

Design Patterns for Mobile Devices

A Comparative Study of Mobile Design Patterns using Android

Shamil Magamadov



Master Thesis submitted for the degree of
Master in Informatics: Software
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2020

Design Patterns for Mobile Devices

*A Comparative Study of Mobile Design Patterns
using Android*

Shamil Magamadov

© 2020 Shamil Magamadov

Design Patterns for Mobile Devices

<http://www.duo.uio.no/>

Printed: Representralen, University of Oslo

Abstract

Design patterns in software engineering is a useful concept for developers to solve a software design problem in a specific context. With the advent of mobile devices and mobile application development, the demand for complicated software running on mobile devices, varying in size and power capabilities, has proliferated. As design patterns were initially not designed for such an environment, mobile design patterns can help overcome the limitations and challenges with mobile application development.

This paper aims to develop mobile design patterns in Android by conducting a comparative study of design patterns for mobile devices and evaluating the design patterns in terms of performance efficiency and maintainability. Our end goal is to identify requirements imposed on design patterns by Android and its mobile devices.

The results show that with techniques and strategies for optimized memory and power usage, mobile design patterns can improve the application in terms of performance and some aspects of maintainability. Two requirements were imposed on the design patterns. One was for the patterns having to consider incorporating components from the mobile platform architecture, in our case, Android. And secondly, having to manage and optimize limited memory and power from the mobile device.

Contents

1	Introduction	1
1.1	This Project	1
1.2	Motivation	1
1.3	Problem Statement	1
1.4	Goal	2
1.5	Approach	2
1.6	Work Done	2
1.7	Evaluation Criteria	3
1.8	Results	4
1.9	Conclusion	5
1.10	Contributions	5
1.11	Limitations	5
1.12	Outline	6
2	Background	9
2.1	Introduction	9
2.2	What is a Design Pattern?	9
2.2.1	Brief History	9
2.2.2	The Essential Elements of a Design Pattern	10
2.3	Documentation	10
2.4	Why should one use Design Patterns?	11
2.5	Classification	11
2.6	Description of our Selected Design Patterns	12
2.6.1	Abstract Factory	12
2.6.2	Singleton	13
2.6.3	Command	14
2.6.4	Visitor	15
2.6.5	Strategy	16
2.6.6	Factory Method	17
2.7	Summary	18
2.8	Mobile Devices	18
2.8.1	Introduction	18
2.8.2	Main Challenges with Mobile Devices	19
2.9	Development of Mobile Applications	20
2.9.1	Introduction	20
2.9.2	The Different Approaches	20
2.9.3	Main Challenges with Mobile Application Development	20
2.10	Why do we need Mobile Design Patterns?	22

2.10.1	Examples of Mobile Design Patterns	22
2.10.2	Summary	24
2.11	Summary	24
3	Evaluation Criteria	25
3.1	Introduction	25
3.2	Quality Model	25
3.3	Analysis of the Quality Models	26
3.3.1	Criteria 1. Performance Efficiency	27
3.3.2	Criteria 2. Maintainability	27
3.4	Quality Metric Object-Oriented Model	28
3.4.1	Quality Attributes (L_1)	29
3.4.2	Design Properties (L_2)	30
3.4.3	Design Metrics (L_3)	32
3.4.4	The Mapping of L_{23}	34
3.4.5	The Mapping of L_{12}	34
3.5	The Use of QMOOD in our Comparative Study	35
3.6	Analysis of QMOOD	35
3.6.1	The Mapping of L_{23}	37
3.6.2	The Mapping of L_{12}	38
3.7	Measurement Process	38
3.7.1	Our Measurement Process of QMOOD	38
3.7.2	Our Measurement Process of ISO/IEC 25010	40
3.8	Summary	41
4	Analysis	43
4.1	Introduction	43
4.2	Mobile Environment vs. Non-Mobile Environment	43
4.2.1	Data Persistence and Application Life Cycle	43
4.2.2	Computing Perspective	45
4.2.3	Software Development Perspective	46
4.3	Mobile Design Pattern vs. Design Pattern	46
4.4	Choosing the Right Platform for our Mobile Application	47
4.5	Creating Mobile Design Patterns	47
4.5.1	Method 1: Combining Design Patterns	47
4.5.2	Method 2: Using various Optimization Techniques and Strategies	48
4.6	Summary	49
5	Case	51
5.1	Overall Architecture	51
5.2	The Main Components	53
5.3	Summary	54
6	Implementation	55
6.1	Abstract Factory vs. Mobile Abstract Factory	55
6.2	Command vs Mobile Command	58
6.3	Visitor vs. Mobile Visitor	61
6.4	Strategy vs Mobile Strategy	64
6.5	Singleton vs. Mobile Singleton	64
6.6	Summary	65

7	Results	67
7.1	QMOOD Results	67
7.1.1	Abstract Factory vs.. Mobile Abstract Factory	67
7.1.2	Command vs. Mobile Command	69
7.1.3	Visitor vs.. Mobile Visitor	71
7.1.4	Strategy vs. Mobile Strategy	73
7.1.5	Project vs.. Mobile Project	75
7.1.6	Summary	77
7.2	Android Profiler Results	77
7.2.1	Abstract Factory vs. Mobile Abstract Factory	78
7.2.2	Command vs. Mobile Command	78
7.2.3	Visitor vs. Mobile Visitor	78
7.2.4	Strategy vs. Mobile Strategy	78
7.2.5	Project vs. Mobile Project	79
7.2.6	Summary	79
7.3	Execution Time Results	80
7.3.1	Abstract Factory vs. Mobile Abstract Factory	80
7.3.2	Command vs. Mobile Command	80
7.3.3	Visitor vs. Mobile Visitor	80
7.3.4	Strategy vs. Mobile Strategy	80
7.3.5	Project vs. Mobile Project	81
7.3.6	Summary	81
7.4	Weaknesses and Mistakes of our Measurement Process	81
7.5	Summary	82
8	Discussion	83
8.1	QMOOD	83
8.2	Improved Execution Times	83
8.3	Android Profiling	85
8.4	Did Mobile Devices Impose new Requirements for Design Patterns?	86
8.5	Summary	86
9	Conclusion	89
	Appendices	91
A	Project Implementation	93
A.1	Project with Regular Design Patterns	93
A.1.1	Package; abstractFactory	93
A.1.2	Package; activity	94
A.1.3	Package; command	96
A.1.4	Package; visitor	97
A.1.5	Package; strategy	98
A.1.6	Package; enums	99
A.1.7	Package; gameObjects	100
A.1.8	Package; project	106
A.2	Project with Mobile Design Patterns	111
A.2.1	Package; mobileAbstractFactory	111
A.2.2	Package; mobileCommand	113
A.2.3	Package; mobileVisitor	114

A.2.4	Package; mobileStrategy	114
A.2.5	Package; enums	115
A.2.6	Package; imageManager	115
A.2.7	Package; mobileGameObjects	117

B Static Method vs Instance Method **119**

List of Figures

2.6.1 General UML representation of Abstract Factory	13
2.6.2 General UML representation of Singleton	14
2.6.3 General UML representation of Command	15
2.6.4 General UML representation of Visitor	16
2.6.5 General UML representation of Strategy	17
2.6.6 General UML Representation of Factory Method.	18
3.2.1 The Quality Model of ISO/IEC 25010	26
3.4.1 The Mapping of Levels in QMOOD [3]	29
4.2.1 Illustration of the activity lifecycle [1]	45
5.1.1 Flowchart of the Overall Architecture	52
6.1.1 Our Regular Abstract Factory Implementation in UML Class Diagram . . .	55
6.1.2 UML Class Diagram of Mobile Abstract Factory	57
6.1.3 Sprite Sheet of Player	58
6.2.1 UML Class Diagram of Command	59
6.2.2 UML Class Diagram of Mobile Command	60
6.3.1 UML Class Diagram of Visitor	61
6.3.2 UML Class Diagram of Mobile Visitor	63
6.4.1 UML Class Diagram of Strategy	64
6.4.2 UML Class Diagram of Mobile Strategy	64
7.1.1 Quality Attribute Graph of Abstract Factory vs. Mobile Abstract Factory .	69
7.1.2 Quality Attribute Graph of Command vs.. Mobile Command	71
7.1.3 Quality Attribute Graph of Visitor vs. Mobile Visitor	73
7.1.4 Quality Attribute Graph of Strategy vs.. Mobile Strategy	75
7.1.5 Quality Attribute Graph of Project vs. Mobile Project	77
8.5.1 Effect of Techniques and Strategies on Mobile Design Patterns	87
8.5.2 Invoking Static Method Versus Instance Method	88

List of Tables

2.5.1 Overview of Design Pattern Classification [15]	12
2.9.1 Overview over Native app Development	22
3.4.1 Quality Attribute Definitions [3]	30
3.4.2 Design Property Definitions [3]	31
3.4.3 Design Metrics Descriptions [3]	33
3.4.4 Mapping of Design Metrics to Design Properties [3]	34
3.4.5 Design Properties relationships with Quality Attributes[3].	35
3.6.1 Descriptions of our Adapted Design Metrics site MetricsReloaded	36
3.6.2 Design Metrics for Design Properties [3]	37
3.6.3 Design Properties connections with Quality Attributes[3].	38
3.7.1 Computation Formulas for Quality Attributes [3]	39
3.7.2 Example 1: Actual Metric Values for App	39
3.7.3 Example 1: Normalized Metric Values for App	40
3.7.4 Example 1: Computed Quality Attribute Values for App	40
4.5.1 Summary of Strategies for Power Conscious System [7]	49
7.1.1 Actual Metric Values of Abstract Factory vs. Mobile Abstract Factory	68
7.1.2 Normalized Metric Values of Abstract Factory vs. Mobile Abstract Factory	68
7.1.3 Actual Metric Values of Command vs. Mobile Command	70
7.1.4 Normalized Metric Values of Command vs.. Mobile Command	70
7.1.5 Actual Metric Values of Visitor vs.. Mobile Visitor	72
7.1.6 Normalized Metric Values of Visitor vs.. Mobile Visitor	72
7.1.7 Actual Metric Values of Strategy vs.. Mobile Strategy	74
7.1.8 Normalized Metric Values of Strategy vs. Mobile Strategy	74
7.1.9 Actual Metric Values of Project vs.. Mobile Project	76
7.1.10 Normalized Metric Values of Project vs.. Mobile Project	76
7.2.1 Profiling of Abstract Factory	78
7.2.2 Profiling of Command	78
7.2.3 Profiling of Visitor	78
7.2.4 Profiling of Strategy	79
7.2.5 Profiling of Project	79
7.2.6 Summary of the Total Improvement	79
7.3.1 Execution Time of Abstract Factory	80
7.3.2 Execution Time of Command	80
7.3.3 Execution Time of Visitor	80
7.3.4 Execution Time of Strategy	81
7.3.5 Execution Time of Project	81
7.3.6 Summary of the Total Improvement	81

8.2.1 Execution Time of Static Method vs Instance Method B.1	85
8.3.1 CPU and Memory Usage of Static Method vs Instance Method B.1	86

List of Code Snippets

6.1	Sample Code of Abstract Factory	56
6.2	Sample Code of Mobile Abstract Factory	57
6.3	Sample Code of Command	59
6.4	Sample Code of Mobile Command	60
6.5	Sample Code of Visitor	62
6.6	Sample Code of Mobile Visitor	63
6.7	Sample Code of Providing Global Access Point for the Context of a Activity using Singleton	65
B.1	Code of our Static Method versus Instance Method Test	119

Acknowledgement

The author would like to thank supervisor Eric Bartley Jul for his generous guidance throughout the thesis, and his invaluable advice and suggestions. Furthermore, the author would like to thank his family, fellow students, the University of Oslo, and the Oslo Metropolitan University who have helped him throughout his academic years.

