Efek Design Pattern Terhadap Duplicated Code dan Efek Terusannya Pada Maintainability Aplikasi Berbasis Mobile

Tugas Akhir

diajukan untuk memenuhi salah satu syarat

memperoleh gelar sarjana

dari Program Studi Informatika

Fakultas Informatika

Universitas Telkom

1301174065

Firdaus Ardhana Indradhirmaya

Program Studi Sarjana Informatika

Fakultas Informatika Universitas

Telkom

Bandung

2021

LEMBAR PENGESAHAN

Efek Design Pattern Terhadap Duplicated Code dan Efek Terusannya Pada Maintainability Aplikasi Berbasis Mobile

The Effects of Design Pattern on Duplicated Code and its Subsequent Effects on Maintainability of Mobile Application

NIM: 1301174065

Firdaus Ardhana Indradhirmaya

Tugas akhir ini telah diterima dan disahkan untuk memenuhi sebagian syarat memperoleh gelar pada Program Studi Sarjana Informatika
Fakultas Informatika
Universitas Telkom

Bandung, 31 Juli 2021
Menyetujui

Pembimbing I	Pembimbing II
Dawam Dwi Jatmiko Suwawi, S.T., M.T.	Shinta Yulia Puspitasari, S.T., M.T.
NIP: 14890033-1	NIP: 18880124

Ketua Program Studi
Sarjana Informatika,

Dr. Erwin Budi Setiawan, S.Si., M.T.
NIP: 0405117601

LEMBAR PERNYATAAN

Dengan ini saya, Firdaus Ardhana Indradhirmaya, menyatakan sesungguhnya bahwa Tugas Akhir saya dengan judul ”Efek Design Pattern Terhadap Duplicated Code dan Efek Terusannya Pada Maintainability Aplikasi Berbasis Mobile” beserta dengan seluruh isinya adalah merupakan hasil karya sendiri, dan saya tidak melakukan penjiplakan yang tidak sesuai dengan etika keilmuan yang belaku dalam masyarakat keilmuan. Saya siap menanggung resiko/sanksi yang diberikan jika dikemudian hari ditemukan pelanggaran terhadap etika keilmuan dalam buku TA atau jika ada klaim dari pihak lain terhadap keaslian karya.

Bandung, 31 Juli 2021

Yang Menyatakan,

Firdaus Ardhana Indradhirmaya

Efek Design Pattern Terhadap Duplicated Code dan Efek Terusannya Pada Maintainability Aplikasi Berbasis Mobile

Firdaus Ardhana Indradhirmaya[1], Dawam Dwi Jatmiko Suwawi[2], Shinta Yulia Puspitasari[3]

[1,2,3] Fakultas Informatika, Universitas Telkom, Bandung
[1]firdausai@students.telkomuniversity.ac.id, [2]dawamdjs@telkomuniversity.ac.id,
[3]shintayulia@telkomuniversity.ac.id

Abstrak

Kemajuan teknologi mobile yang sangat pesat tidak lepas dari berbagai macam permasalahan, khususnya pengabaian proses pengembangan aplikasi mobile yang ideal. Hal tersebut telah menimbulkan berbagai macam masalah, salah satunya adalah meningkatnya jumlah duplicate code, sebuah permasalahan yang sering terjadi pada aplikasi mobile berbasis android. Selebihnya, hal tersebut juga mengakibatkan menurunnya tingkat maintainability pada sebuah aplikasi. Penelitian-penelitian sebelumnya menyatakan bahwa pembuatan abstract class dapat mengatasi permasalahan duplicate code, namun juga menurunkan tingkat maintainability. Tujuan penelitian ini adalah untuk menerapkan dan mengamati efek design pattern, yang melibatkan pembuatan abstract class, untuk mengatasi permasalahan duplicate code dan juga melihat efek terusannya pada maintainability sebuah aplikasi. Metodologi penelitian yang telah dilakukan adalah membandingkan tingkat maintainability dan jumlah duplicate code sebelum dan sesudah penerapan design pattern terpilih. Jumlah baris duplicate code dan ISO 25010 akan digunakan sebagai metrik duplicate code dan maintainability masing-masing. Penerapan template pattern terbukti mampu untuk menekan jumlah duplicate code, namun gagal untuk meningkatkan atau mempertahankan tingkat maintainability secara keseluruhan. Fitur inheritance yang digunakan oleh template pattern akan selalu mengakibatkan metrik depth of inheritance dan coupling untuk memburuk. Selebihnya, efek dari kode yang di abstraksi memungkinkan terjadinya pemburukan pada metrik cohesion, complexity, dan number of methods. Dimana metrik-metrik yang disebutkan berpengaruh terhadap tingkat maintainability, yang meliputi aspek reusability, modifiability, modularity, testability dan analysability

Kata kunci : design pattern, duplicate code, maintainability, mobile

Abstract

The rapid advancement of mobile technology brings a variety of issues along with it. One of those issues is neglecting the ideal mobile application development process. Such neglect has caused the increased number of duplicate codes, the most occurring issue in the android application. Furthermore, it also has caused the level of maintainability in mobile apps to drop. Previous studies have shown that abstract classes can decrease code duplication while also decrease the level of maintainability. This research aims to see the effects of implementing design patterns, most of which involve creating abstract class, to reduce duplicate code while also observing its subsequent effects on the maintainability aspect of a mobile application. The amount of duplicate code and the maintainability aspect were measured before and after the implementation of design pattern. In addition, the number of lines of duplicate code and ISO 25010 were referenced and used as metrics to measure duplicate code and maintainability, respectively. The template pattern was proven to reduce duplicate codes but could not maintain or increase the maintainability aspect of an android mobile application as a whole. The depth of inheritance and coupling metric will always worsen due to the use of inheritance by the template pattern. Furthermore, the effect of the code being abstracted could negatively affect the cohesion, complexity, and number of methods metric. The metrics mentioned contributed to the reusability, modifiability, modularity, testability, and analysability aspect of maintainability.

Keywords: design pattern, duplicate code, maintainability, mobile

1. Introduction

Background

The rapid growth of mobile technology has been inseparable from various challenges in the last ten years [1]. The increasing demand and complex features in mobile applications cause developers to put aside quality

during the development process [1]. As a result, the maintainability aspect of mobile applications is neglected [15]. Moreover, the amount of duplicate code has also increased because of this negligence. Code duplication is the most occurring issue in Android-based applications [13].

Software are estimated to contain 5% to 50% of duplicate codes in them [14]. The existence of duplicate codes decrease the evolution, readability, reliability, and design aspects of software [14]. For instance, if a duplicate code needs to be modified, the developer must change all the scattered duplicate code in the application [14]. The modification of duplicate codes can increase maintenance costs and the possibility of faults in the system [14]. On the other hand, a study stated that 76% of duplicate code, classified as replicate and specialize, positively affects the maintainability aspect of software [11]. The construction of abstract classes, a solution for duplicate code problems, was observed to increase the system's complexity, thus decreasing the maintainability aspect [11].

A solution is needed to decrease the amount of duplicate code while also maintaining the maintainability aspect of an application. Design patterns are a repeatable solution to a commonly occurring problem in software design [6]. Most design patterns involve creating an abstract class, which according to Barbosa [3], is a solution to decrease the number of duplicate codes. Furthermore, Panca [15] observed that the implementation of design patterns increased the level of maintainability of applications.

Problem Statement

The most recurring problem in android applications is duplicate code [13]. Barbosa [3] proposed that the creation of abstract classes is one of the solutions that can reduce duplicate codes. However, a study conducted by Juergens [10] found that creating abstract classes negatively impacts the maintainability aspect of applications. Therefore, there needs to be a solution that can reduce the amount of duplicate code while maintaining or increasing the level of maintainability of an android application. Hence, this research will answer how the implementation of design patterns, most of which involved creating abstract classes, reduce duplicate codes and increase or maintain the maintainability aspect of an android application.

The scope of this research is to only utilize design patterns in order to solve duplicate code problems. Out of all of the duplicate code problems, only those identified to have a design problem that a design pattern can solve are refactored. Any other problem and duplicate code that does not correspond to any design pattern will be ignored. Furthermore, the ranking mechanism in the maintainability measurement system by Barbosa [3] that will be used in this research will be replaced by trend analysis. Since there is a new maintainability standard (ISO 25010), the previous ranking system will no longer be accurate to the current maintainability standard.

