic models. *IBM Systems Journal*, 36(3):464.

Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G., and Rajlich, V. (2007). Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432.

Rubin, J. and Chechik, M. (2013). A survey of feature location techniques. In *Domain Engineering*.

Wang, J., Peng, X., Xing, Z., and Zhao, W. (2013). How developers perform feature location tasks: a human-centric and process-oriented exploratory study. *Journal of Software: Evolution and Process*, 25(11):1193–1224.

Xue, Y. (2011). Reengineering legacy software products into software product line based on automatic variability analysis. *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1114–1117.

A

Activities

INFORMATION

- **Activity**: Running and testing the games
- **Activity ID**: A1
- **Activity type**: Domain analysis
- **Start date**: 2019-02-18
- **End date**: 2019-02-27
- **Description**: Installed via Google Play and played each game for a while, to find out how they work as well as to find similarities between the games.

DATA

- **Person hours spent**: 30
- **Number of commits**: N/A
- **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

ARTIFACTS

- **Input**: Mobile application and source code.
- **Output**: Understanding of the games, what different features there might be and similarities between the different games.
- **Tools used**: Smartphone with Android OS, Android Studio

ACTIVITY DESCRIPTION

- **Complexity**: Easy to install and run the games on the phones.
- **Importance**: Important to understand what the game is about and how it works. Also gives us a first set of features to work with.
- **Dependencies on other activities**: None

Table A.1: Log of activity A1

INFORMATION

- **Activity**: Architecture analysis
- **Activity ID**: A2
- **Activity type**: Domain analysis, architecture identification
- **Start date**: 2019-02-21
- **End date**: 2019-02-26
- **Description**: IntelliJ was used to create Java class diagrams for each of the games. This is to get an architectural overview of the components of the games.

DATA

- **Person hours spent**: 10
- **Number of commits**: N/A
- **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

ARTIFACTS

- **Input**: Source code
- **Output**: Class diagram
- **Tools used**: IntelliJ IDE

ACTIVITY DESCRIPTION

- **Complexity**: Creating the diagrams is a bit time consuming, but nothing difficult. The hard part is understanding the diagrams and comparing them to the other ones.
- **Importance**: Class diagrams are highly useful in that we can see how each of the games are built. We can see the classes and the dependencies between classes.
- **Dependencies on other activities**: None

Table A.2: Log of activity A2

INFORMATION

- **Activity**: Manually analysing code
- **Activity ID**: A3
- **Activity type**: Domain analysis
- **Start date**: 2019-02-25
- **End date**: 2019-02-27
- **Description**: While performing A2, we wanted a better understanding of the code which meant we had to manually look at it.

DATA

- **Person hours spent**: 15
- **Number of commits**: N/A
- **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

ARTIFACTS

- **Input**: Source code
- **Output**: Deeper understanding of the domain.
- **Tools used**: IntelliJ IDE, Sublime Text, Visual Studio Code.

ACTIVITY DESCRIPTION

- **Complexity**: Challenging to begin with, but not too hard to understand how the games are built with the help of class diagrams.
- **Importance**: Very important. The code needs to be understood to be able to locate and extract features.
- **Dependencies on other activities**: A2

Table A.3: Log of activity A3

INFORMATION

- **Activity**: Building and running games in Android Studio
- **Activity ID**: A4
- **Activity type**: Domain analysis
- **Start date**: 2019-02-26
- **End date**: 2019-03-01
- **Description**: To be able to verify that the product line works at the end of the thesis, we have to make sure that all the games can be built and run on an emulator in Android Studio or Eclipse. In the end, we managed to build only in Android Studio and not in Eclipse.

DATA

- **Person hours spent**: 40
- **Number of commits**: N/A
- **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

ARTIFACTS

- **Input**: Source code
- **Output**: Game running in emulator
- **Tools used**: Android Studio and Eclipse

ACTIVITY DESCRIPTION

- **Complexity**: Since we had no documentation of how to build the games, and no previous experience in importing old projects into Android Studio/Eclipse, this activity was hard and time consuming. First we tried to use Android Studio but we couldn't get it to work, then we went to Eclipse but that was even worse since Android programming in Eclipse is no longer supported. Then we went back to Android Studio and figured it out.
- **Importance**: Very important, since we will have to build the games after we have completed the migration.
- **Dependencies on other activities**: None

Table A.4: Log of activity A4

INFORMATION

- **Activity**: Installing and learning FeatureIDE
- **Activity ID**: A5
- **Activity type**: SPLE training, feature modeling
- **Start date**: 2019-02-28
- **End date**: 2019-03-04
- **Description**: Installing FeatureIDE and figuring out how it works. Creating a small project to learn.