Chapter 1

Introduction

1.1 This Project

This master thesis presents "Design Patterns for Mobile Devices" written by a master student in coordination with supervisor Eric Bartley Jul at the Department of Informatics, University of Oslo. The thesis is a comparative study of design patterns for mobile devices, combined with the actual implementation of multiple design patterns.

Design patterns are a well-known concept in software engineering that has been applied to object-oriented programming. It is a general repeatable solution to a commonly occurring problem in software design. The pattern is a description or template for how to solve a problem that can be used in many different situations. The authors referred to as the Gang of Four (GoF) released a book (1994) [15], describing software design patterns and 23 classical design patterns.

1.2 Motivation

The design patterns proposed by the GoF have had a significant impact on desktop applications and software development. The design patterns have provided well-tested solutions to recurring problems in software design and made the development process more efficient and less time-consuming. However, these design patterns were initially designed for desktop computers as the advent of mobile computing was not yet arrived. The development of mobile applications (hereafter named mobile apps) and their environment poses new challenges and limitations, making the design of mobile apps different from regular computers. The designer has to take into account different considerations and possibly new requirements imposed by mobile devices and mobile app development. These problems can be overcome with improved or new solutions, namely mobile design patterns.

1.3 Problem Statement

Designing applications for mobile devices differ from desktop computers as we have to take into account different considerations such as battery life, power consumption, variations in screen size, multiple types of user interfaces, different operating systems, and types of applications. As a result, some of the design patterns are not suitable for developing a mobile app. With a short time to market and mobile apps being more demanding in performance and sophisticated systems, there is a need for design patterns for mobile devices.

1.4 Goal

Our goal is to identify new requirements imposed by mobile app development and mobile devices on design patterns. Our end goal is to understand better what requirements and affect mobile app development and its mobile devices have on the design patterns, so we can more easily develop mobile design patterns that further enhance mobile apps in terms of performance, reusability, and maintainability.

1.5 Approach

This paper is an experimental study where we conduct a comparative study of design patterns for mobile devices to discover possible new requirements for mobile design patterns. A comparative study is essentially a method of comparing two similar objects to identify similarities and differences between them, and then trying to conclude which object of the two is superior. In this case, we investigate which design patterns are more suitable for mobile devices and try to conclude through testing and evaluation, if they are worth using to develop a better mobile app. Our approach of the comparative study will start with picking different design patterns, proposed by the GoF, for which we will implement in the development of a mobile game. Furthermore, we use various techniques and strategies for memory and energy consumption, and other examples of mobile design patterns from research articles to develop a mobile design pattern of the design patterns we have implemented. We will end up with two game projects where one is implemented with the regular design patterns, and the other has design patterns that are more suited for mobile devices. The projects and its design patterns will be compared and tested to relevant measurements against different criteria. We conclude whether the mobile design patterns are suitable for mobile devices or not through the evaluation of the projects and their design patterns. In the end, we analyze and identify possible requirements imposed by the mobile app development to the design patterns.

1.6 Work Done

The work done in this master thesis is summarized in chronological order in the following list.

- **Master Essay**

A master essay of around 15 pages related to the master thesis was written. The essay acted as a starting point for the thesis and served as guidance, especially for the introduction and background of the first and second chapters. The essay discussed one or two issues in at least two different scientific articles and a self-assessment. It aimed to give a background about design patterns for mobile devices and to understand the possible benefits of design patterns and why we need mobile design patterns in mobile app development.

- **Research**

During the writing of the thesis, most of the time was invested in reading research articles and books relevant to design patterns and mobile devices, software quality assessment of our app and design patterns, documentation about Android and its architecture, tools for profiling our app, and more.

- **The Execution of the Comparative Study**

We implemented the following five design patterns: Abstract Factory, Command, Visitor, Strategy, and Singleton in our development of a mobile maze game in Android using its official integrated development environment (IDE) Android Studio. The maze game was a continuation from the example used in the GoF book [15]. Two game projects were developed where one was implemented with the regular design patterns and the other with mobile design patterns, created by using various techniques and strategies for memory and energy optimization [17] [7]. In our measurement process of the projects and its design patterns, a hierarchical model was used for assessing some design quality attributes [3], Android's CPU and Memory Profiler was used for assessing the CPU and memory usage [1], and the execution time of the design patterns was measured. A plugin called MetricsReloaded from the Android Studio Marketplace was installed and used for calculating relevant metrics to the hierarchical model we used. The emulator and device Nexus 6 with API version 23 was used in our measurement process. After the measurement process, we analyzed and discussed the results of the design patterns. By the end, we identified the design patterns that were more suitable for mobile devices and requirements imposed on the design patterns from the process of developing a mobile maze game in Android.

1.7 Evaluation Criteria

In our comparative study, our regular and mobile design patterns Abstract Factory, Command, Visitor, Strategy and Singleton, were evaluated mainly towards two criteria or characteristics from the quality model in ISO/IEC 25010[11]: Performance Efficiency and Maintainability. In our evaluation, we defined Performance Efficiency as the extent to which our app performs to the required performance and amount of resources needed. Maintainability was defined as the degree to which the design patterns can be reused for other mobile apps, and the ease of use such that one change to a component should not affect another.

High performance is a crucial feature for our game app, and it is, therefore, important for our design patterns to manage the resources and the components of Android appropriately. Consequences of low performance can cause slow response time, freezing, crashing, and stutters in animations. The GoF Gang mentions in [15] that design patterns promote easier maintainability. Maintainability can be beneficial as it can e.g., accelerate the development process, avoid code duplication, and remove bugs and faults. As we change the structure of the design patterns when developing the mobile design patterns, it is critical to assess whether the maintainability of the mobile design pattern has maintained, improved, or worsen. Ideally, when evaluating the two criteria, we want to prove that the mobile design patterns further enhances the performance efficiency and maintainability of the mobile app.

The apps were tested with the emulator Nexus 6 API 23, which reflects the limited resources and processing power of a mobile device. Our verification criteria were first assessed using the Quality Metric Object-Oriented Model(QMOOD) [3] to connect information from the low-level source code metrics to the high-level design quality attributes efficiency, understandability, and reusability. We objectively measured the performance efficiency by using the Android Profiler tools: CPU Profiler and Memory Profiler, provided by Android. The tools were used to measure the real-time CPU usage of the design patterns, tracking its memory allocations and memory usage. Finally, we

also measured the execution time of the design patterns. A good score in performance efficiency can indicate that the app or design pattern is more resourceful, effective, and will experience lesser game experience issues such as lag, slow loading time, freezing, and crashes. A good score in maintainability can indicate that the design pattern is more reusable, easier to comprehend, and less complicated.

The requirements that we have identified to be imposed on the design patterns will be evaluated subjectively from our experiment, the development process, and our analysis of the differences between implementing design patterns in a non-mobile and mobile environment. This was decided due to the thesis being an empirical study and the lack of research we found for developing mobile design patterns in Android and other mobile platforms.

1.8 Results

Overall, the mobile design patterns had a better performance than the regular design patterns, but the patterns generally used more memory than the regular design patterns. Notably, the mobile Abstract Factory and mobile Command improved the execution time 36% and 29% compared to its regular design pattern, while the mobile Visitor and Strategy pattern improved only 3% and 2%. However, the CPU usage of the mobile Visitor and Strategy pattern decreased with 10% and 4%, compared to the mobile Abstract Factory and Command of 0% and -8%. A common theme in the QMOOD showed that the reusability of the mobile design patterns decreased while understandability increased. The theme was due to the techniques and strategies we used for optimizing memory and energy consumption.

The successful optimization techniques and strategies used for developing the mobile design patterns were to convert, if convenient, instance methods to static and combining multiple images to one. During our development, we noticed that a static method allocates less memory than an instance method and is also more CPU efficient. This is mainly due to the static method only needing one CPU instruction to be called, while calling an instance method requires one first to allocate an object to the heap. However, a drawback of converting the method of a class to static is that it can remove its inheritance capability, resulting in poorer reusability. Combining several images to one became a successful technique that led to faster execution time and better memory usage. It was used to combine several images of a game object to one, and instead of having to draw individual images, we extracted and drew the desired part from a single image.

We identified two requirements for the design patterns imposed by the architecture of Android and their mobile devices. The first requirement was for the design patterns to consider if it was necessary to work with the Activity component and incorporate its context. The second requirement was for the design patterns to manage the mobile device's limited memory in a more optimized way, leading to a more efficient and faster game app. Requirement one was first imposed on the Abstract Factory as the products or objects it created, needed the context of the activity when created. This was solved by passing down the context from the activity and eventually to the Abstract Factory methods. If requirement two was not properly handled, e.g., allocating unnecessary objects, it could quickly cause the app to freeze or crash. The requirement can be solved by applying the correct memory saving or/and power optimization techniques.

1.9 Conclusion

Developing a mobile game in Android imposed the following requirements to the design patterns. The first requirement was for the design patterns to consider if it was necessary to incorporate the components from the mobile platform's architecture, in our case, Android's architecture. From our experiment, the design patterns had to consider if it was necessary to manage the context of the activities and the activities themselves. If not managed properly, they could mess up the design patterns, e.g., the Abstract Factory could not create objects without the activity's context. The second requirement was for the design patterns to be more optimized and efficient, managing the memory appropriately, as they were not originally designed for mobile devices and its limited memory and computing capabilities.