Research Goal

Design patterns, most of which involved creating abstract classes, fit one of the characteristics described by Barbosa [3] to decrease the number of duplicate codes. Additionally, design patterns have been observed to increase the maintainability level of applications [15]. The implementation of design pattern and its effect on duplicate code will be observed along with the maintainability level of the selected application. The goal is to determine if the implementation of design pattern can decrease duplicate code, while also increase, or at least maintain, the maintainability aspect of an android application.

Metrics pertaining to code duplication and maintainability will be measured before and after design patterns are implemented. The increase and decrease of value in all of the metrics used will help determine whether design pattern is the solution to this research's problem.

Paper Structure

In the beginning, this paper explores the various literature the scientific community has on android applications, mobile applications, duplicate code, maintainability, and static code analyzer. Then, it describes the methodology and the experiment conducted for this research. Next, the analysis and all of the findings that were discovered during this research are presented. Lastly, the conclusion of the research.

2. Literature Review

2.1 Challenges of Mobile App Development

Aldayel [1] discussed various challenges during the development process of mobile-based applications. Some of the challenges mentioned were security, operating system, sensors utilization, cross-platform compatibility, and limited resources. So, Aldayel [1] designed a guideline to mitigate these issues. The guideline consisted of planning, requirement gathering, design, architecture, user experience, development, testing, implementation, maintenance, support, and security.

Shahbudin [19] emphasize that design goals are the key to building and developing high-quality mobile applications quickly. What is more, a good design can ensure that errors and crashes are avoided. Additionally, the implementation of design patterns can increase the efficiency, usability, and reusability of components in an application.

2.2 The Effects of Code Duplication on Maintainability

Monden [14] examined the relationship between duplicate code and software quality. Using an application that has been continuously developed for the past 20 years as a case study, Monden [14] found that modules with duplicate codes are less maintainable due to having greater revision number than modules without duplicate codes. Moreover, Monden [14] found that 5% to 50% of applications consist of duplicate code.

Kapser [11] discovered that 71% of the duplicate code found in the applications they studied had a positive effect on the maintainability of the applications. What makes Kapser's [11] research different from other research is the use of motivation, advantages, disadvantages, management and long-term problems in measuring the impact of duplicate code on an application.

2.3 Design Pattern

Design patterns are a repeatable solution to a commonly occurring problem in software design [6]. Three groups that design patterns divide into are creational, structural, and behavioral. The creational design pattern provides various object creation mechanics, which can increase flexibility, and reuse of existing codes. Some design patterns that fall into the creational category are the builder method, the factory method, and the singleton pattern. The structural design pattern provides various ways to assemble objects and classes into larger structures while keeping these structures flexible and efficient. Facade pattern and decorator pattern are some of the patterns that fall into this category. Finally, behavioral patterns concerned themselves with algorithms and the assignment of responsibilities between objects. The majority of design patterns fall into this category, such as template method, strategy pattern, state pattern, and a lot more.

Panca [15] implemented various design patterns into three different applications. Panca [15] implemented different design patterns one by one and measured the changes in maintainability and modularity. They conclude that the more design patterns implemented in the application, the higher the application's maintainability will be. However, the level of modularity of the three applications decreases the more design patterns were implemented. In relative to this research, Panca's [15] research chooses their application based on domain. In comparison, this research chooses its application based on whether it has duplicate code problems or not.

2.4 Static Code Analyzer

Metric is an excellent way to understand, monitor, control, predict, and test software development [21]. One way to collect metrics from software is by analyzing them with a static code analyzer. A tool that analyzed source code without executing the program [2]. There are a variety of tools with different purposes. For instance, there are tools to check unit tests, dependency analysis, structural code, bug detection, and much more. For this research, the static code analyzers that are required are the ones that can detect duplicate code of at least type-1 and the ones that can measure a variety of object-oriented and traditional metrics. Object-oriented metrics measure the class and object characteristics, such as coupling, cohesion, and depth of inheritance. On the other hand, traditional metrics cover a broader range of metrics such as lines of codes and cyclomatic complexity [17].

2.5 Duplicate Code

Duplicate code, also known as code clone, are two or more pieces of code that have similarities [4] in terms of syntax or functionality [10]. The primary cause duplicate code appears is due to code reuse from one part of the application to another part [14]. Duplicate code can have several consequences for applications, such as decreased maintainability and increased maintenance costs.

There are four categories of duplicate code [3]: type-1, type-2, type-3, and type-4. Type-1 duplicate codes have similar code fragments [3]. However, there might be some variation in white spaces, comments, or layout [3]. Type-2 duplicate codes have the same code as the original, but with possible variations in the variable name, constants, class name, and more [3]. Type-3 duplicate codes are codes with its statement changed, added, or deleted [3]. Finally, type-4 duplicate codes have the same functionality but have different syntax [3].

2.6 Metric for Measuring Duplicate Code

The duplicate code measurement metric that was used was based on the scientific research by Barbosa [3] and Heitlager [8]. In both studies, counting the number of LOC (lines of code) was used to measure duplicate code. In this study, a static code analyzer was used to measure the lines of duplicate code. The lower the duplicate code, the better the application will be in terms of duplicate code metric.

2.7 Metric for Measuring Maintainability

Heitlager [8] assessed the maintainability index, a metric to measure software maintainability based on Halstead volume, cyclomatic complexity, and lines of codes, to be an ineffective way to measure maintainability. They argued that it was difficult to know the reason why the maintainability index change. Therefore, they designed a new maintainability measurement system that improves upon what lacks in the maintainability index. Instead of creating an equation, they use the maintainability definition from ISO 9126 to break maintainability into smaller components: analyzability, changeability, stability, and testability. These components are then broken down further into metrics that can be easily measured. For example, the analyzability component was made up of volume, duplication, unit size, and unit testing. That way, developers know exactly which part they need to improve on to increase the quality of specific components. However, because there is a new maintainability standard, namely ISO 25010, the component for this research was adjusted into modularity, reusability, analyzability, modifiability, and testability.

Due to the change in ISO reference, some maintainability components were not discussed in Heitlager's research. Those components are modularity, modifiability, and reusability. For all of the components that were not discussed, other papers were referenced to complete the maintainability measurement system. The modularity component was referenced from Emanuel [5], in which they use coupling and cohesion to represent the component. The modifiability component was referenced from Harun [7], in which they also use coupling and cohesion. Finally, the reusability component was referenced from Papamichail [16], which states that reusability is assessed from complexity, cohesion, coupling, inheritance, documentation, and unit size. Furthermore, the original ranking system will no longer be accurate, as there are metrics that were not discussed previously. Thus, the ranking system was replaced with trend analysis.

Table 1. A modified maintainability measurement system with ISO 25010

	Components	Metrics									
		Coupling	Cohesion	Complexity	Inheritance	Unit Size	Volume	Duplication	Unit Testing	# of Methods	Documentation
ISO 25010 Maintainability	Modularity		X			X				X	
	Reusability	X	X	X	X	X					X
	Analyzability					X	X	X	X		
	Modifiability	X	X								
	Testability			X		X			X		

By analyzing the difference before and after the design pattern is implemented, it is possible to gain insights into which specific metrics and components the design pattern effect. The following are descriptions of the maintainability category along with its related metrics:

Modularity

Modularity is the degree to which a system or computer program is composed of distinct components [9]. Emanuel [5] stated that modularity is the internal quality attribute of the software system. Furthermore, modularity is assessed from the total amount of non-comment lines of code, cohesion, and number of methods, according to Emanuel [5].

Reusability

Reusability is the degree to which an asset can be used in more than one system or in building other assets [9]. Papamichail [16] stated that the level of reusability is assessed from coupling, cohesion, complexity, inheritance, documentation, and unit size. However, the documentation part of the metric will not be utilized since the implementation of the design pattern does not change any part of the documentation for the application. The value of documentation will always be constant in this study.

Analyzability

Analyzability is the degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts [9]. Based on Heitlager [8], the level of analyzability of a component is assessed by volume, duplicate code, unit size, and unit testing. The unit testing metric will not be included in the result, as the value will always decrease in this study due to the newly created

abstract classes and methods will not be covered in the original unit test by the original developers.

Modifiability

Modifiability is the degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality [9]. According to Harun [7], the modifiability of a component is assessed from coupling and cohesion.

Testability

Testability is the degree of effectiveness and efficiency with which test criteria can be established for a system [9]. Based on Heitlager [8], testability can be assessed from complexity, unit size, and unit testing. The unit testing metric in this component will also be excluded for the same reason as in the analyzability component.