DATA

- **Person hours spent**: 20
- **Number of commits**: N/A
- **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

ARTIFACTS

- **Input**: Source code
- **Output**: Feature model
- **Tools used**: Eclipse, FeatureIDE - Antenna

ACTIVITY DESCRIPTION

- **Complexity**: Not too complex, it took a while to install since we needed to downgrade to an older Eclipse version.
- **Importance**: Very important since we will build our feature models and configure the games in FeatureIDE.
- **Dependencies on other activities**: None

Table A.5: Log of activity A5

INFORMATION

- **Activity**: Code diffing
- **Activity ID**: A6
- **Activity type**: Diffing, SPLE training
- **Start date**: 2019-03-05
- **End date**: 2019-03-08
- **Description**: Diffing was performed using Meld to find similarities between the games. Each game was compared to all other games and the results can be found in the table on the next page.

DATA

- **Person hours spent**: 40
- **Number of commits**: N/A
- **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

ARTIFACTS

- **Input**: Source code
- **Output**: Similar classes that might be features
- **Tools used**: Meld

ACTIVITY DESCRIPTION

- **Complexity**: While not too complex, the tough thing about diffing is deciding whether or not two classes are similar. Since all files are different in some way, it's tough to draw a line on what is similar and what is not.
- **Importance**: Diffing and/or clone detection is important since it is a way to find features in cloned variants, since they are often cloned between products.
- **Dependencies on other activities**: None

Table A.6: Log of activity A6

INFORMATION

- **Activity**: Transformation: ApoSnake and ApoDice
- **Activity ID**: A7
- **Activity type**: Transformation
- **Start date**: 2019-03-08
- **End date**: 2019-03-13
- **Description**: As a first transformation test, we performed a simple transformation with the games that we found most similar in A6, which were ApoSnake and ApoDice. We transformed the games into a common platform with a simple feature model where either game can be chosen, along with a menu theme from either of the games.

DATA

- **Person hours spent**: 25
- **Number of commits**: 3
- **LOC added**:
- **LOC removed**: 2250
- **LOC modified**:
- **Number of files added**: N/A
- **Number of files removed**: 19
- **Number of files modified**: 19

ARTIFACTS

- **Input**: Source code from two different games, ApoDice and ApoSnake
- **Output**: A combined source code for both games and a feature model
- **Tools used**: Meld, Eclipse, FeatureIDE - Antenna, Android Studio

ACTIVITY DESCRIPTION

- **Complexity**: Medium complexity, since it was the first time we performed a transformation. It was tough to work in Eclipse and Android Studio, since we had lots of problem syncing the code before me managed to link it.
- **Importance**: Very important. This is the first step in making the transformation.
- **Dependencies on other activities**: A4, A5, A6

Table A.7: Log of activity A7

INFORMATION

- **Activity**: Feature identification
- **Activity ID**: A8
- **Activity type**: Feature location, domain analysis, feature identification
- **Start date**: 2019-03-11
- **End date**: 2019-03-14
- **Description**: Following the diffing process, further feature identification has to be done to be able to create an extensive feature model.

DATA

- **Person hours spent**: 20
- **Number of commits**: N/A
- **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

ARTIFACTS

- **Input**: Source code, games, class diagrams, diffing.
- **Output**: Common and unique features of the games.
- **Tools used**: Meld, Eclipse, Android Studio

ACTIVITY DESCRIPTION

- **Complexity**: High complexity.
- **Importance**: Very important to identify the different features in order to create a complete feature model.
- **Dependencies on other activities**: A1, A2, A3, A6

Table A.8: Log of activity A8

INFORMATION

- **Activity**: Transformation: ApoSnake, ApoDice and ApoClock
- **Activity ID**: A9
- **Activity type**: Transformation
- **Start date**: 2019-03-26
- **End date**: 2019-03-29
- **Description**: As a first transformation test, we performed a simple transformation with the games that we found most similar in A6, which were ApoSnake and ApoDice. We transformed the games into a common platform with a simple feature model where either game can be chosen, along with a menu theme from either of the games.