Overall, using mobile design patterns improved the mobile app in terms of performance efficiency but had mixed results in maintainability. The use of power-optimized and memory saving techniques such as converting the instance methods to static would most likely increase the execution time of the design patterns, but it could also remove its inheritance mechanism. This would, according to the QMOOD results, decrease the reusability of the design pattern and instead increase the understandability.

Noticeably, the combination of replacing instance methods with static and combining multiple images to one improved the execution time of Abstract Factory with execution time 36%. An important lesson we learned and believe is important to understand when developing mobile design patterns is memory allocation, to avoid unnecessary allocation of memory, especially to the heap, and managing memory in a more optimized way.

1.10 Contributions

Proposing a design pattern for mobile devices has been achieved before. The author, Fang-Fang Chua [8], presented an extended version of the model-view-controller pattern that can be used to separate the presentation and the app code. In [6], the author introduced an Energy Concerned Factory Method designed for a power limited environment. We have contributed to clarifying more about what requirements mobile devices and its app development impose on design patterns through our comparative study and analysis. We attempted to clarify the mobile and non-mobile environment's differences and what effect the mobile environment has on the design patterns so that we can better understand how to develop and use design patterns for mobile devices. From our experiment, we have translated some regular design patterns to mobile design patterns for the Android platform and identified two requirements imposed on mobile design patterns by Android and its mobile devices.

1.11 Limitations

This master's thesis was limited to 60 points written by a single student with the help of a supervisor over three semesters. With limited time and human resources, the comparative study and the apps developed were of a small size.

There are several weaknesses and faults in our measurement process and evaluation criteria. One fundamental mistake was that we started the measurement process after we had finished the development of the regular and mobile design patterns, making it difficult to pinpoint which changes had the least and most impact in our measurement process.

Due to adequate time, we used and analyzed only one component of the Android architecture was in our study. Other components from the Android system was not included in our app or code. Thus one can argue, our study was not complete as these components could also impose new requirements on mobile design patterns.

One of our main tasks was to develop design patterns for mobile devices. However, we tested the design patterns on only one mobile device or emulator. Testing the design patterns on several different emulators would have better reflected the different characteristics of a mobile device, thus lead to a better validation of the results.

The end goal of our evaluation criteria was to clarify what possible requirements are imposed on mobile design patterns. However, this thesis is limited to the development of a mobile game whose primary concern was memory usage, performance, and some aspects of maintainability. Hence, there is a need for research to develop design patterns in other types of mobile apps, such as creating an app with varying user interfaces, an app using sensors, GPS, or reliance on stable internet connection.

1.12 Outline

This paper has five main parts and nine chapters. The parts are essentially the chapters between the introduction and conclusion, but part 4 and 5 combine two chapters as they are highly relevant to each other. The following list describes the five parts.

- **Part 1: Background**

We start by going through the background of this thesis, where we describe the design patterns introduced by the GoF and the ones we use, the main challenges of mobile devices, and mobile app development. In the end, we introduce some examples of mobile design patterns from other research articles.

- **Part 2: Evaluation Criteria**

Part two is the evaluation criteria of our comparative study, where we describe our evaluation criteria and measurement process of the design patterns. The main topics are the ISO/IEC 25010 standard, the QMOOD model, and the Android Profiler tools, which we will use in our measurement process.

- **Part 3: Analysis**

Part three analyzes and discusses the challenges with mobile design patterns, mobile app development, and how it can affect the design patterns. Focusing on Android and its architecture, we discuss how it is different from a non-mobile environment and what issues a developer might face when developing an app. In the end, we describe the methods we use to develop our mobile design patterns.

- **Part 4: Project and Implementation**

Part four describes the project case of our game app, its overall architecture, and the task of our design patterns. Next, we present the implementation of our regular and mobile design patterns, and its differences with the help of code snippets and UML (Unified Modeling Language) class diagrams [16].

- **Part 5: Results and Discussion**

Part five showcases first the results of the measurement process. Afterward, we discuss the results and analyze the methods which we have implemented in our mobile design patterns. In the end, we discuss the requirements we have identified to be imposed on the design patterns.

Chapter 2

Background

2.1 Introduction

We start by introducing design patterns in software engineering and the background of the thesis. As this is about studying mobile design patterns, we will not go in-depth on the technical part of a design pattern, for instance, how design patterns use the principles from object-oriented programming to solve design problems. This is better explained in the GoF book[15], which we recommend reading if one wants to know more about design patterns.

2.2 What is a Design Pattern?

According to Wikipedia, a design pattern in software engineering is defined as "a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source code or machine code. It is a description or template for how to solve a problem that can be used in many different situations." [28]. A design pattern is about how the classes and interacting objects are used to solve a design problem in a specific context. As mentioned in the definition, a design pattern does not provide us the implementation of the solution for a problem. We can rather think of it as a solid and a formal blueprint that shows us how to solve the problem. The result of the blueprint will be unique from developer to developer.

2.2.1 Brief History

The concept of design pattern was created and introduced by the influential architect Christopher Alexander. Mr. Alexander can be seen as the early pioneer of design patterns as formal ideas. As he said in his book *A Pattern Language*, "... Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [2].

Now design patterns have also become a useful and powerful concept in software engineering that has been applied to object-oriented programming. The book *Design Patterns: Elements of Reusable Object-oriented Software* [15], known as the GoF book from 1994, had an influential impact on the field of software engineering, describing 23 object-oriented design patterns. Still, to this day, the book is regarded as an important source for object-oriented design theory and practice.

2.2.2 The Essential Elements of a Design Pattern

A design pattern describes and evaluates a solution to a common problem by its four essential elements [15]:

- **The pattern name** is the name of the design pattern. Having a good and short name is important because it describes the design problem, its solutions, and consequences in a word or two. The name of the pattern is also added to the design vocabulary in the pattern language. Having a common vocabulary for design problems enables us to design at a higher level of abstraction. Also, it makes it easier to communicate with others.
- **The problem** describes what problem the pattern solves and what context it is in. The problem that the design pattern solves usually consists of general intent and motivation, describing in the end when to apply the pattern.
- **The solution** describes the elements that make up the pattern, such as classes and objects. It also includes their relationships, responsibilities, and collaborations. The solution does not provide us with a concrete design or implementation, but rather abstract description of how to solve the design problem. The benefit of this is that we can apply the solution in many different situations and implement the solution in our way.
- **The consequences** is about the results and trade-offs we get for applying the pattern. More than one pattern can likely solve the problem. Thus the consequences often become the determining factor. Therefore, understanding the consequences of choosing a design decision is crucial for evaluating design alternatives and acknowledging the drawbacks and benefits of applying the pattern.

2.3 Documentation

According to the GoF book, design patterns are described using a consistent format for making learning, comparison, and use easier. The pattern is divided into different sections of the template, and down below are some sections that are usually present in the pattern description.

- **Pattern Name and Classification**

A short and meaningful name of a pattern which will be added to our design vocabulary.
The pattern's classification is a way to organize and group the patterns.

- **Intent**

A short description of the purpose of the pattern by briefly describing the solution of the pattern and what design problem it addresses.

- **Motivation**

A scenario that shows how the classes and object structures in the pattern solve the problem to describe the problem in detail further. The scenario is a guide for the users to help understand the more abstract description of the pattern that follows.

- **Structure**

A graphical representation of the classes in the pattern using a modified UML (Unified Modeling Language) notation [16]. Also, we can use interaction diagrams to illustrate sequences of requests and collaborations between objects.

- **Consequences**

Discussing the results and trade-offs one gets by using the pattern and how the pattern support its objectives. One can also discuss what aspect of system structure does the pattern let us vary independently.

- **Implementation**

What pitfalls, hints, or technique is one should be aware of when implementing the pattern. Are there also any issues with language dependency?

- **Sample Code**

Code examples that illustrate how one might implement the pattern in Java, C++, and more.

- **Related Patterns**

Discussing other design patterns that are closely related to this pattern. Furthermore, describing the important differences between them and with which other patterns can or should this one be used.

2.4 Why should one use Design Patterns?

Design patterns are a set of tried and tested solutions for common problems in software design and has arguably become a standard of essential knowledge for software developers nowadays. One advantage of using design patterns is that we do not have to reinvent the wheel for every time one comes across a similar problem. Instead, one can save time by implementing a proven and well-documented solution. Furthermore, one will reduce the risk of implementing a solution that is untested and new or, in the worst-case scenario, implementing a design that is later discovered not to be viable. Design patterns also aid communication for teammates by defining a common language. Instead of describing our suggestion to a design problem in detail, we can instead say the name of the design pattern that solves the problem, and everyone will understand the idea behind our proposal.

2.5 Classification

Design patterns differ in their granularity and level of abstraction [15]. Some patterns only concern the process of creating objects, while other patterns might deal with the composition of classes or objects. This is where classification comes to play. The design patterns are classified so that we can refer to families of related patterns. Since there are many patterns, having this classification helps us to learn the patterns faster and deepen our understanding of what the patterns do, how they compare, and when to apply them. There are three main groups to which design patterns are categorized to:

- **Creational patterns** cover the process of object creation and abstracts the process from the code that relies on it.

- **Structural patterns** is about composing objects and classes to form a larger and flexible structure.
- **Behavioural patterns** focuses on how classes or objects communicate effectively and distribute responsibility between them.

Table 2.5.1 shows an overview of the classification of the GoF design patterns with their category. The red highlighted design patterns in the table are the patterns we implemented in our mobile game. The next section describes them in more detail.

	Creational	Structural	Behavioral
Class	Factory Method	Adapter	Interpreter Template Method
Object	Abstract Factory Builder Singleton Prototype	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Table 2.5.1: Overview of Design Pattern Classification [15]

2.6 Description of our Selected Design Patterns

The patterns we selected for our study are Abstract Factory, Command, Visitor, Strategy, and Singleton. These five patterns were simply selected because they were commonly used, and the author did not have adequate time to use more. The patterns are described with their intent, motivation, and structure, and we also describe the Factory Method as it is a part of our Abstract Factory implementation. For more details, please read the GoF book [15].