3. Experiment

3.1 Methodology

Below is the flowchart representing the methodology that was followed in this study. In addition, a detailed explanation for the methodology can be found below the flowchart:

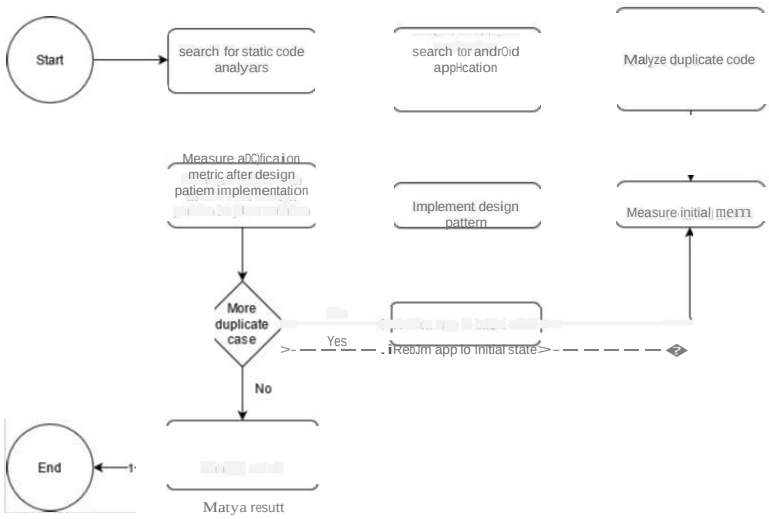


Fig 1. Methodology Flowchart

3.1.1 Determining Static Code Analyzers

Static code analyzers that can detect and measure duplicate code of at least type-1, object-oriented metric, and traditional metric are required for this study. Based on Lenarduzzi's [12] selection of static code analyzer, sonarQube was the analyzer that fits the requirement to detect and measure duplicate code. The rest were unable to detect duplicate code, outdated, incompatible with the android java version, or has only a narrow scope of detecting duplicate code. On the other hand, the MetricsReloaded plugin, a static code analyzer tool used by Saifan [18], was used to measure both object-oriented and traditional metrics due to its ability to measure all the required metrics for this study.

3.1.2 Determining Android Application

Seventeen apps were analyzed using sonarQube to determine the number of duplicate codes on each of the applications. The amount of duplicate codes ranges from 0% to 7.9% between the 17 applications. An open-source expense tracking app with 21 contributors, 261 commits, and 6.3% duplicate code known as MoneyWallet was chosen as the study case for this study. Unfortunately, due to limited computational power, the top app in this selection with the highest duplicate code could not be analyzed.

3.1.3 Analyze Duplicate Code

The duplicate code analysis from sonarQube was then turned into a diagram to visualize the relationship between files better. As seen on fig. 2 and fig. 3, each table represents a file with a list of duplicate code ranges in said file. The line that connects one table to another means that they share the same duplicate codes.

All of the identified duplicate code was then broken down further to gain insights on what code was the most duplicated in the app. There were six types of duplicate code found: duplicate variable, duplicate method, duplicate partial method, duplicate interface, duplicate class, and duplicate enum. Duplicate method and duplicate partial method made up 87.9% of the duplicate code type. Since those two type made up a large percentage of the duplicate code problem, duplicate method and partial method were the ones that was further analyzed.

The GOF book of design patterns provides a variety of approaches to determine which design pattern to use. Since this study only focuses on duplicate code problems, three approaches were chosen to fit the scope and requirement of this study. Those approaches were identifying design problems, matching design problems with design pattern intent, and identify what varies on the design problem. Only those duplicate code with a design problem that can be solved with a design pattern will be solved.

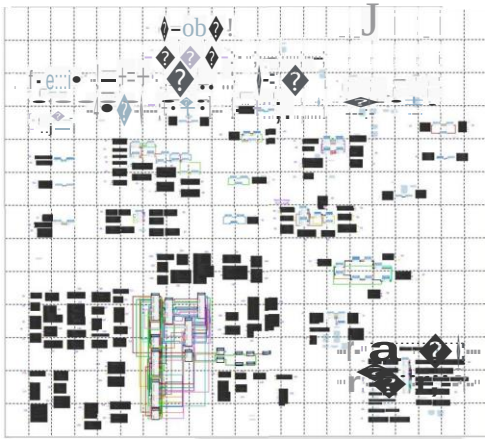


Fig 2. Duplicate Code Diagram

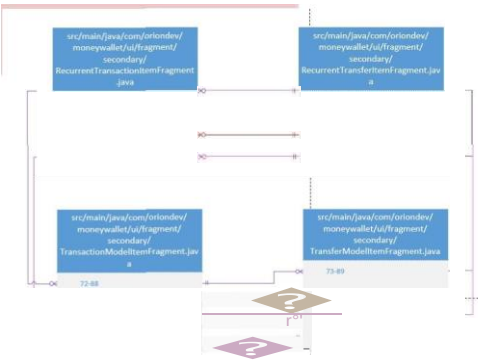


Fig 3. Duplicate Code Diagram Closeup

3.1.4 Design Pattern Implementation

For this research, there will only be one design pattern solution that can exist and be implemented at a time. As there were various duplicate cases that were able to be implemented with the template pattern, this ensures there will be no insights from individual duplicate codes that will be overshadowed. Furthermore, to ensure the implementation of the design pattern is as objective as possible, the refactoring process was only based on the duplicate code detected by sonarQube.

For instance, the grey line on the left-hand side of fig. 4 represents the code detected as duplicate by SonarQube. Two types of duplicate codes were involved, a duplicate method (line 64-72) and a duplicate partial method (line 74-77). The duplicate method could not be implemented with a design pattern, as it contains a private variable. On the other hand, the duplicate partial method was able to be implemented with a template pattern. It contains a variant part unique to each subclass and an invariant part that all of the subclasses have in common. Even though it contains private variables, it was located in the variant part of the pattern; thus, there is no need to modify its access modifier. To keep this study as objective as possible, only the invariant part detected by SonarQube was moved to the abstract class. Thus, the super syntax and the branch were moved to the abstract class, while the codes inside and after the branch were made into individual steps delegated to each of the subclasses. fig. 5 and fig. 6 represent the before and after template pattern was implemented. The abstract class on fig. 7 consists of the method that contains the duplicate code, the duplicate code itself, and also three empty abstract methods (steps). The abstract class define the skeleton of an algorithm in a method and implement the invariant part of an algorithm [6]. Furthermore, fig. 8 displays the class diagram when the template pattern is implemented. Similar class diagram structures can also be observed across all of the classes that could be implemented with the template pattern. The three classes on the bottom of fig. 8 are the concrete classes with duplicate codes, while the class they inherit from is the abstract class. A line can also be observed going out of the abstract class, which points to the original parent class the concrete class extends from, which now the abstract class extends from. By incorporating inheritance in its implementation, the template pattern is able to accomplish two tasks. First, it is able to delegate

the implementation of the variant steps to the concrete classes. Second, it allows the concrete classes to inherit the common method and steps.

Moreover, the study case that was used in this research was an android mobile application. Most of the classes that implemented the template pattern were not positioned on top of the class hierarchy. Thus, causing the abstract class to be positioned almost at the bottom of the hierarchy. However, design patterns are concepts and are not an algorithm [20]. As long as the abstract class defines an algorithm and delegates variant steps to the subclass, it can be classified as a template pattern.