DATA

- **Person hours spent**: 15
- **Number of commits**: 4
- **LOC added**:
- **LOC removed**: 5447
- **LOC modified**:
- **Number of files added**: 11
- **Number of files removed**: 38
- **Number of files modified**: 19

ARTIFACTS

- **Input**: Source code from three different games, ApoDice, ApoSnake and ApoClock
- **Output**: A combined source code for the three different games and a feature model
- **Tools used**: Meld, Eclipse, FeatureIDE - Antenna, Android Studio

ACTIVITY DESCRIPTION

- **Complexity**: Medium complexity, since it was the first time we performed a transformation. It was tough to work in Eclipse and Android Studio, since we had lots of problem syncing the code before me managed to link it.
- **Importance**: Very important. This is the first step in making the transformation.
- **Dependencies on other activities**: A4, A5, A6

Table A.9: Log of activity A9

INFORMATION

- **Activity**: Transformation: ApoSnake, ApoDice, ApoClock and ApoMono
- **Activity ID**: A10
- **Activity type**: Transformation
- **Start date**: 2019-04-08
- **End date**: 2019-03-12
- **Description**: Merged the game ApoMono into the common code base Apo-Hybrid.

DATA

- **Person hours spent**: 20
- **Number of commits**: 4
- **LOC added**:
- **LOC removed**: 7770
- **LOC modified**:
- **Number of files added**: 6
- **Number of files removed**: 18
- **Number of files modified**: 18

ARTIFACTS

- **Input**: Source code from four different games, ApoDice, ApoSnake, ApoClock and ApoMono
- **Output**: A combined source code for the four games that were merged
- **Tools used**: Meld, Eclipse, FeatureIDE - Antenna, Android Studio

ACTIVITY DESCRIPTION

- **Complexity**:
- **Importance**:
- **Dependencies on other activities**: A4, A5, A6, A7, A8, A9

Table A.10: Log of activity A10

INFORMATION

- **Activity**: Branding - Font
- **Activity ID**: A11
- **Activity type**: Transformation
- **Start date**: 2019-04-12
- **End date**: 2019-04-17
- **Description**: Adds ability to change the font in all of the games to the fonts of the other games

DATA

- **Person hours spent**: 15
- **Number of commits**: 2
- **LOC added**: 100
- **LOC removed**: 40
- **LOC modified**:
- **Number of files added**: 0
- **Number of files removed**: 0
- **Number of files modified**: 3

ARTIFACTS

- **Input**: Source code, font files
- **Output**: Source code
- **Tools used**: Android Studio

ACTIVITY DESCRIPTION

- **Complexity**: Medium, since the games handles fonts in different ways.
- **Importance**: Decently important for branding purposes.
- **Dependencies on other activities**:

Table A.11: Log of activity A11

INFORMATION

- **Activity**: Transformation: All games
- **Activity ID**: A12
- **Activity type**: Transformation
- **Start date**: 2019-04-15
- **End date**: 2019-03-
- **Description**:

DATA

- **Person hours spent**: 25
- **Number of commits**: 4
- **LOC added**:
- **LOC removed**: 8970
- **LOC modified**:
- **Number of files added**: 7
- **Number of files removed**: 20
- **Number of files modified**: 20

ARTIFACTS

- **Input**: Source code from all different games, ApoDice, ApoSnake, ApoClock, ApoMono and MyTreasure
- **Output**: A combined source code for all games and a feature model
- **Tools used**: Meld, Eclipse, FeatureIDE - Antenna, Android Studio

ACTIVITY DESCRIPTION

- **Complexity**: High, since MyTreasure is the most different game to all other games. This meant that we could not integrate as much as we would like into the existing code base.
- **Importance**: Important, since we needed to include all games in the SPL.
- **Dependencies on other activities**: A4, A5, A6, A7, A8, A9, A10

Table A.12: Log of activity A12

INFORMATION

- **Activity**: Branding - Menu
- **Activity ID**: A13
- **Activity type**: Transformation
- **Start date**: 2019-04-18
- **End date**: 2019-03-23
- **Description**: Create a universal menu for all games except MyTreasure and ApoMono which have their own unique menus. Mostly for branding purposes but also to reduce redundancy in code.