2.6.1 Abstract Factory

Intent

Abstract Factory provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Motivation

Imagine developing a shopping website where one sells different products that belong to different brands. One has several shoes, t-shirts, and jackets from Nike, Puma, and Adidas, and the website should have a feature to filter the products so that one only shows products from a specific brand. Here, we want to avoid the cumbersome process of manually changing the code, each time we display the products from a particular brand. Abstract Factory solves this problem by defining sub-factories,

creating objects related to their family. Thus, one can easily switch between families of objects by simply changing the factory that makes them.

Structure

From figure 2.6.1, we see AbstractFactory defining interfaces for creating different products. The sub-factories of AbstractFactory implements the interfaces, returning a concrete product related to their family. For instance, the method CreateProductA() in ConcreteFactory1 returns the ConcreteProductA1 while the same method in ConcreteFactory2 returns the ConcreteProductA2. Furthermore, the client that uses a concrete factory does not need to specify or know the exact class that will be created.

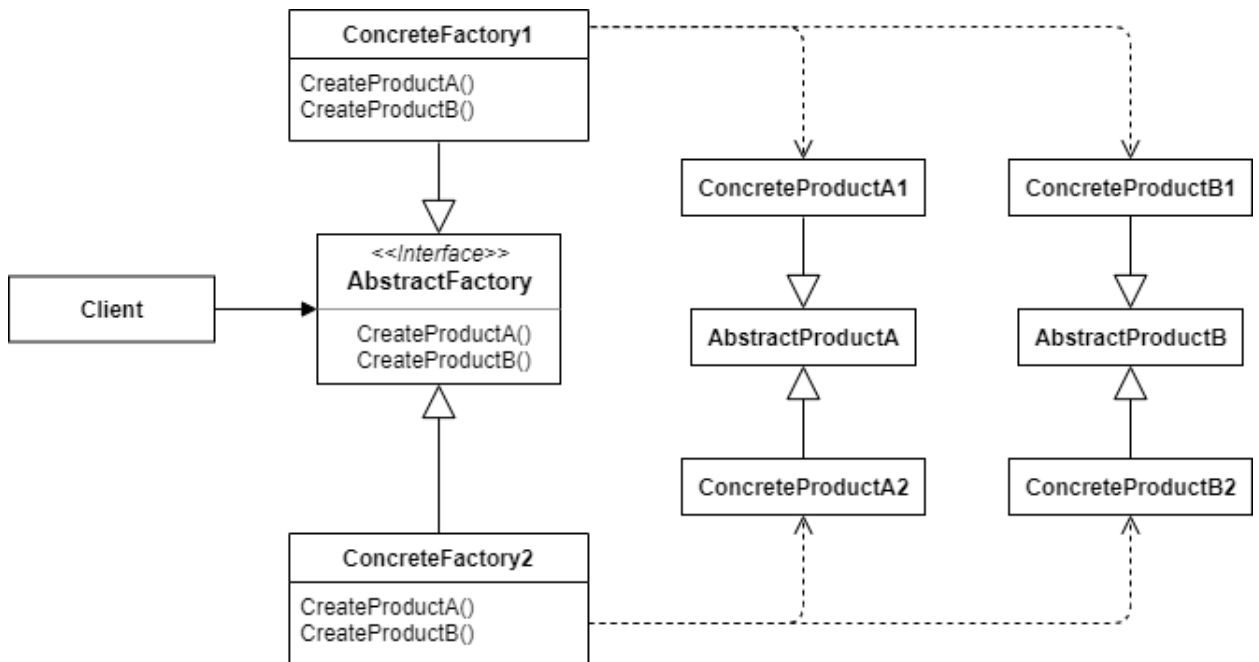


Figure 2.6.1: General UML representation of Abstract Factory

2.6.2 Singleton

Intent

Singleton ensures a class only has one instance and provides a global access point to it.

Motivation

Sometimes one wants a class to have no more than one instance. There should be only one accounting system committed to serving a company. A country may have only one official government with a global access point that refers to those in charge. A game app should be running on only one game engine.

Structure

The class itself is responsible for keeping track of its instance and making sure that no other instance can be created. It accomplishes this by having a private constructor and a static method getInstance(), which intercepts requests to create new instances

of the class. The method also provides a global access point to the instance by returning the same instance of the class.

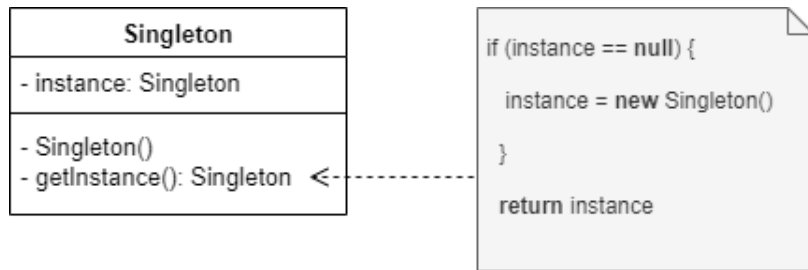


Figure 2.6.2: General UML representation of Singleton

2.6.3 Command

Intent

Command encapsulates a request as an object, thereby letting us parameterize clients with different requests, queries or log requests, and support undoable operations.

Motivation

The command pattern acts as a middle layer between the sender and the receiver. The sender can then issue requests without knowing about the receiver of the request or the operation that is being performed. The command pattern handles this part. For instance, imagine we are working on a toolbar, developing buttons and menus that perform different requests based on user input. A flawed solution would be to implement each request in each button or menu. When further developing other menus, the code can then be duplicated or have menus dependent on the buttons. Besides, the buttons and other menus would be highly dependent on the business logic of the receivers. Any change of its business logic would lead us to have to change the code of every object that was dependent on it. The Command patterns let the toolbar objects (sender) make requests to unspecified receiver objects by converting the request itself into an object. This object can then be passed and stored around, avoiding duplication of code and making the toolbar objects unaware of the receiver of the request and the operation that will be carried out.

Structure

The crucial part of this pattern is an abstract Command class, which defines the abstract method `Execute()` for executing an operation. As we can see from figure 2.6.3, the `ConcreteCommand` class defines a binding between the receiver and the action. Its `Execute()` method will then invoke the corresponding operation on the receiver. The receiver should know how to execute the action associated with the request. With the important parts in place, the client can create a `ConcreteCommand` object and sets its receiver, and the invoker can call the command to carry out a request.

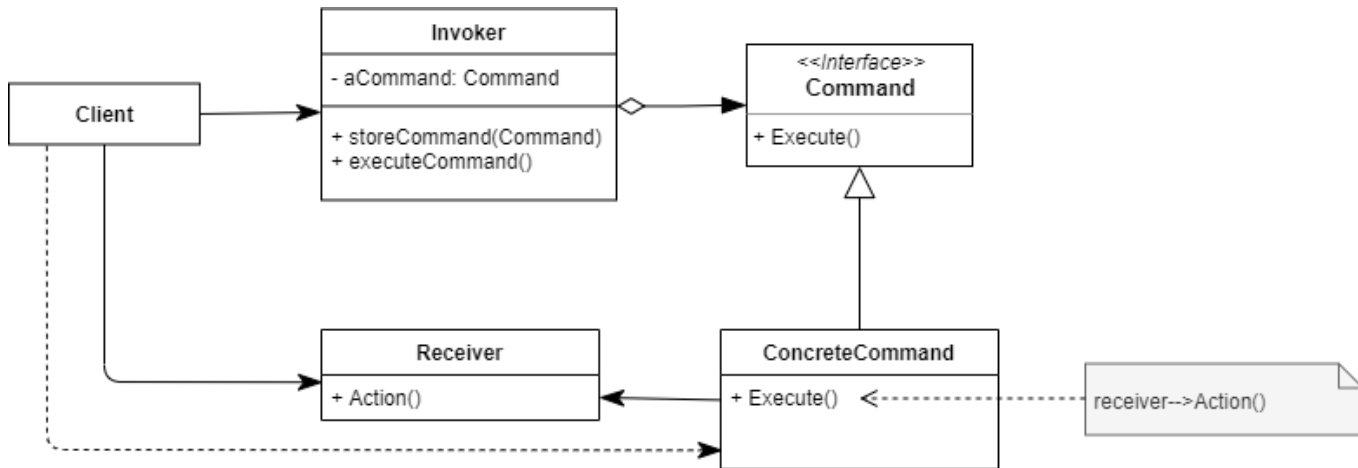


Figure 2.6.3: General UML representation of Command

2.6.4 Visitor

Intent

‘Represent an operation to be performed on the elements of an object structure. Visitor lets one define a new operation without changing the classes of the elements on which it operates.’[15]

Motivation

The Visitor pattern lets one separate code or operations that are distinct and unrelated to the structure of objects, leading to a system that is easier to maintain, change, and understand. Say we have a shopping cart full of products (elements). The products have different ways of calculating their price, some might need to be weighted, and others have prices. To avoid adding additional code in the element itself for calculating their total price. The cashier will instead act as a visitor, taking the different set of products and weighting them with prices to provide us with the total cost.

Structure

From figure 2.6.4, we see that the interface Visitor declares a visit operation for each concrete element of Element. Each visit operation takes in a specific element as a parameter, which lets the visitor identify the concrete class of the element it visits, and then give direct access to the element. The ConcreteVisitor implements each visitor operation declared by the Visitor and stores its local state. The Element simply defines an Accept operation that accepts a visitor to let it perform some actions on itself. The concrete elements implement the accept operation.