Fig 4. SonarQube GUI showing the duplicate code

Fig 5. Before design pattern was implemented

Fig 6. After design pattern was implemented

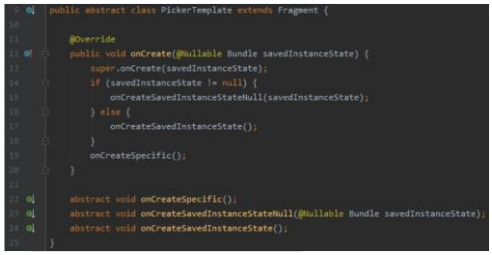


Fig 7. The created abstract class due to the implementation of template pattern

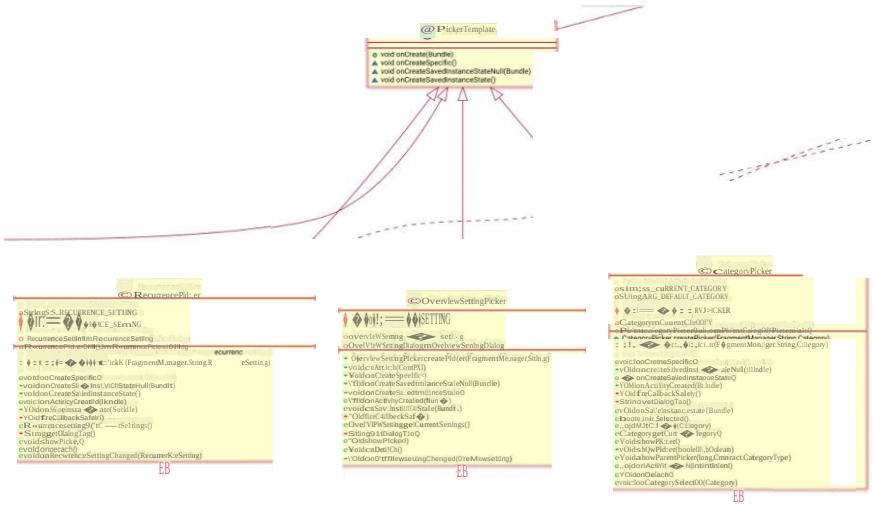


Fig 8. Class Diagram of Template Pattern Implementation

3.1.5 Analyze Result

All of the metric values were then be compared from before and after implementing the design pattern. A trend from each metric will then be derived and analyzed to explore the reason behind their changes and how it affects the duplicate code and maintainability metrics of the application. The result is further discussed in the next section.

4. Evaluation

4.1 Result

Table 2 represent each of the maintainability components along with its metrics. There were a total of fourteen groups of duplicate code that could be implemented with a design pattern. A total of thirty-four classes were involved between the fourteen groups. The three columns represent the condition of the class: improve (I), stagnant (S), and worsen (W). For example, in the number of methods column in the modularity table, there were twenty-six classes with the same number of methods after implementing the design pattern (stagnant), and eight classes experienced an increase in the number of methods (worsen).

Table 2. Maintainability Result

(a) Modularity Component

Modularity (# of Files)								
NCLOC			LCOM			# of Methods		
I	S	W	I	S	W	I	S	W
34	0	0	0	32	2	0	26	8

(b) Reusability Component

Reusability (# of Files)														
CBO			LCOM			WMC			DIT			NCLOC		
I	S	W	I	S	W	I	S	W	I	S	W	I	S	W
0	0	34	0	32	2	16	8	10	0	0	34	34	0	0

(c) Analysabaility Component

Analysability (# of Files)								
Total NCLOC			Duplicate Code			NCLOC		
I	S	W	I	S	W	I	S	W
0	0	34	34	0	0	34	0	0

(d) Modifiability Component

Modifiability (# of Files)					
CBO			LCOM		
I	S	W	I	S	W
0	0	34	0	32	2

(e) Testability Component

Testability (# of Files)					
WMC			NCLOC		
I	S	W	I	S	W
16	8	10	34	0	0

The unit size metric (NCLOC) showed an improvement across all of the duplicate case, which positively impact the modularity, reusability, analysability, and testability components. Likewise, the duplicate code metric showed an improvement on all of the classes that has duplicate codes and can be implemented with a design pattern, which positively impact the analysability metric. On the other hand, the volume (Total NCLOC) negatively effect the analysability component across all of the classes. Similarly, the coupling (CBO) metric negatively impact the reusability and modifiability component while the depth of inheritance metric (DIT) negatively impact the reusability component for all of the classes. The implementation of template pattern did not affect the number of methods metric in most classes, except for the eight classes that experienced an increase in the number of methods, which negatively impact the modularity component. The same behaviour can also be observed with the cohesion metric (LCOM), in which the modularity, reusability, and modifiability component were mostly unaffected except for two of the classes, in which they worsen. Lastly, the complexity metric (WMC) showed a mixture of effects which impacted the reusability and testability metric.

4.2 Analysis

4.2.1 Duplicate Code Analysis

After finishing the analysis on the identified duplicate codes, it was discovered that only the template pattern could be implemented. The limit sonarQube has on detecting duplicate codes and technical limitation were the two factors why template pattern was the only design pattern able to be implemented. First, sonarQube only considers a piece of code to be duplicated when at least ten consecutive lines are identical. This setting could not be modified and was an indication that sonarQube was only able to detect type-1 duplicate codes. SonarQube's duplicate code requirement limits the ability to detect duplicate method names. This creates a challenge to identify classes with the same method name but with different implementation, a design problem that the factory method pattern and strategy pattern could potentially solve. Even if other classes and methods that were not detected as duplicates

were analyzed, there were still restraints that prevent the implementation of the design pattern in general. Those restraints are further described below.

The majority of the identified duplicate code by sonarQube were unable to be implemented by a design pattern. The reason why the majority of duplicate code were not able to incorporate a design pattern in their designs are as follows:

a).Duplicated Private Method

There were private methods that were identified as duplicate code by sonarQube. Design patterns such as factory method patterns and strategy patterns handle duplicate methods by moving them to a separate class. Moving the method to any other class or changing the access modifier would give access to other classes to utilize the method, which would defeat the purpose of making the method private in the first place. Private methods are only allowed to be utilized by their own class.

b).Method Containing Private Variable

Some duplicated methods and duplicated partial methods contain private variables. By abstracting the method or partial method to a parent class, the private variable would have to be abstracted. Moving the variable or changing its access modifier would defeat the purpose of making the variable private in the first place. Implementing a design pattern such as template pattern, factory method pattern, or strategy pattern would require moving the private variable or manipulate the variable's access modifier.

c).Extend Different Parent Class

Some of the classes that contain duplicated methods or duplicated partial methods extend from a different parent class. Design patterns such as template patterns and factory method patterns require the classes with duplicate code to share the same parent class. If an abstract class was to be created, either for the implementation of template pattern or factory method pattern, the classes with the duplicate codes will have to extend from the abstract class. In turn, the abstract class would have to extend from the initial parent class of the classes with the duplicate codes. If the classes extend from different parent classes, then the abstract class would have to extend from two different parent classes simultaneously, which is prohibited by the Java language.

d).Complex Duplicate Code

Some of the partial method duplicates were a part of a complex method with many nested branches. The codes that were identified as duplicates were part and also inside various nested branches. Causing them to be unable to be abstracted and made into their method. Duplicate codes that contain partial branches prevent the implementation of the template pattern.

4.2.2 Design Pattern Effect

The implementation of template pattern on duplicate codes had caused a positive impact on the amount of duplicate codes. However, there were mixed result regarding the effect it had on the application's maintainability aspect as a whole. Though, not only did the characteristic of the template pattern had influenced the value of the metrics, the duplicate code that was being implemented on also had a role in affecting some of the value of the metrics. The depth of inheritance, coupling, unit size, duplicate code, cohesion, volume, complexity, and number of methods metrics were affected differently, which impacted all of the maintainability components in various manner.

Due to the creation of an abstract class every time a template pattern wanted to be implemented, there was an increase in the depth of inheritance metric that caused the metric to worsen. By extending an abstract class, the length from the class that had duplicate codes to its root class increases. Therefore, the reusability component will always be negatively impacted by implementing the template pattern.

Similarly, the coupling metric was also affected negatively due to the implementation of the template pattern. By creating an abstract class, it creates dependency between the abstract class and the class with duplicate codes, since the class with duplicate codes has to extend to the abstract class. This unavoidable behaviour of the template pattern will negatively impact the reusability and the modifiability components.

The volume metric was also observed to increase in all of the classes that implemented the template pattern. The increase in volume was caused by the added lines of code needed to create an abstract class. In all of the duplicate case, the amount of duplicate code abstracted was less than the amount of lines of code added. The increase in the volume metric negatively affect the analysability component.

On the other hand, the unit size metric was observed to always improve on all of the classes that implemented the template pattern. The pattern make use of the inheritance feature, in which a default implementation can be

inherited to all of the subclass that extends from the abstract class. In this case, the duplicate code was treated as a default implementation, hence it was moved from individual subclasses to the abstract class. Since the duplicate codes were moved to the abstract class, there were less codes in the class that experienced the duplicate code issue. The modularity, reusability, analysability, and testability components were all improved due to the reduced code in each of the classes that implemented the template pattern.