DATA

- **Person hours spent**: 40
- **Number of commits**: 5
- **LOC added**: 270
- **LOC removed**: 350
- **LOC modified**: 50
- **Number of files added**: 0
- **Number of files removed**: 0
- **Number of files modified**: 4

ARTIFACTS

- **Input**: Source code from ApoSnake, ApoDice and ApoClock
- **Output**: Source code
- **Tools used**: Eclipse, FeatureIDE, Android Studio

ACTIVITY DESCRIPTION

- **Complexity**: Quite challenging, mostly because ApoClock's menu is different from Snake and Dice and they use different rendering technologies.
- **Importance**: Decently important for branding purposes and to reduce redundancy in code.
- **Dependencies on other activities**: A12

Table A.13: Log of activity A13

INFORMATION
- **Activity**: General SPLE learning
- **Activity ID**: A14
- **Activity type**: SPLE training
- **Start date**: 2019-02-11
- **End date**: 2019-02-26
- **Description**: Studying the subject, which includes all literature based on the SPLE topic as well as information from a course on the subject.

DATA
- **Person hours spent**: 80
- **Number of commits**: N/A
- **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

ARTIFACTS
- **Input**: Literature, lecture slides
- **Output**: General SPLE knowledge
- **Tools used**: N/A

ACTIVITY DESCRIPTION
- **Complexity**: Not too hard, but time consuming to read and understand.
- **Importance**: Very important to understand the subject which is the foundation of the thesis.
- **Dependencies on other activities**: N/A

Table A.14: Log of activity A14

INFORMATION

- **Activity**: Annotating features
- **Activity ID**: A15
- **Activity type**: Transformation
- **Start date**: 2019-03-07
- **End date**: 2019-05-03
- **Description**: Transforming source code into annotated features. Note that the branding activities could have been included in this activity.

DATA

- **Person hours spent**: 40
- **Number of commits**: 24
- **LOC added**: 80
- **LOC removed**: 60
- **LOC modified**: 110
- **Number of files added**: N/A
- **Number of files removed**: N/A
- **Number of files modified**: 5

ARTIFACTS

- **Input**: Source code
- **Output**: Source code
- **Tools used**: FeatureIDE, Android Studio, Eclipse

ACTIVITY DESCRIPTION

- **Complexity**: Challenging, since code for a feature is scattered in the code
- **Importance**: Of utmost importance. Features are the building blocks of the SPL
- **Dependencies on other activities**: A7, A9, A10, A12

Table A.15: Log of activity A15

INFORMATION

- **Activity**: Quality assurance
- **Activity ID**: A16
- **Activity type**: Quality assurance, transformation
- **Start date**: 2019-03-07
- **End date**: 2019-05-05
- **Description**: After each major addition to the SPL, each game has to be built and run with their default configurations, to make sure no functionality has been lost. This is also when new features and configurations are tested.

DATA

- **Person hours spent**: 60
- **Number of commits**: N/A
- **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

ARTIFACTS

- **Input**: Source code
- **Output**: Running game
- **Tools used**: FeatureIDE, Android Studio

ACTIVITY DESCRIPTION

- **Complexity**: Medium, usually some errors occur from annotating features in different places, but they are small most of the time.
- **Importance**: Very important, to maintain the quality of the products.
- **Dependencies on other activities**: A7, A9, A10, A12, A15