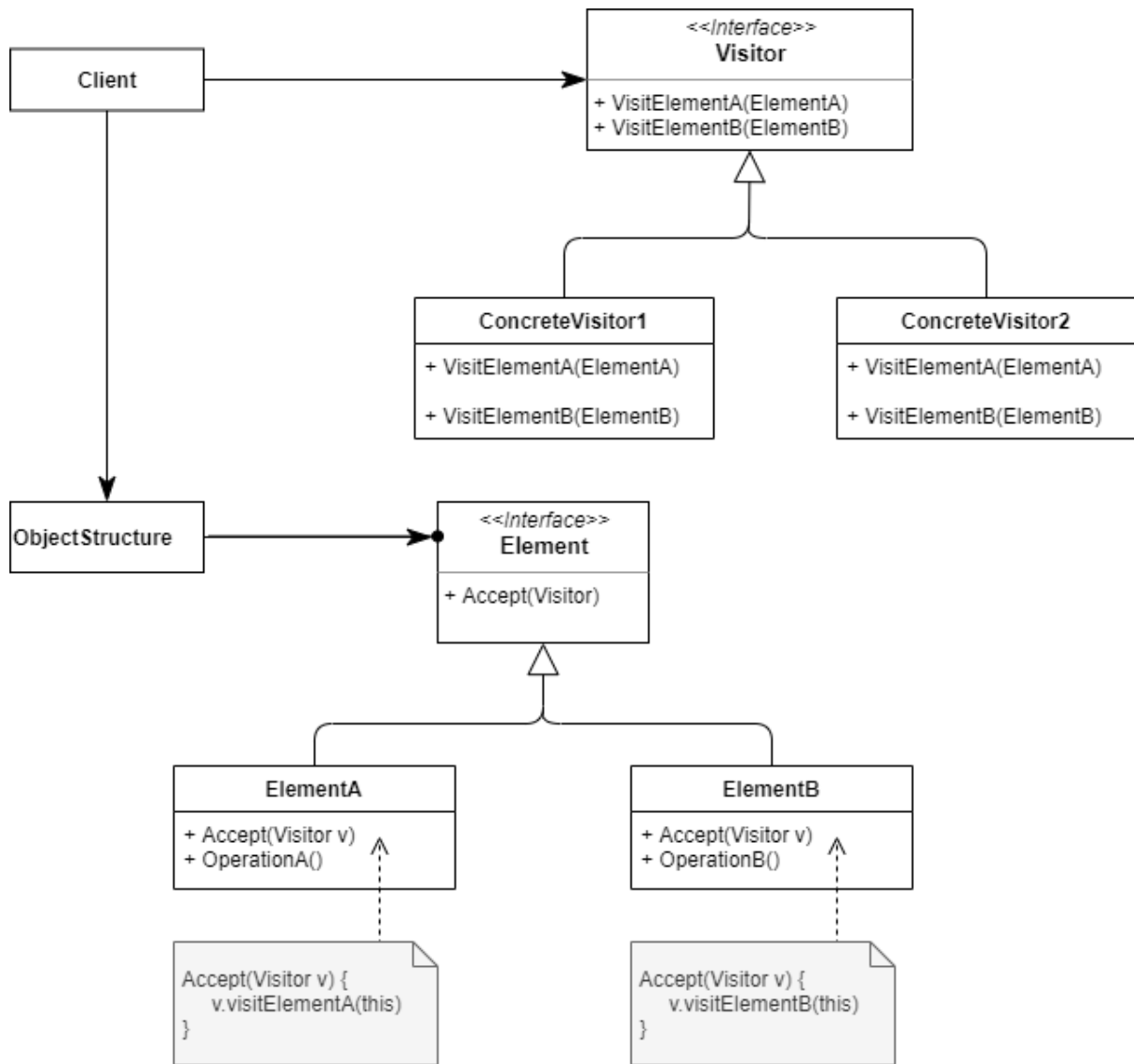


Figure 2.6.4: General UML representation of Visitor

2.6.5 Strategy

Intent

‘Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.’[15]

Motivation

The Strategy is useful when we want to choose or change between interchangeable algorithms in runtime to carry out a specific behaviour. For instance, when we want to compress files using different approaches such as ZIP and RAR. We can define classes that encapsulate different compression algorithms, which will be our concrete strategies.

Structure

Following figure 2.6.5, shows the general UML representation of Strategy. The context is composed by a Strategy. The Strategy defines a interface common to all concrete strategies. This interface is used by the context to call concrete strategies. The concrete strategies implements the algorithm interface.

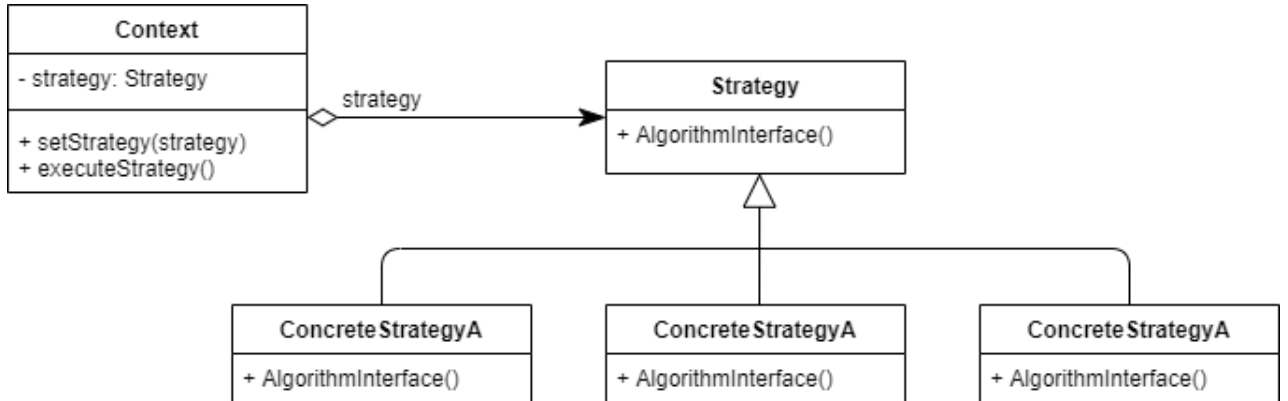


Figure 2.6.5: General UML representation of Strategy

2.6.6 Factory Method

Intent

The factory method defers the instantiation of a class to its subclasses.

Motivation

The intent is accomplished by defining an interface for creating an object in a superclass and then letting the subclasses alter which type of object to be created. Usually, one wants to use this pattern when we create one or several classes that share a common superclass. Some indicators to use this pattern are when we do not know ahead of time what class object we need and when we do not want the user to know every subclass.

There are several advantages to implementing this pattern. One avoids tight coupling between the ConcreteCreator and the ConcreteProducts 2.6.6, and the user does not need to specify or know the exact class of object that will be created.

Structure

Figure 2.6.6 shows the general class diagram for the factory method. The ConcreteCreator class is the factory class which has a FactoryMethod that we call for generating a ConcreteProduct. Usually, we decide what type of ConcreteProduct we want by sending a parameter to the FactoryMethod in ConcreteCreator. Furthermore, the Product is the parent class to ConcreteProduct, and normally, there is a group of different concrete products that inherit from a Product class. For instance, the product class can be an abstract Vehicle class. Then we have several ConcreteProducts classes such as Ship, Car, and a Boat that inherits from Vehicle.

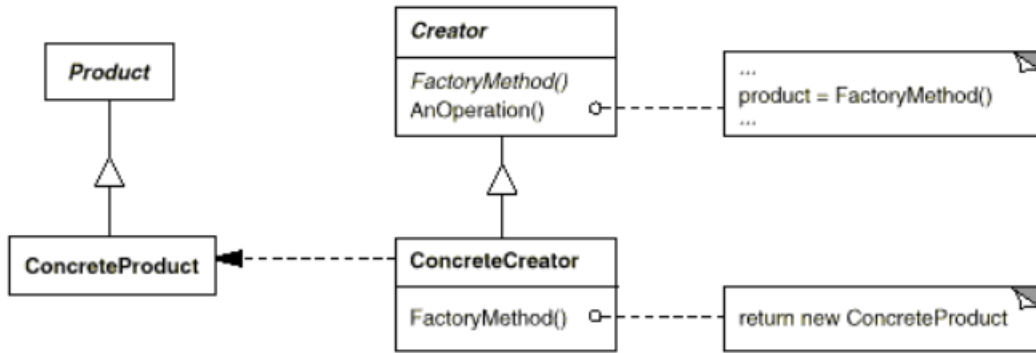


Figure 2.6.6: General UML Representation of Factory Method.

2.7 Summary

Design patterns are a powerful and useful concept in software engineering, a set of tried and tested solutions for common problems in a specific context. The design patterns are classified and grouped into either creational, structural, or behavioral for making it easier to help us learn the patterns and more profound our understanding about them. The catalog of design patterns was introduced by the Gang of Four and consisted of 23 design patterns that are widely known and used to this day. The different patterns are described using a consistent format consisting of sections such as the pattern name, the intent of the pattern, and more. Finally, we have illustrated an example of the Factory Method, which is, in our case, a part of the Abstract Factory. This pattern is later referenced in section "Mobile Devices" when we look at a mobile version of this pattern called Energy-conscious Factory Method [6].

2.8 Mobile Devices

2.8.1 Introduction

Over the last decade, mobile devices' evolution in the market and demand for complicated software running on mobile devices has proliferated. As an example, mobile phones went from having only simple features at the beginning of the 2000s, such as messaging, camera pictures, phone calls, to becoming central in our modern life offering payment services, apps, high video streaming capabilities, and more. With mobile devices available on the market, differing in sizes, power capabilities, battery life, different platforms, and a short time to market have made developing and designing mobile apps a challenging task. In this section, we will look into the challenges and issues with mobile app development, and then briefly showcase some examples of existing mobile design patterns and how they can overcome some of these challenges. But firstly, we describe what a mobile device is.

What is a Mobile Device?

Mobile devices, also known as handheld computers, are defined as a "computing device that is small enough to hold and operate in hand" [27]. The significant difference between mobile devices and computers is that mobile devices are made for portability, and is, therefore, both lightweight and compact [26]. Mobile devices like smartphones, tablets, or e-readers vary in size and computing capability; some devices are small enough to fit in our pocket, and others may be larger with more processing power and memory.

In this paper, we define "mobile device" as a smartphone where it is possible to develop mobile apps on, as we will develop mobile apps using Android.

2.8.2 Main Challenges with Mobile Devices

Different mobile devices on the market vary in screen size, memory usage, processing power, battery life, and operating systems. As we can see, these are boundaries one has to take into account when developing a mobile app, making it challenging to develop a runnable app for the different mobile devices and keep up with the fast-evolving and short to time market.

Below are some of the main issues and challenges related to mobile devices, according to [12] [8].

Battery Life

Unlike desktops, which are always attached to AC power, mobile devices have the desire to use and operate on batteries (DC power). Due to this desirability combined with the size constraints of mobile devices, a new constraint is created; limited power supply.

Computing Power

Mobile devices have less computational processing power than regular computers. Normally an app on a computer can not run on a mobile device as it requires heavy computation. Nevertheless, the app can redesign to manage the resources more appropriate for mobile devices with various techniques. The consequences of a low efficient mobile app can cause problems such as slow load time, crashing or freezing when accessing it.

Wireless Network and Communication

Mobile devices require wireless network access, that allows network communication even while a user is mobile. As a result, wireless networks and communication introduce challenges such as lower bandwidth, high bandwidth variability, heterogeneous network, and security risks [12].

Mobility

While a stationary computer can configure to prefer the nearest network address, mobile devices need to determine which server to use while still connected to the network. As a result, mobility introduces issues like address migration, location-dependent information, and migrating locality.