Additionally, the duplicate code metric was also observed to decrease on all of the classes that implemented the template pattern. When it comes to the template pattern, the code that is abstracted is considered to be invariant, while the code that each of the subclasses implement is considered to be variant. The duplicate code is considered to be the invariant part, as it is the code that is constant among the subclasses. Due to the nature of the template pattern, there will always be less duplicate codes in classes that implement the template pattern. As a result, the decreasing amount of duplicate codes will always positively impact the analysability component of maintainability.

The cohesion metric mostly showed a constant value before and after the implementation of template pattern, except for one duplicate case. MetricsReloaded classify related methods by the amount variables they share and if one methods calls on another method. The plugin then measure cohesion by calculating the total number of components in the method relation graph. In most cases, the implementation of template pattern did not effect the cohesion metric, as it did not increase or decrease the amount of shared variable and the amount of calls a method makes to another method. The duplicate case in question was a switch statement containing two cases, in which the switch statement was abstracted to the abstract class while the two cases were made into two separate methods in the subclass. Both of the cases called the same method, which causes the plugin to decrease the cohesion level of the class since there were two different methods calling another method instead of one method calling another method twice.

The complexity metric was the only metric that experienced all of the possible conditions in this study. Sixteen classes were less complex after the implementation because the code that was abstracted decreased the amount of possible path a method can take. Causing the total amount of complexity of the class to decrease. Eight classes were unaffected due the the code that was being abstracted did not contribute to the number of path a method can take in the first place. Thus, moving them to the abstract class showed no affect to the total amount of complexity of the class. Ten classes were more complex after implementing the template pattern because the original method was divided into a lot of smaller methods. This caused the complexity of each method to be lower, but it increases the total amount of complexity of the class.

The majority of classes that implemented the template pattern did not experienced any change in their number of methods metric, except for eight classes that experience an increase in the total number of methods. When implementing the template pattern, the method that contained duplicate codes were divided into two parts: invariant and variant part. In most cases, there were only one invariant and variant part. In which each of the part were made into its own method and implemented on either the abstract class or the subclass. For the eight classes, they were observed to contain more than one variant part. Which caused the subclass to contain more method than it originally had.

5. Conclusion

The implementation of the design pattern, particularly the template pattern, was able to decrease the amount of duplicate codes in all of the duplicate case. However, it was not able to maintain or increase the maintainability aspect of the application as a whole. First, The unit size and the duplicate code metric will always improve due to abstraction of the invariant part to the abstract class. Second, the depth of inheritance and the coupling metric will always worsen due to the use of inheritance by the template pattern. Third, the volume metric was also observed to always worsen in this study, due to the amount of code removed was less than the amount of code added to create an abstract class. Fourth, the effect the cohesion metric have on the application is determined by the codes that is being abstracted to the abstract class. Fifth, the complexity metrics is reliant on if the abstracted code contributed to the amount of path a method can take and how many smaller methods will be created. Finally, the number of methods metric is dependent on how many method implementation will be delegated to the subclass. In future work, investigating the effect other design patterns have on duplicate codes and its subsequent effects on the maintainability aspect might prove necessary. Using a mobile application with a more significant amount of duplicate codes might widen the chance of other design patterns being implemented and the possibility of more insight being discovered. Lastly, creating a metric that determines the value of implementing a design pattern on duplicate code will prove useful, as developers can use it to consider weather it is best to implement a design pattern.

References

[1]A. Aldayel and K. Alnafjan. Challenges and best practices for mobile application development. In Proceedings of the International Conference on Compute and Data Analysis, pages 41–48, 2017.

[2]Q. Ashfaq, R. Khan, and S. Farooq. A comparative analysis of static code analysis tools that check java code adherence to java coding standards. In 2019 2nd International Conference on Communication, Computing and Digital systems (C-CODE), pages 98–103. IEEE, 2019.

[3]F. S. Barbosa and A. Aguiar. Removing code duplication with roles. In 2013 IEEE 12th International Conference on Intelligent Software Methodologies, Tools and Techniques (SoMeT), pages 37–42. IEEE, 2013.

[4]X. Chen, A. Y. Wang, and E. Tempero. A replication and reproduction of code clone detection studies. In Proceedings of the Thirty-Seventh Australasian Computer Science Conference-Volume 147, pages 105–114, 2014.

[5]A. W. R. Emanuel, R. Wardoyo, J. E. Istiyanto, and K. Mustofa. Modularity index metrics for java-based open source software projects. arXiv preprint arXiv:1309.5689, 2013.

[6]E. Gamma, R. Helm, R. Johnson, J. Vlissides, and D. Patterns. Elements of reusable object-oriented software. Design Patterns. massachusetts: Addison-Wesley Publishing Company, 1995.

[7]F. B. HARUN. Review of ios architectural pattern for testability, modifiability, and performance quality. Journal of Theoretical and Applied Information Technology, 97(15), 2019.

[8]I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In 6th international conference on the quality of information and communications technology (QUATIC 2007), pages 30–39. IEEE, 2007.

[9]ISO. Iso/iec 25010:2011(en) systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models.

[10]E. Juergens, F. Deissenboeck, and B. Hummel. Code similarities beyond copy & paste. In 2010 14th European Conference on Software Maintenance and Reengineering, pages 78–87. IEEE, 2010.

[11]C. J. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. Empirical Software Engineering, 13(6):645–692, 2008.

[12]V. Lenarduzzi, A. Sillitti, and D. Taibi. A survey on code analysis tools for software maintenance prediction. In International Conference in Software Engineering for Defence Applications, pages 165–175. Springer, 2018.

[13]I. Malavolta, R. Verdecchia, B. Filipovic, M. Bruntink, and P. Lago. How maintainability issues of android apps evolve. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 334–344. IEEE, 2018.

[14]A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto. Software quality analysis by code clones in industrial legacy software. In Proceedings Eighth IEEE Symposium on Software Metrics, pages 87–94. IEEE, 2002.

[15]B. S. Panca, S. Mardiyanto, and B. Hendradjaya. Evaluation of software design pattern on mobile application based service development related to the value of maintainability and modularity. In 2016 International Conference on Data and Software Engineering (ICoDSE), pages 1–5. IEEE, 2016.

[16]M. D. Papamichail, T. Diamantopoulos, and A. L. Symeonidis. Measuring the reusability of software components using static analysis metrics and reuse rate information. Journal of Systems and Software, 158:110423, 2019.

[17]D. Rodriguez and R. Harrison. An overview of object-oriented design metrics. 2001.

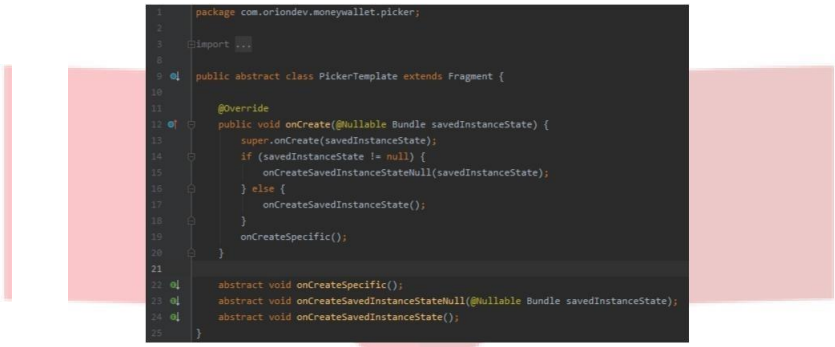
[18]A. A. Saifan and A. Al-Rabadi. Evaluating maintainability of android applications. In 2017 8th International Conference on Information Technology (ICIT), pages 518–523. IEEE, 2017.

[19]F. E. Shahbudin and F.-F. Chua. Design patterns for developing high efficiency mobile application. Journal of Information Technology & Software Engineering, 3(3):1, 2013.

[20]A. Shvets. What's a design pattern?

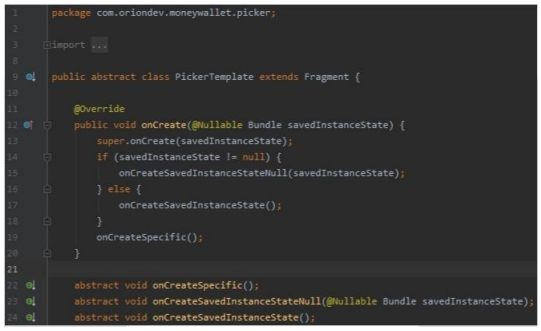
[21]P. Tomas, M. J. Escalona, and M. Mejias. Open source tools for measuring the internal quality of java software products. a survey. Computer Standards & Interfaces, 36(1):244–255, 2013.