Table A.16: Log of activity A16

B

Apo-Games

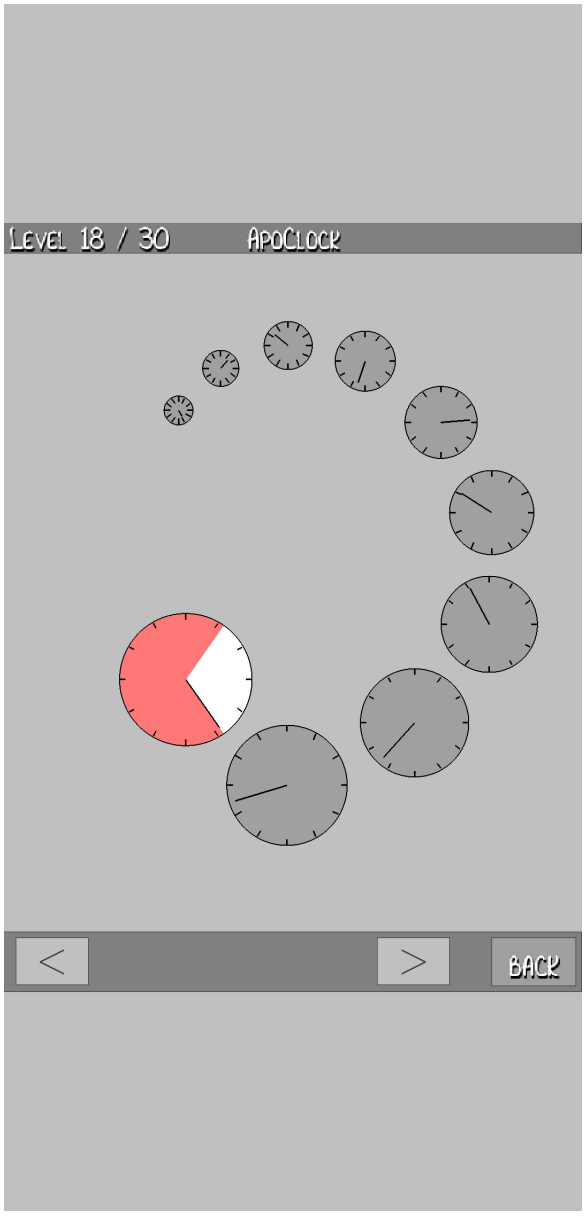

Figure B.1: Ingame overview of ApoClock

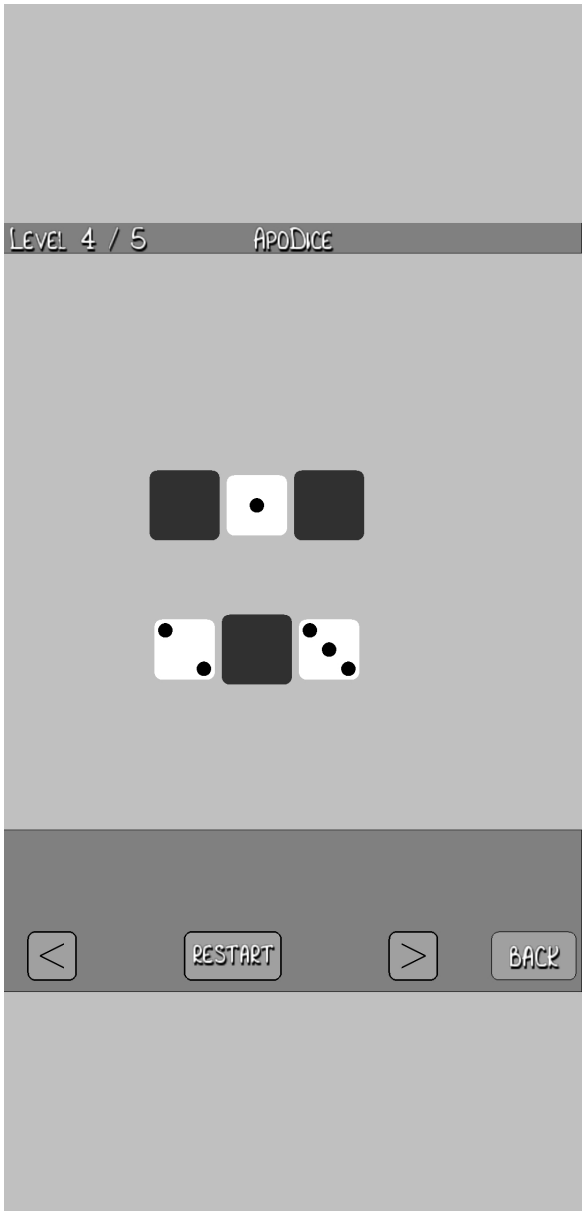

Figure B.2: Ingame overview of ApoDice

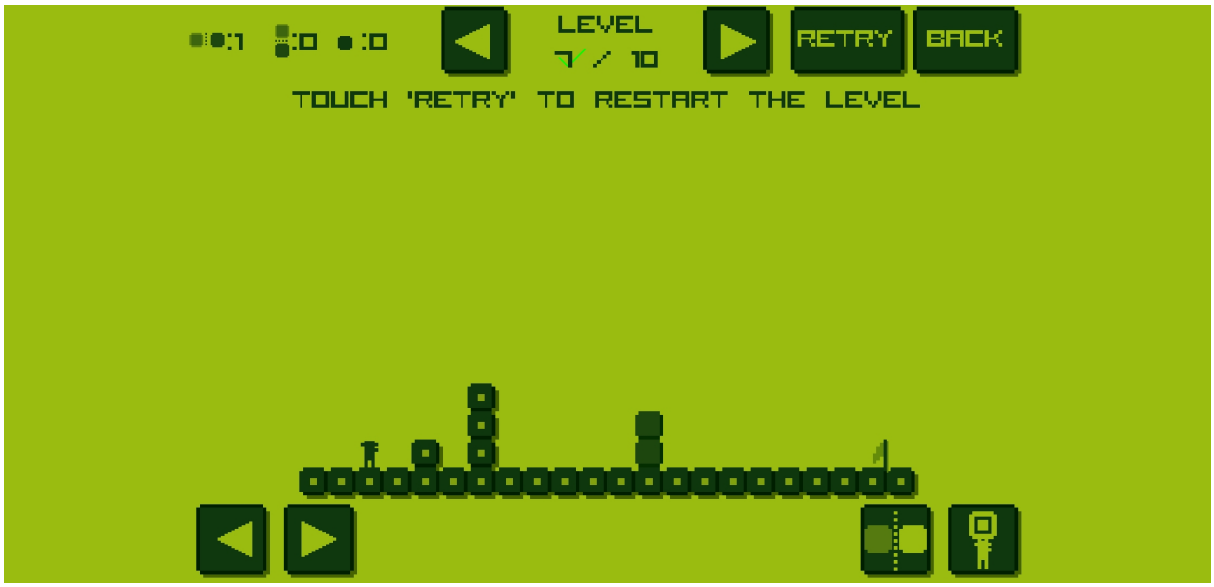

Figure B.3: Ingame overview of ApoMono

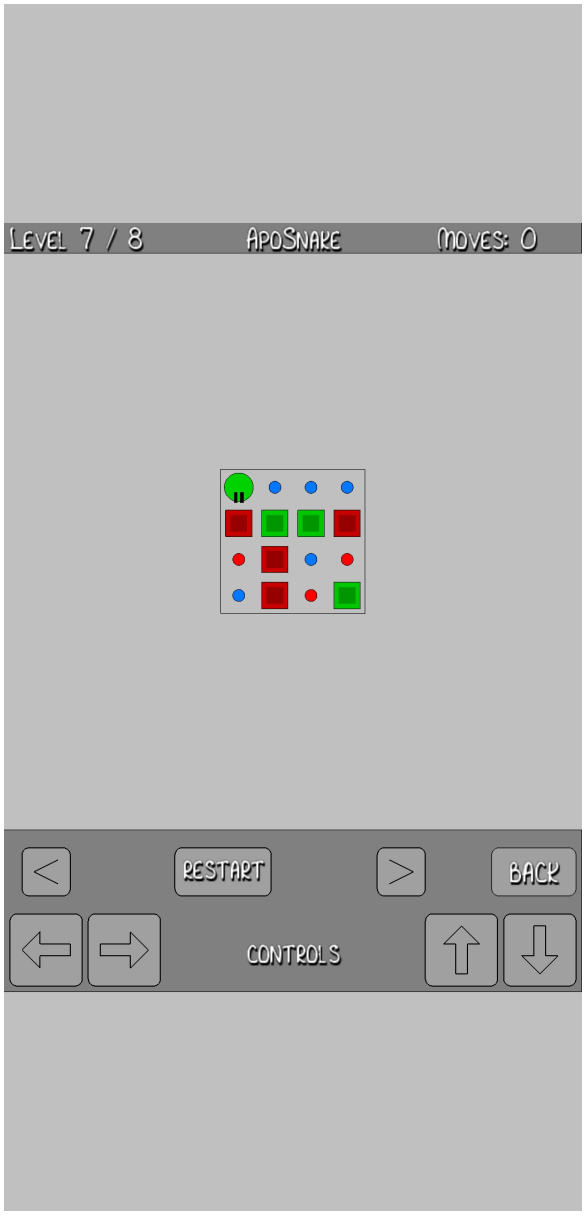

Figure B.4: Ingame overview of ApoSnake

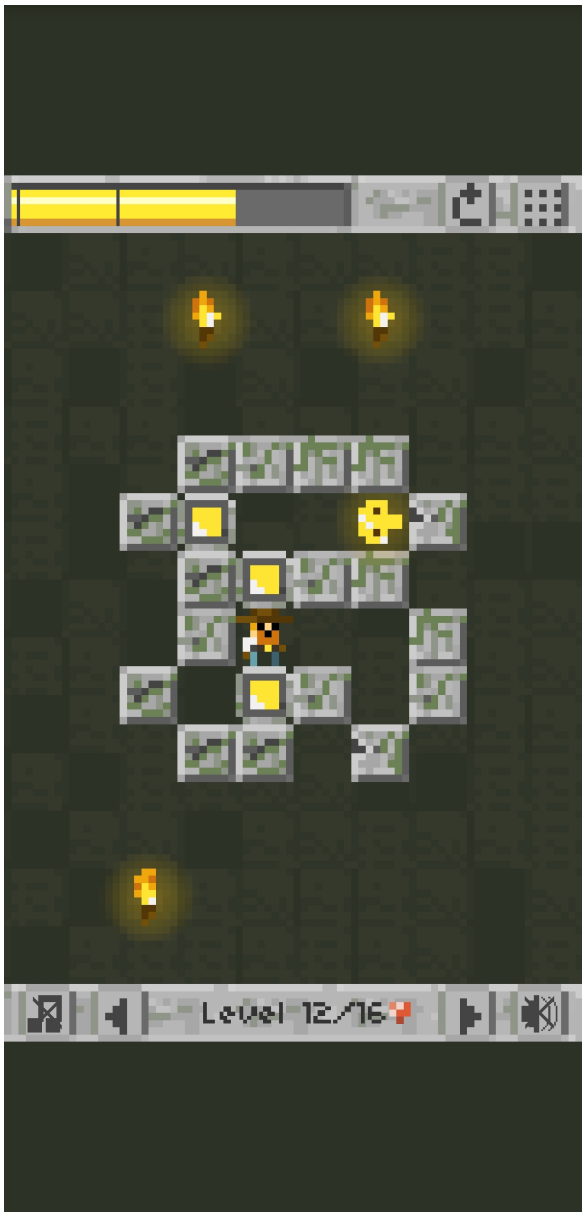

Figure B.5: Ingame overview of MyTreasure

C
Feature Model

Menu: The main menu of the game. Have three different themes: **MonoMenu** for ApoMono, **TreasureMenu** for MyTreasure and **ApoMenu** which is our own branded menu for ApoSnake, ApoDice and ApoClock.
Editor: The user can create their own custom made level.
Userlevels: Here can the user select between all the custom made levels made by all users.