Portability

Being portable causes issues such as a smaller battery, user interface, storage capacity, and also increases the risk of data (risk of physical damage, unauthorized access, loss, and theft).

2.9 Development of Mobile Applications

2.9.1 Introduction

Mobile app development is a relatively new phenomenon that is increasing rapidly due to the ubiquity and the popularity of smartphones among end-users. As we will see, there are different approaches to developing mobile apps, and choosing the right approach for one's mobile app is important, as different apps and platforms have different requirements. Understanding the differences between the different approaches and what they offer will give us a better insight into developing mobile apps.

2.9.2 The Different Approaches

There are different approaches for developing a mobile app, namely native, web, and hybrid. Native apps are executed entirely on the operating system (OS) of the device. The advantage of native apps is that we get access to all the features (in many cases, unique features) and functionality made available by the OS vendor. A critical disadvantage of native apps is that the code is written platform-specific, making the development of native apps for different OSs an expensive task. Web apps have a device-based client executing on a remote server and thus support multi-platform. The main disadvantage of web apps is that they have limited access to the features and functionality made available by the OS vendor. Web apps are not capable of developing apps where key elements of the app are required to be developed natively; for example, Skype having access to user contacts is necessary to achieve the desired functionality. The hybrid approach is both native and web-based, giving us "native-wrapped" web apps. Essentially, with this approach, we get the best of both worlds: native features and multi-platform support.

2.9.3 Main Challenges with Mobile Application Development

Different mobile platforms are running their own OS, such as Apple's iOS, Google's Android, Blackberry OS, and Windows Phone. From [table 2.9.1, p. 22], we can see that they have different languages, tools, file formats, and more for their mobile app development. As [8] mentions, mobile devices having these different requirements, operative systems, and platforms dedicated to them, makes it a challenging task to design and develop a mobile app. For instance, some platforms are more suited for deploying the type of application: web, native, or hybrid. The Software Development Kit (SDK) provided by Android consists of a set of tools for creating native apps for Android devices. [22] presents six design considerations when designing a mobile app:

- Decide if one will build a native, web, or hybrid app
- Determine device types one will support considering screen size and resolution, CPU performance characteristics, memory, and storage space.
- Take into account limited bandwidth scenarios such as connection issues, choosing hardware and software protocols based on speed, power consumption, and granularity.
- Design a user interface (UI) while taking into account platform constraints: memory, battery life, adaptations to different screen sizes and orientations, security, and network bandwidth.

- Design an appropriate layered architecture for mobile devices by using the concept of layers to maximize the separation of concern and to improve reusability and maintainability.
- Consider device resource constraints like battery life, memory speed, and processor speed in every design decision.

[19] conducted a qualitative study, identifying the main challenges developers face in practice when developing apps for different mobile devices. The following are some of the main challenges the developers from the study faced with developing a mobile app with different mobile platforms such as Microsoft, Apple, and Windows.

- **Open/Closed Source Development Platforms** There was a dilemma in the study about whether the platform should be open or closed. Participants expressed difficulties with both types. One of the drawbacks they expressed of a closed platform like Apple and Windows was that they lacked control since they did not have an API to control. For example, it was difficult for them to determine if they were connected to Bluetooth. On the other hand, some participants had difficulties with the open platform Android where manufacturers did not always adhere to the standards by modifying the source code to their wishes and launching it. They gave the following example: "the standard Android uses commas to separate items in a list, but Samsung phones use a semicolon".
- **Frequent changes and steep learning curve** A common challenge between the many developers was learning the languages and API of the different platforms and keeping up with the frequent changes to the software development kits (SDK). Most mobile developers would want their app to support different platforms, but this can be a cost of quality as learning the different platforms is a steep learning curve, since each platform are completely different (languages, marketplaces, guidelines, tools).
- **Data-intensive apps** Dealing with a large volume of data for apps is problematic, especially when one is limited to mobile constraints. A respondent explained this furthermore that: "So much data cannot be stored on the device, and using a network connection to sync up with another data source in the backend is challenging". Concerning hybrid apps, a participant quoted, "Our apps have a lot of data and offline caching does not seem to work well".
- **Reusing code across different platforms** The majority of the participants concluded that reusing code or porting functionality across the platform was challenging or impossible. Also, when reusing code across a platform, the quality of the result was not adequate. This seems logical, as we have mentioned in Table 2.9.1 that the platforms have different requirements for development (programming language, packaging, and operative systems). A participant explained in further detail a simple example of how the platforms push messages, for why porting code is not possible: "... In Android, a push message wakes up parts of the app, and it requests for CPU time. In iOS, the server would pass the data to the Apple push server. The server then sends it to the device, and no CPU time to process the data is required."

Varying User Interfaces

In [4], the authors mention creating apps for stationary users on a PC or a similar device, using the typical user interfaces keyboard, mouse, and monitor have proved to be fairly

efficient. For mobile apps, these user interfaces are not applicable. There are varying and alternative interfaces for mobile apps such as voice user interfaces, small displays, touch-screen displays, pointing devices, and buttons. Having to account for multiple user interfaces and often used in combination can lead to heavy user interaction, tightly coupled apps, and an unstructured design.

Overview over Native Application Development

As previously mentioned, choosing the right OS and platform for one's mobile app is important as the different OS have different requirements for their mobile app development. Following table is a overview over the native app development and the different requirements of the operative systems.

Operative Systems	Apple iOS	Android	Blackberry OS	Windows Phone
Languages	Objective-C, C, C++	Java and some C,C++	Java	C#, VB.NET and more
Tools	Xcode	Android SDK	BB Java Eclipse Plug-in	Visual Studio Windows Phone Dev Tools
File format	.app	.apk	.cod	.xap
App marketplaces	App Store	Google Play	Blackberry App World	Windows Phone Marketplace

Table 2.9.1: Overview over Native app Development

2.10 Why do we need Mobile Design Patterns?

The GoF introduced 23 design patterns that have been influential in software development. The design patterns have proven to be well tested and documented solutions for common problems in a specific context. However, can we use the same solutions in the context of mobile app development? Most likely not, as the issue with applying the design patterns directly to mobile devices is that there were initially meant for regular computers. The challenges of mobile app development, like minimizing power consumption, had never been a concern [6]. However, the design patterns can be adapted and designed for such an environment and overcome the limitations and challenges with developing apps for mobile devices.

2.10.1 Examples of Mobile Design Patterns

From studying research articles, we found some examples of mobile design patterns that overcame different challenges with developing mobile apps. The following is a brief description of the examples we found.

High-efficiency Model-view-controller pattern for mobile apps

The author Fang-Fang Chua [8] proposed an extended version of the model-view-controller (MVC) pattern, which combines the Observer, Command, Composite, Mediator and Strategy design patterns. The MVC pattern is a well-known pattern consisting of three components: model, view, and controller, where the intent is to separate the business logic (model) and the presentation logic (view) from each other. In the research article, Dr. Chua claims that the "Extended MVC pattern is used to separate the presentation and the app code. This allowed us to develop an app

with loose coupling and separation of concern.". The extended MVC made it easier to reuse modules and remove and add features without affecting other modules.

The extended MVC and MVC patterns were implemented, tested, and evaluated for a Student Planner Android app on different mobile devices to verify if the proposed design pattern was more reusable and efficient in mobile app development. The test results showed that the extended MVC pattern greatly improved the efficiency of the mobile app. Furthermore, the pattern reduced code duplication and made the app compatible with various devices and properly handled elements such as edit text, buttons, and scrolling.

Energy conscious factory method design pattern for mobile devices

In this research article [6], the author talks about how the design patterns for object-oriented-programming, introduced by the GoF were targeted for regulars computers and not mobile devices. Thus, most of the design patterns are not appropriated for developing mobile apps, mainly power concerned systems due to limited battery life. The author intends to solve this constraint by introducing Energy Concerned Design Patterns, which are based on the existing design patterns from the GoF book. Starting with the popular Factory Method pattern, the author introduces an Energy Concerned Factory Method or EC Factory Method.

The difference between the EC Factory Method and the regular Factory method is that the EC Factory Method is designed for a power limited environment by making use of struct, classes and static methods in the most efficient combination. Instead of having the regular Factory Method which consists of a ConcreteProduct class, ConcreteCreator class, and a non-static FactoryMethod, it had a ConcreteProduct class, ConcreteCreator struct, and a static FactoryMethod. The results of the research showed that the EC Factory Method consumed less than approximately 11% CPU time than the regular Factory Method, and around 80% in the worst-case scenario, making the EC Factory Method more appropriate for developing mobile apps rather than implementing the regular Factory Method.

Design pattern based approach for development of game engines in mobile platforms

This paper [23] examines the design of a mobile game engine, that satisfies both functional and non-functional requirements, using design goals and design patterns.

The author focuses on four main design goals for developing a game engine that is suitable for mobile games. The design goals were mostly determined by hardware and software specifications of mobile platforms, thus the main design goals were: Usability, Efficiency, Portability, and Adaptability. Furthermore, the design patterns were used to help achieve these goals by using efficient conscious design patterns such as Flyweight, Frame limiting, and other tested and proven game development patterns. For instance, memory efficiency was obtained by using the design pattern Flyweight. Flyweight is a structural design pattern where the intent is to minimize memory usage by sharing as much common data between similar objects, so one can fit more objects into the available amount of RAM instead of holding all the data in each object. In this example, they held the static properties of a game object on a separate registry associated with each type of element. As a result, when a new instance was requested, it got provided with allocated dynamic properties and mapped static properties, reducing the size of the element by double.

The tests showed that including mobile-centric design patterns, the performance of the game engine increased drastically. The memory consumption was reduced up to 56% and power consumption decreased up to 6%. Both ratios increased with increasing the number of game elements up to a certain extent. In conclusion, the paper concluded that "Using design patterns allowed us to use industrial-strength, tested and proven methods in the mobile development context."

2.10.2 Summary

The fast evolvement of mobile devices and the popularity of mobile apps in the market demand complicated software to be run on mobile devices without problems such as crashing and freezing. Mobile devices are known as handheld computers and they have different characteristics such as size, resolution, battery life, and processing power. These are some boundaries one has to take into account when developing a mobile app which can make it difficult to design and develop mobile apps for different mobile devices. Design patterns, a set of tested and proven solutions for common problems were originally designed for desktop computers, thus applying these patterns in mobile apps might not fulfill one's requirements such as minimizing power consumption. However, mobile design patterns that have been adapted or designed for such an environment, can be used to overcome the challenges and limitations with mobile app development.