Supplements



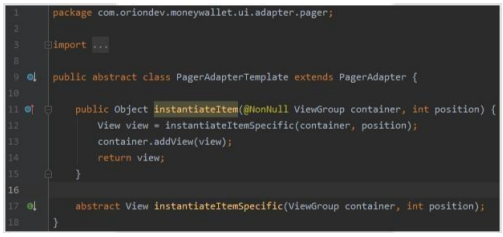
```
package com.oriondev.moneywallet.picker;

import ...

public abstract class PickerTemplate extends Fragment {

    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (savedInstanceState != null) {
            onCreateSavedInstanceStateNull(savedInstanceState);
        } else {
            onCreateSavedInstanceState();
        }
        onCreateSpecific();
    }

    abstract void onCreateSpecific();
    abstract void onCreateSavedInstanceStateNull(@Nullable Bundle savedInstanceState);
    abstract void onCreateSavedInstanceState();
}
```

Fig 9. Template Pattern Group 3



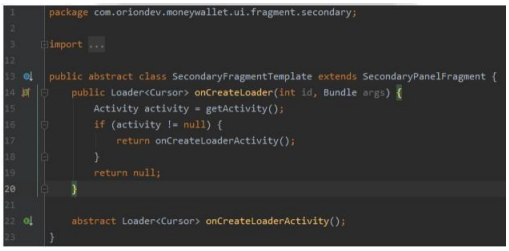
```
package com.oriondev.moneywallet.picker;

import ...

public abstract class PickerTemplate extends Fragment {

    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (savedInstanceState != null) {
            onCreateSavedInstanceStateNull(savedInstanceState);
        } else {
            onCreateSavedInstanceState();
        }
        onCreateSpecific();
    }

    abstract void onCreateSpecific();
    abstract void onCreateSavedInstanceStateNull(@Nullable Bundle savedInstanceState);
    abstract void onCreateSavedInstanceState();
```

Fig 10. Template Pattern Group 4



```
package com.oriondev.moneywallet.ui.adapter.pager;

import ...

public abstract class PagerAdapterTemplate extends PagerAdapter {

    public Object instantiateItem(@NonNull ViewGroup container, int position) {
        View view = instantiateItemSpecific(container, position);
        container.addView(view);
        return view;
    }

    abstract View instantiateItemSpecific(ViewGroup container, int position);
}
```

Fig 11. Template Pattern Group 31



```
package com.oriondev.moneywallet.ui.fragment.secondary;

import ...

public abstract class SecondaryFragmentTemplate extends SecondaryPanelFragment {
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
        Activity activity = getActivity();
        if (activity != null) {
            return onCreateLoaderActivity();
        }
        return null;
    }

    abstract Loader<Cursor> onCreateLoaderActivity();
}
```

Fig 12. Template Pattern Group 51

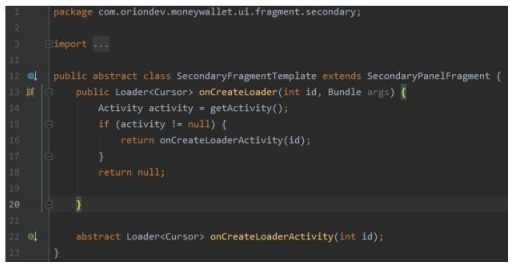


Fig 13. Template Pattern Group 54

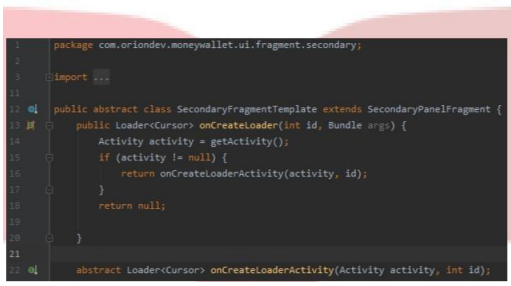


Fig 14. Template Pattern Group 57

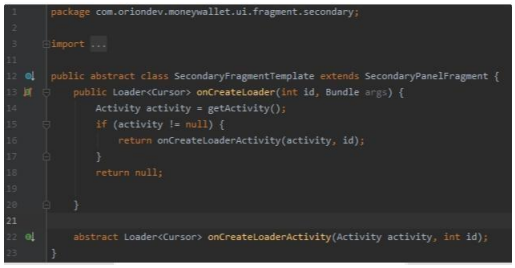


Fig 15. Template Pattern Group 58

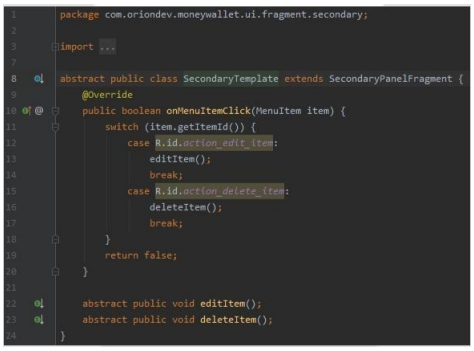


Fig 16. Template Pattern Group 63

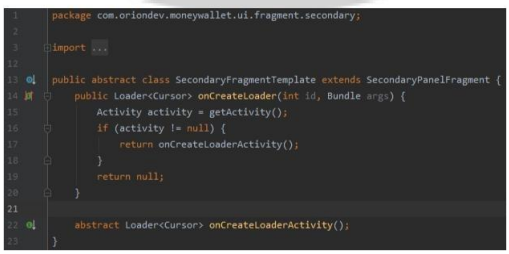


Fig 17. Template Pattern Group 64

```
package com.oriondev.moneywallet.ui.fragment.secondary;

import ...

public abstract class SecondaryFragmentTemplate extends SecondaryPanelFragment {
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
        Activity activity = getActivity();
        if (activity != null) {
            return onCreateLoaderActivity();
        }
        return null;

    }

    abstract Loader<Cursor> onCreateLoaderActivity();
}
```

Fig 18. Template Pattern Group 67

```
package com.oriondev.moneywallet.ui.activity;

import android.os.Bundle;

abstract public class ActivityTemplate extends NewEditItemActivity {

    protected void onViewCreated(Bundle savedInstanceState) {
        super.onViewCreated(savedInstanceState);
        onViewCreatedSpecific(savedInstanceState);
    }

    abstract void onViewCreatedSpecific(Bundle savedInstanceState);
}
```

Fig 19. Template Pattern Group 108

```
package com.oriondev.moneywallet.ui.activity;

import android.os.Bundle;

public abstract class ActivityTemplate extends NewEditItemActivity {
    protected void onViewCreated(Bundle savedInstanceState) {
        super.onViewCreated(savedInstanceState);
        onViewCreatedSpecific(savedInstanceState);
    }

    protected abstract void onViewCreatedSpecific(Bundle savedInstanceState);
}
```

Fig 20. Template Pattern Group 124

```
package com.oriondev.moneywallet.ui.activity;

import android.os.Bundle;

public abstract class ActivityTemplate extends NewEditItemActivity {
    protected void onViewCreated(Bundle savedInstanceState) {
        super.onViewCreated(savedInstanceState);
        onViewCreatedSpecific(savedInstanceState);
    }

    protected abstract void onViewCreatedSpecific(Bundle savedInstanceState);
}
```

Fig 21. Template Pattern Group 142

```
package com.oriondev.moneywallet.ui.activity;

import ...

public abstract class ActivityClass extends NewEditItemActivity {
    protected void onViewCreated(Bundle savedInstanceState) {
        super.onViewCreated(savedInstanceState);
        long moneyFrom = 0L;
        double conversionRate = 0D;
        long tax = 0L;
        Wallet walletFrom = null;
        Wallet walletTo = null;
        onViewCreatedSpecific(savedInstanceState, moneyFrom, conversionRate, tax, walletFrom, walletTo);
    }

    abstract protected void onViewCreatedSpecific(Bundle savedInstanceState, long moneyFrom, double conversionRate, long tax, Wallet walletFrom, Wallet walletTo);
}
```

Fig 22. Template Pattern Group 143

Table 3. Code Duplication Type

Type	Amount	Percentage
Duplicate Variable	37	10.6%
Duplicate Method	152	43.8%
Duplicate Partial Method	153	44.1%
Duplicate Interface	2	0.6%
Duplicate Class	1	0.3%
Duplicate Enum	2	0.6%
Total	347	100%