World: Contains features that are part of the ingame world. Can either be **Dynamic** or **Static** objects.
Dice: The dice object for ApoDice.
Clock: The clock object for ApoClock.
Avatar: Avatar contains three different avatars depending on which game. **MonoAvatar** for ApoMono, **TreasureAvatar** for MyTreasure and **SnakeAvatar** for ApoSnake.
TreasureBox and **TreasureBlock**: Dynamic objects that are used in MyTreasure.
MonoObjects: Static objects that are part of the ApoMono game.
TreasureKey: Static key, which is the objective of the game, that are part of the MyTreasure game.
Canvas: The canvas that renders the playing field. Can be **Grey** which is used for ApoDice, ApoSnake and ApoClock, **TreasureCanvas** that is for MyTreasure and **MonoCanvas** which is for ApoMono.
Controls: Decides how the user controls the game. Can either be by **Drag**, **Press** or **MoveButtons**.
Score: To calculate the score of game. Can either be by **Counter** or **Moves**.
Options: The options menu includes **Sound**, **Language**, **Nickname** and **MonoColor**.
Sound: Sound can either be **Effects** or **Music**. Effects has two different types of sound which are **TreasureEffect** and **MonoEffect**. Music has two different sounds which are **TreasureMusic** and **MonoMusic**.
Language: Language of the game. Can either be **German** or **English**.
MonoColor: Chooses the color that are used in the background of ApoMono. Can either be **White** or **Green**.
GameLogic: Which game logic should be used. Can be **SnakeLogic**, **DiceLogic**, **ClockLogic**, **MonoLogic** or **TreasureLogic**.
Font: Decides what font to use in the game. Can be **ApoFont**, **MonoFont** or **TreasureFont**.