2.11 Summary

This chapter has introduced design patterns, documentation, and the catalog of design patterns. The catalog of design patterns consists of proven solutions to common design problems in a specific context. We believe that these proven solutions can be implemented and adapted to mobile app development, overcoming the limitations and challenges with developing mobile apps for mobile devices.

We have identified and analyzed some design patterns for mobile app development and how they have overcome some of the limitations and challenges with developing apps for mobile devices. We have looked at an energy-conscious factory method that is suited for power-conscious apps and a high-efficiency MVC design pattern, which improves the quality of mobile apps in terms of usability, efficiency, and re-usability.

This chapter aims to give a background about design patterns and to understand the possible benefits of design patterns and why we need mobile design patterns in mobile app development. Our proposed work is through developing, testing, and evaluating different mobile design patterns, find the requirements imposed by the mobile app development and its mobile devices onto design patterns. By the end, we hope to understand better how to develop mobile design patterns.

Chapter 3

Evaluation Criteria

3.1 Introduction

In this chapter, we explain the history of quality models, its positives and negatives, and then describe our selected verification criteria from the quality model in international standard ISO/IEC 25010, for which our mobile apps and its implemented design patterns have been evaluated towards. Relevant to our defined criteria, we performed a source code metric analysis with the Quality Metric Object-Oriented Model (QMOOD) introduced by Bansiya et al. [3]. In the end, we describe the tools and plugins provided by Android, which we use in our measurement process.

3.2 Quality Model

A quality model is used for evaluating software quality and can be used during the development stage to ensure and follow that the quality of the software is maintained. One of the first quality models was introduced in 1977 by McCall [21], and later that year, another popular model was introduced by Boehm [5]. These models became important predecessors for the widely known quality model described in the international standard ISO/IEC 9126 [9]. The quality model of the standard describes six quality characteristics with their sub-characteristics, for classifying software quality [10]. The main characteristics are functionality, reliability, usability, efficiency, maintainability, and portability. Its successor, ISO/IEC 25010 described in addition to two more design characteristics, namely compatibility and security [11]. In this paper, we focus on the quality model of ISO/IEC 25010, because it replaced its predecessor ISO/IEC 9126, and is therefore arguably the most relevant quality model we can use.

ISO/IEC 25010 Quality Model

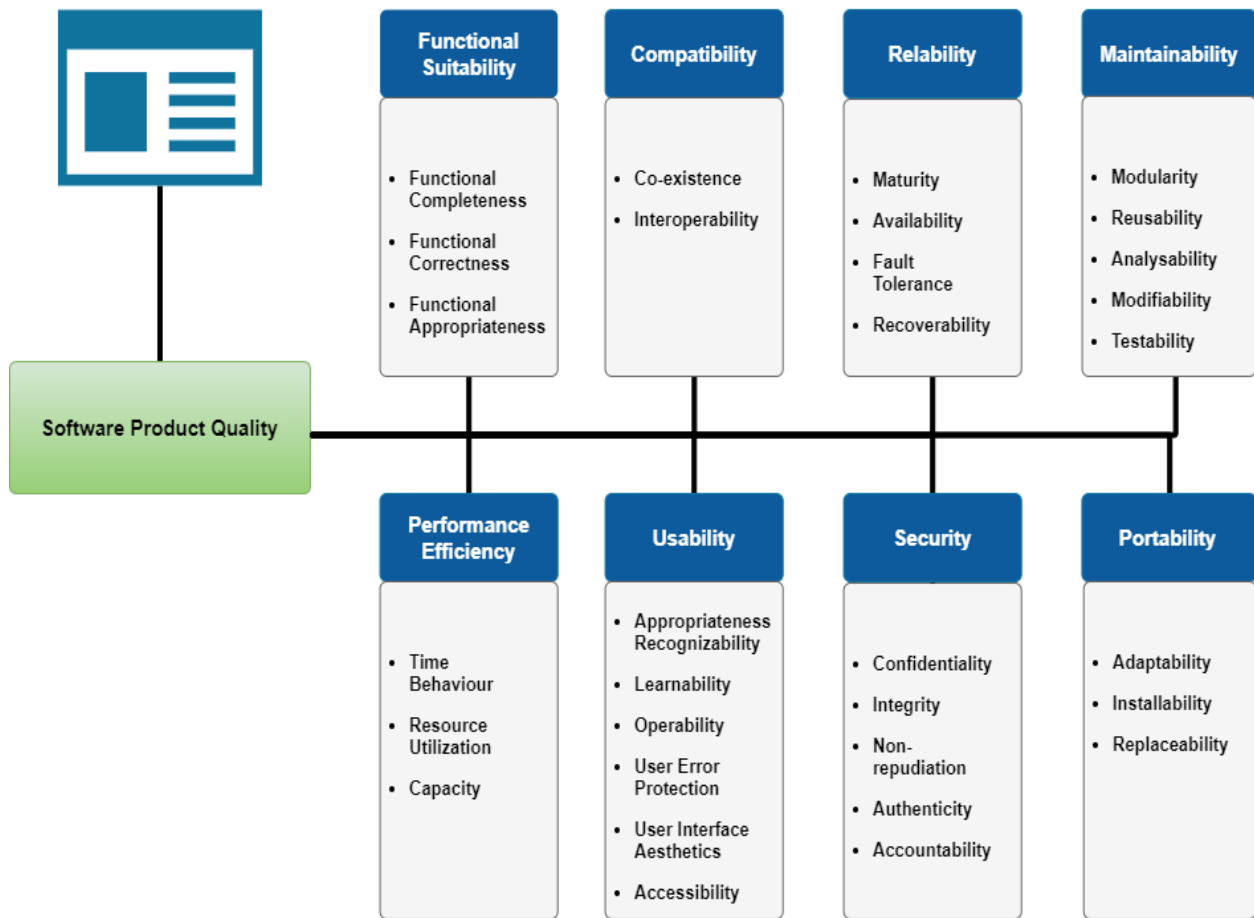


Figure 3.2.1: The Quality Model of ISO/IEC 25010

3.3 Analysis of the Quality Models

All the quality models from above share some common faults, which have been pointed out by several research papers, like being too generic and universal [14] [13][3][?]. [14] points out that the models are too general to meet the unique needs of specific software systems intended to e.g., mobile devices. The author also noticed that the universal models covered other needs that are not necessary for mobile apps, making it challenging to find the necessary parts for evaluating mobile apps. Consequently, the author presented a mobile quality model, based on the established quality models, that already focuses on the crucial qualities of mobile software. [?] summaries the disadvantage of the models from McCall, Boehm, ISO 9126, and ISO 25010; they combine as many perspectives of quality as possible into one framework, leading it to a high abstraction level. In conclusion, using only a generic quality model can make it challenging to assess software quality, especially for specific software systems accurately.

Considering that we are implementing design patterns for a mobile game with limited capabilities and memory, we extract and scope down the verification criteria from the quality model of ISO 25010 to Performance Efficiency and Maintainability, as we believe these two are the crucial qualities to assess. The following list below shows our selected

verification criteria or characteristics, and its sub-characteristics.

Verification Criteria

1. Performance Efficiency
 - (a) Resource Utilization
 - (b) Capacity
2. Maintainability
 - (a) Reusability
 - (b) Modularity

Note that we evaluate only the sub-characteristics Reusability and Modularity from Maintainability as its other sub-characteristics are out of our scope.

3.3.1 Criteria 1. Performance Efficiency

Performance Efficiency and its sub-characteristics are defined as following [11].

Performance Efficiency - ‘A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.’

- (a) **Time behaviour** — ‘Degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements’
- (b) **Resource Utilization** — ‘Degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements.’
- (c) **Capacity** — ‘Degree to which the maximum limits of a product or system parameter meet requirements.’

For our study, we define Performance Efficiency as the extent to which our app performs to the required performance and amount of resources needed. As our game app, with its implemented design patterns, allocate a high number of objects and also work with views, context, and activities, it is crucial to manage the resources and components appropriately and preserve the battery life. The consequences of low-performance efficiency for the app can be slow response time, freezing, crashing, and stutters in animations. As we will develop design patterns suited for the mobile environment, our goal is to see if the mobile design patterns, compared to the regular design patterns, enhance the app’s performance efficiency.

3.3.2 Criteria 2. Maintainability

Maintainability and our selected sub-characteristics of maintainability are defined as following [11].

Maintainability - ‘A set of attributes that bear on the effort needed to make specified modifications.’

- (a) **Modularity** — ‘Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.’

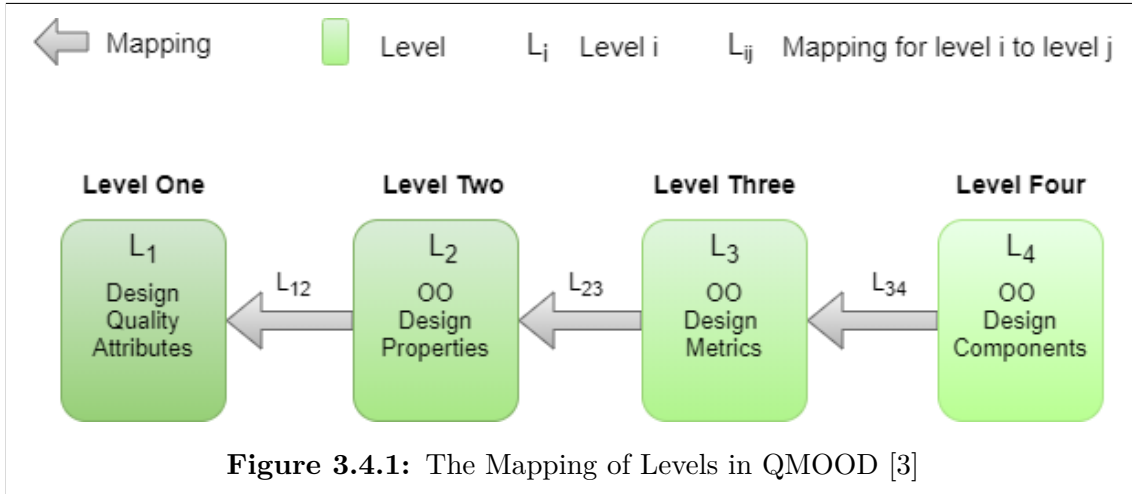
- (b) **Reusability** — ‘Degree to which an asset can be used in more than one system, or in building other assets’

For our study, we define Maintainability as the degree to which the design patterns can be reused for other mobile apps and the ease of using the design patterns such that a change to one component should not affect another. Maintainability can be significantly beneficial for mobile app development as it can, among other things, accelerate the development process, improve performance, avoid code duplication, and remove bugs and faults. The GoF says in [15], that design patterns promote easier maintainability, reusability, and flexibility. As we change the structure of the design patterns and develop mobile design patterns, it is crucial to test and see whether the maintainability of the mobile design patterns has maintained, improved, or worsen. Ideally, as mobile app development introduces challenges such as short time to market, developing an interactive app with varying user interfaces, and more, we want to prove that the mobile design patterns can improve maintainability, especially reusability and modularity.