Table 4. A Detailed Modularity Result

Duplicate Group	File	Modularity					
		NCLOC		LCOM		# of Methods	
		Before	After	Before	After	Before	After
3	.../picker/CategoryPicker.java	108	107	1	1	16	18
	.../picker/DateTimePicker.java	124	123	1	1	16	18
	.../picker/OverviewSettingPicker.java	116	115	2	2	11	13
	.../picker/RecurrencePicker.java	80	79	1	1	11	13
4	.../picker/BudgetTypePicker.java	81	80	1	1	12	14
	.../picker/ImportExportFormatPicker.java	82	81	2	2	13	15
31	.../pager/BarChartViewPagerAdapter.java	64	62	4	4	6	6
	.../pager/PieChartViewPagerAdapter.java	36	34	4	4	6	6
51	.../secondary/CategoryItemFragment.java	178	173	4	4	11	11
	.../secondary/EventItemFragment.java	129	124	4	4	6	6
	.../secondary/PersonItemFragment.java	121	116	4	4	6	6
54	.../secondary/TransactionItemFragment.java	254	250	6	6	13	13
	.../secondary/TransferItemFragment.java	254	250	6	6	13	13
57	.../secondary/DebtItemFragment.java	274	270	4	4	13	13
	.../secondary/SavingItemFragment.java	228	223	4	4	13	13
	.../secondary/WalletItemFragment.java	208	203	4	4	13	13
58	.../secondary/DebtItemFragment.java	274	270	4	4	13	13
	.../secondary/WalletItemFragment.java	208	203	4	4	13	13
63	.../secondary/TransactionModelItemFragment.java	170	164	5	6	11	12
	.../secondary/TransferModelItemFragment.java	180	174	5	6	11	12
64	.../secondary/TransactionModelItemFragment.java	170	165	5	5	11	11
	.../secondary/TransferModelItemFragment.java	180	175	5	5	11	11
67	.../secondary/RecurrentTransactionItemFragment.java	177	171	5	5	11	11
	.../secondary/RecurrentTransferItemFragment.java	187	181	5	5	11	11
108	.../activity/NewEditTransactionActivity.java	916	914	3	3	23	23
	.../activity/NewEditTransferActivity.java	697	695	3	3	20	20
124	.../activity/NewEditRecurrentTransactionActivity.java	358	356	1	1	12	12
	.../activity/NewEditRecurrentTransferActivity.java	421	419	1	1	12	12
142	.../activity/NewEditRecurrentTransactionActivity.java	358	356	1	1	12	12
	.../activity/NewEditRecurrentTransferActivity.java	421	419	1	1	12	12
	.../activity/NewEditTransactionModelActivity.java	331	329	1	1	11	11
	.../activity/NewEditTransferModelActivity.java	394	392	1	1	12	12
143	.../activity/NewEditRecurrentTransferActivity.java	421	414	1	1	12	12
	.../activity/NewEditTransferModelActivity.java	394	387	1	1	11	11

Table 5. A Detailed Reusability Result

Duplicate Group	File	Reusability									
		CBO		LCOM		WMC		DIT		NCLOC	
		Before	After	Before	After	Before	After	Before	After	Before	After
3	.../picker/CategoryPicker.java	12	13	1	1	25	26	2	3	108	107
	.../picker/DateTimePicker.java	11	12	1	1	30	31	2	3	124	123
	.../picker/OverviewSettingPicker.java	9	10	2	2	24	25	2	3	116	115
	.../picker/RecurrencePicker.java	6	7	1	1	18	19	2	3	80	79
4	.../picker/BudgetTypePicker.java	6	7	1	1	18	19	2	3	81	80
	.../picker/ImportExportFormatPicker.java	6	7	2	2	20	21	2	3	82	81
31	.../pager/BarChartViewPagerAdapter.java	7	8	4	4	10	10	2	3	64	62
	.../pager/PieChartViewPagerAdapter.java	6	7	4	4	9	9	2	3	36	34
51	.../secondary/CategoryItemFragment.java	20	21	4	4	29	28	3	4	178	173
	.../secondary/EventItemFragment.java	19	20	4	4	20	19	3	4	129	124
	.../secondary/PersonItemFragment.java	17	18	4	4	20	19	3	4	121	116
54	.../secondary/TransactionItemFragment.java	29	30	6	6	37	36	3	4	254	250
	.../secondary/TransferItemFragment.java	25	26	6	6	37	36	3	4	254	250
57	.../secondary/DebtItemFragment.java	24	25	4	4	42	41	3	4	274	270
	.../secondary/SavingItemFragment.java	23	24	4	4	35	34	3	4	228	223
	.../secondary/WalletItemFragment.java	21	22	4	4	31	30	3	4	208	203
58	.../secondary/DebtItemFragment.java	24	25	4	4	42	41	3	4	274	270
	.../secondary/WalletItemFragment.java	21	22	4	4	31	30	3	4	208	203
63	.../secondary/TransactionModelItemFragment.java	18	19	5	5	23	22	3	4	170	164
	.../secondary/TransferModelItemFragment.java	18	19	5	5	24	23	3	4	180	174
64	.../secondary/TransactionModelItemFragment.java	18	19	5	5	23	22	3	4	170	165
	.../secondary/TransferModelItemFragment.java	18	19	5	5	24	23	3	4	180	175
67	.../secondary/RecurrentTransactionItemFragment.java	18	19	5	5	22	21	3	4	177	171
	.../secondary/RecurrentTransferItemFragment.java	18	19	5	5	23	22	3	4	187	181
108	.../activity/NewEditTransactionActivity.java	64	65	3	3	138	138	14	15	916	914
	.../activity/NewEditTransferActivity.java	52	53	3	3	94	94	14	15	697	695
124	.../activity/NewEditRecurrentTransactionActivity.java	38	39	1	1	50	50	14	15	358	356
	.../activity/NewEditRecurrentTransferActivity.java	36	37	1	1	57	57	14	15	421	419
142	.../activity/NewEditRecurrentTransactionActivity.java	38	39	1	1	50	50	14	15	358	356
	.../activity/NewEditRecurrentTransferActivity.java	36	37	1	1	57	57	14	15	421	419
	.../activity/NewEditTransactionModelActivity.java	35	36	1	1	48	48	14	15	331	329
	.../activity/NewEditTransferModelActivity.java	33	34	1	1	55	55	14	15	394	392
143	.../activity/NewEditRecurrentTransferActivity.java	36	37	1	1	57	57	14	15	421	414
	.../activity/NewEditTransferModelActivity.java	33	34	1	1	55	55	14	15	394	387

Table 6. A Detailed Analysability Result

Duplicate Group	File	Analysability					
		Total System NCLOC		Duplicate Code		NCLOC	
		Before	After	Before	After	Before	After
3	.../picker/CategoryPicker.java	78599	78611	14	0	108	107
	.../picker/DateTimePicker.java	78599	78611	14	0	124	123
	.../picker/OverviewSettingPicker.java	78599	78611	14	0	116	115
	.../picker/RecurrencePicker.java	78599	78611	14	0	80	79
4	.../picker/BudgetTypePicker.java	78599	78618	17	0	81	80
	.../picker/ImportExportFormatPicker.java	78599	78618	17	0	82	81
31	.../pager/BarChartViewPagerAdapter.java	78599	78608	21	20	64	62
	.../pager/PieChartViewPagerAdapter.java	78599	78608	21	20	36	34
51	.../secondary/CategoryItemFragment.java	78599	78603	16	13	178	173
	.../secondary/EventItemFragment.java	78599	78603	16	13	129	124
	.../secondary/PersonItemFragment.java	78599	78603	16	13	121	116
54	.../secondary/TransactionItemFragment.java	78599	78608	129	125	254	250
	.../secondary/TransferItemFragment.java	78599	78608	129	125	254	250
57	.../secondary/DebtItemFragment.java	78599	78602	47	32	274	270
	.../secondary/SavingItemFragment.java	78599	78602	28	14	228	223
	.../secondary/WalletItemFragment.java	78599	78602	15	0	208	203
58	.../secondary/DebtItemFragment.java	78599	78601	47	32	274	270
	.../secondary/WalletItemFragment.java	78599	78601	15	0	208	203
63	.../secondary/TransactionModelItemFragment.java	78599	78610	54	52	170	164
	.../secondary/TransferModelItemFragment.java	78599	78610	54	52	180	174
64	.../secondary/TransactionModelItemFragment.java	78599	78607	54	51	170	165
	.../secondary/TransferModelItemFragment.java	78599	78607	55	51	180	175
67	.../secondary/RecurrentTransactionItemFragment.java	78599	78605	66	62	177	171
	.../secondary/RecurrentTransferItemFragment.java	78599	78605	66	62	187	181
123	.../activity/NewEditTransactionActivity.java	78599	78604	455	453	916	914
	.../activity/NewEditTransferActivity.java	78599	78604	470	468	697	695
124	.../activity/NewEditRecurrentTransactionActivity.java	78599	78604	238	233	358	356
	.../activity/NewEditRecurrentTransferActivity.java	78599	78604	312	305	421	419
142	.../activity/NewEditRecurrentTransactionActivity.java	78599	78600	238	236	358	356
	.../activity/NewEditRecurrentTransferActivity.java	78599	78600	312	310	421	419
	.../activity/NewEditTransactionModelActivity.java	78599	78600	280	278	331	329
	.../activity/NewEditTransferModelActivity.java	78599	78600	353	351	394	392
143	.../activity/NewEditRecurrentTransferActivity.java	78599	78600	312	305	421	414
	.../activity/NewEditTransferModelActivity.java	78599	78600	353	346	394	387

Table 7. A Detailed Modifiability Result

Duplicate Group	File	Modifiability			
		CBO		LCOM	
		Before	After	Before	After
3	.../picker/CategoryPicker.java	12	13	1	1
	.../picker/DateTimePicker.java	11	12	1	1
	.../picker/OverviewSettingPicker.java	9	10	2	2
	.../picker/RecurrencePicker.java	6	7	1	1
4	.../picker/BudgetTypePicker.java	6	7	1	1
	.../picker/ImportExportFormatPicker.java	6	7	2	2
31	.../pager/BarChartViewPagerAdapter.java	7	8	4	4
	.../pager/PieChartViewPagerAdapter.java	6	7	4	4
51	.../secondary/CategoryItemFragment.java	20	21	4	4
	.../secondary/EventItemFragment.java	19	20	4	4
	.../secondary/PersonItemFragment.java	17	18	4	4
54	.../secondary/TransactionItemFragment.java	29	30	6	6
	.../secondary/TransferItemFragment.java	25	26	6	6
57	.../secondary/DebtItemFragment.java	24	25	4	4
	.../secondary/SavingItemFragment.java	23	24	4	4
	.../secondary/WalletItemFragment.java	21	22	4	4
58	.../secondary/DebtItemFragment.java	24	25	4	4
	.../secondary/WalletItemFragment.java	21	22	4	4
63	.../secondary/TransactionModelItemFragment.java	18	19	5	6
	.../secondary/TransferModelItemFragment.java	18	19	5	6
64	.../secondary/TransactionModelItemFragment.java	18	19	5	5
	.../secondary/TransferModelItemFragment.java	18	19	5	5
67	.../secondary/RecurrentTransactionItemFragment.java	18	19	5	5
	.../secondary/RecurrentTransferItemFragment.java	18	19	5	5
108	.../activity/NewEditRecurrentTransactionActivity.java	64	65	3	3
	.../activity/NewEditRecurrentTransferActivity.java	52	53	3	3
124	.../activity/NewEditRecurrentTransactionActivity.java	38	39	1	1
	.../activity/NewEditRecurrentTransferActivity.java	36	37	1	1
142	.../activity/NewEditRecurrentTransactionActivity.java	38	39	1	1
	.../activity/NewEditRecurrentTransferActivity.java	36	37	1	1
	.../activity/NewEditTransactionModelActivity.java	35	36	1	1
	.../activity/NewEditTransferModelActivity.java	33	34	1	1
143	.../activity/NewEditRecurrentTransferActivity.java	36	37	1	1
	.../activity/NewEditTransferModelActivity.java	33	34	1	1

Table 8. A Detailed Testability Result

Duplicate Group	File	Testability			
		WMC		NCLOC	
		Before	After	Before	After
3	.../picker/CategoryPicker.java	25	26	108	107
	.../picker/DateTimePicker.java	30	31	124	123
	.../picker/OverviewSettingPicker.java	24	25	116	115
	.../picker/RecurrencePicker.java	18	19	80	79
4	.../picker/BudgetTypePicker.java	18	19	81	80
	.../picker/ImportExportFormatPicker.java	20	21	82	81
31	.../pager/BarChartViewPagerAdapter.java	10	10	64	62
	.../pager/PieChartViewPagerAdapter.java	9	9	36	34
51	.../secondary/CategoryItemFragment.java	29	28	178	173
	.../secondary/EventItemFragment.java	20	19	129	124
	.../secondary/PersonItemFragment.java	20	19	121	116
54	.../secondary/TransactionItemFragment.java	37	36	254	250
	.../secondary/TransferItemFragment.java	37	36	254	250
57	.../secondary/DebtItemFragment.java	42	41	274	270
	.../secondary/SavingItemFragment.java	35	34	228	223
	.../secondary/WalletItemFragment.java	31	30	208	203
58	.../secondary/DebtItemFragment.java	42	41	274	270
	.../secondary/WalletItemFragment.java	31	30	208	203
63	.../secondary/TransactionModelItemFragment.java	23	22	170	164
	.../secondary/TransferModelItemFragment.java	24	23	180	174
64	.../secondary/TransactionModelItemFragment.java	23	22	170	165
	.../secondary/TransferModelItemFragment.java	24	23	180	175
67	.../secondary/RecurrentTransactionItemFragment.java	22	21	177	171
	.../secondary/RecurrentTransferItemFragment.java	23	22	187	181
108	.../activity/NewEditTransactionActivity.java	138	138	916	914
	.../activity/NewEditTransferActivity.java	94	94	697	695
124	.../activity/NewEditRecurrentTransactionActivity.java	50	50	358	356
	.../activity/NewEditRecurrentTransferActivity.java	57	57	421	419
142	.../activity/NewEditRecurrentTransactionActivity.java	50	50	358	356
	.../activity/NewEditRecurrentTransferActivity.java	57	57	421	419
	.../activity/NewEditTransactionModelActivity.java	48	48	331	329
	.../activity/NewEditTransferModelActivity.java	55	55	394	392
143	.../activity/NewEditRecurrentTransferActivity.java	57	57	421	414
	.../activity/NewEditTransferModelActivity.java	55	55	394	387

Table 9. A Detailed Duplicate Code Result

Duplicate Group	File	Amount of Duplicate Code	
		Before	After
3	.../picker/CategoryPicker.java	14	0
	.../picker/DateTimePicker.java	14	0
	.../picker/OverviewSettingPicker.java	14	0
	.../picker/RecurrencePicker.java	14	0
4	.../picker/BudgetTypePicker.java	17	0
	.../picker/ImportExportFormatPicker.java	17	0
31	.../pager/BarChartViewPagerAdapter.java	21	20
	.../pager/PieChartViewPagerAdapter.java	21	20
51	.../secondary/CategoryItemFragment.java	16	13
	.../secondary/EventItemFragment.java	16	13
	.../secondary/PersonItemFragment.java	16	13
54	.../secondary/TransactionItemFragment.java	129	125
	.../secondary/TransferItemFragment.java	129	125
57	.../secondary/DebtItemFragment.java	47	32
	.../secondary/SavingItemFragment.java	28	14
	.../secondary/WalletItemFragment.java	15	0
58	.../secondary/DebtItemFragment.java	47	32
	.../secondary/WalletItemFragment.java	15	0
63	.../secondary/TransactionModelItemFragment.java	54	52
	.../secondary/TransferModelItemFragment.java	54	52
64	.../secondary/TransactionModelItemFragment.java	54	51
	.../secondary/TransferModelItemFragment.java	55	51
67	.../secondary/RecurrentTransactionItemFragment.java	66	62
	.../secondary/RecurrentTransferItemFragment.java	66	62
123	.../activity/NewEditTransactionActivity.java	455	453
	.../activity/NewEditTransferActivity.java	470	468
124	.../activity/NewEditRecurrentTransactionActivity.java	238	233
	.../activity/NewEditRecurrentTransferActivity.java	312	305
142	.../activity/NewEditRecurrentTransactionActivity.java	238	236
	.../activity/NewEditRecurrentTransferActivity.java	312	310
	.../activity/NewEditTransactionModelActivity.java	280	278
	.../activity/NewEditTransferModelActivity.java	353	351
143	.../activity/NewEditRecurrentTransferActivity.java	312	305
	.../activity/NewEditTransferModelActivity.java	353	346