3.4 Quality Metric Object-Oriented Model

The Quality Metric Object-Oriented Model (QMOOD), introduced by Bansiya et al. [3], connects information from the low-level source code metrics to the high-level design quality attributes like efficiency and reusability. Hence why we are using the QMOOD to verify further the quality of our mobile design patterns and its app. The model is meant for comparing software design that has similar requirements and objectives. The reason is that the internal characteristics of software design can be significantly different based on their requirements and objectives. In our study, we use the model to compare mobile apps with the same case, implemented with the same design patterns where the difference is that one app has more mobile design patterns. In conclusion, it is a valid case for using the model.

From [figure 3.4.1, p. 29], we can see that QMOOD consists of four levels and three relationships or mappings used to connect the levels. L_4 consists of the object-oriented design components that define the architecture of a object-oriented design with objects and classes, and the relationships between them. L_4 delivers the numerical information from the design components (objects, classes, methods and more) which is used to measure the design metrics in L_3 . This process is reflected by the relationship L_{34} . The design metrics are assigned to the design properties and their measurement is used to assess the properties. L_2 contains the design properties that may influence the quality attributes in L_1 .



In the following sections, we describe the levels (L_1 to L_4) and the mappings between them (L_{12} , L_{23} , L_{34}) from Bansiya et al. perspective.

3.4.1 Quality Attributes (L_1)

The QMOOD model describe six quality attributes: "functionality", "effectiveness", "understandability", "extendibility", "reusability" and "flexibility". Note the quality attributes are similar to the attributes of the ISO 9126 standard as its attributes were selected as the initial set in the QMOOD model. However, the authors decided to exclude the attributes they reviewed as not contributing to defining design quality and replacing the terms with terms they meant more descriptive and appropriate. For example, the term "maintainability" of the ISO standard was replaced by "understandability" as it concentrates more upon design characteristics[3].

Quality Attribute	Definition
Reusability	Reflects the presence of object-oriented design that allows a design to be reapplied to a new problem without significant effort.
Flexibility	Characteristics that allow the incorporation of changes in design. The ability of a design to be adapted to provide functionality related capabilities.
Understandability	The properties of the design that enables it to be easily learned and comprehended. This directly relates to the complexity of the design structure.
Functionality	The responsibilities assigned to the classes of a design, which are made available by the classes through their public interfaces.
Extendibility	Refers to the presence and usage of properties in an existing design that allows for the incorporation of new requirements in design.
Effectiveness	This refers to a design's ability to achieve the desired functionality and behaviour using object-oriented design concepts and techniques.

Table 3.4.1: Quality Attribute Definitions [3]

3.4.2 Design Properties (L_2)

Design properties are defined as "tangible concepts that can be directly assessed by examining the internal and external structure, relationship, and functionality of the design components, attributes, methods, and classes" [3]. The following table 3.4.2 describes the definitions of our design properties. Bansiya uses only one metric to assess a design property Hence, we can say that the mapping between the design property and metrics are direct and easy to follow. The metrics defined in [table 3.4.3, p. 33] are chronologically mapped to the design properties in table 3.4.2.

Design Property	Definition
Design Size	A measure of the number of classes used in a design.
Hierarchies	Hierarchies are used to represent different generalization-specialization concepts in design. It is a count of the number of non-inherited classes that have children in a design.
Abstraction	A measure of the generalization-specialization aspect of the design. Classes in a design which have one or more descendants exhibit this property of abstraction.
Encapsulation	Defined as the enclosing of data and behavior within a single construct. In object-oriented designs, the property specifically refers to designing classes that prevent access to attribute declarations by defining them as private, thus protecting the internal representation of the objects.
Coupling	Defines the interdependency of an object on other objects in a design. It is a measure of the number of other objects that would have to be accessed by an object in order for that object to function correctly.
Cohesion	Assesses the relatedness of methods and attributes in a class. Strong overlap in the method parameters and attribute types is an indication of strong cohesion.
Composition	Measures the "part-of", "has", "consists-of," or "part-whole" relationships, which are aggregation relationships in an object-oriented design.
Inheritance	A measure of the "is-a" relationship between classes. This relationship is related to the level of nesting of classes in an inheritance hierarchy.
Polymorphism	The ability to substitute objects whose interfaces match for one another at run-time. It is a measure of services that are dynamically determined at run-time in an object.
Messaging	A count of the number of public methods that are available as services to other classes. This is a measure of the services that a class provides.
Complexity	A measure of the degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships.

Table 3.4.2: Design Property Definitions [3]

3.4.3 Design Metrics (L_3)

The design properties identified in QMOOD are assessed by using one or more defined design metrics. While some well-defined metrics for assessing design properties exist, the authors argued that metrics that could not be calculated before nearly completing the implementation of classes, could not be used in the model. This resulted in them defining five new metrics which could only be calculated from design information: Data Access Metric (DAM), Direct Class Coupling (DCC), Cohesion Among Methods of Class (CAM), Measure of Aggregation (MOA), and Measure of Functional Abstraction (MFA). All the metrics identified by the authors are described in table 3.4.3 below.

Metric	Name	Description
DSC	Design Size in Classes	This metric is a count of the total number of classes in the design.
NOH	Number of Hierarchies	This metric is a count of the number of class hierarchies in the design
ANA	Average Number of Ancestors	This metric value signifies the average number of classes from which a class inherits information. It is computed by determining the number of classes along all paths from the "root" class(es) to all classes in an inheritance structure.
DAM	Data Access Metric	This metric is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class. A high value for DAM is desired. (Range 0 to 1)
DCC	Direct Class Coupling	This metric is a count of the different number of classes that a class is directly related to. The metric includes classes that are directly related by attribute declarations and message passing (parameters) in methods.
CAM	Cohesion Among Methods of Class	This metric computes the relatedness among methods of a class-based upon the parameter list of the methods. The metric is computed using the summation of the intersection of parameters of a method with the maximum independent set of all parameter types in the class. A metric value close to 1.0 is preferred. (Range 0 to 1).
MOA	Measure of Aggregation	This metric measures the extent of the part-whole relationship, realized by using attributes. The metric is a count of the number of data declarations whose types are user-defined classes.
MFA	Measure of Functional Abstraction	This metric is the ratio of the number of methods inherited by a class to the total number of methods accessible by member methods of the class. (Range 0 to 1)
NOP	Number of Polymorphic Methods	This metric is a count of the methods that can exhibit polymorphic behavior. Such methods in C++ are marked as virtual.
CIS	Class Interface Size	This metric is a count of the number of public methods in a class.
NOM	Number of Methods	This metric is a count of all the methods defined in a class

Table 3.4.3: Design Metrics Descriptions [3]


3.4.4 The Mapping of L_{23}


The following table 3.4.4 represents the mapping L_{23} , showing the relation between the design properties and the design metrics identified by Bansiya et. al. The higher the measurement of a design metric is, the higher value the design property gets. For instance, the property Design Size will increase as the number of classes increases.

Design Property	Derived Design Metric
Design Size	Design Size in Classes (DSC)
Hierarchies	Number of Hierarchies (NOH)
Abstraction	Average Number of Ancestors (ANA)
Encapsulation	Data Access Metric (DAM)
Coupling	Direct Class Coupling (DCC)
Cohesion	Cohesion Among Methods of Class (CAM)
Composition	Measure of Aggregation (MOA)
Inheritance	Measure of Functional Abstraction (MFA)
Polymorphism	Number of Polymorphic Methods (NOP)
Messaging	Class Interface Size (CIS)
Complexity	Number of Methods (NOM)

Table 3.4.4: Mapping of Design Metrics to Design Properties [3]

3.4.5 The Mapping of L_{12}

Table 3.4.5 displays the connections between the design properties and quality attributes. The quality attributes are evaluated in terms of the influence and weightings of the design properties. A design property can either negatively or positively influence the quality of an attribute. A  indicates a positive influence, while the opposite indicates a negative influence.

 Positive Influence
  Negative Influence

	Reusability	Understandab.	Effectiveness	Flexibility	Functionality	Extendib.
Design Size	↑	↓			↑	
Hierarchies					↑	
Abstraction		↓	↑			↑
Encapsulation		↑	↑	↑		
Coupling	↓	↓		↓		↓
Cohesion	↑	↑			↑	
Composition			↑	↑		
Inheritance			↑			↑
Polymorphism		↓	↑	↑	↑	↑
Messaging	↑				↑	
Complexity		↓				

Table 3.4.5: Design Properties relationships with Quality Attributes[3].

3.5 The Use of QMOOD in our Comparative Study

QMOOD can first of all be used for evaluating design patterns, as they are implemented in object-oriented design. Design patterns usually contains objects, classes and relationships between them. We especially chose QMOOD for its ability to analyze the metrics and design properties, up to the quality attributes. This is relevant for our study, as we will change the structure of our regular design patterns, in a attempt to develop mobile design patterns for enhancing performance efficiency and maintainability. The mobile version of the pattern will have different values of its design properties, and thus possibly quality attributes. The results of the QMOOD will further verify whether the changes made for developing the mobile design patterns, has improved the patterns in terms of maintainability and performance efficiency.

3.6 Analysis of QMOOD

When using QMOOD, the mapping of the levels and the levels itself has to be defined. However, the QMOOD quality model was made to be non-intrusive and adaptable; thus, it allows changes to be made to address different perspectives, objectives easily, and in our case, assessing mobile apps, implemented with design patterns [3]. We can adjust the design properties, design metrics, quality attributes, and even the mappings between them to fit our case. In our use case, we evaluate only three quality attributes of our mobile app; "reusability", "effectiveness", and "understandability", as these are the only relevant attributes to our selected criteria from the ISO 25010. For a detailed definition of our selected quality attributes, see [table 3.4.1, p. 30]. We adapted the design metrics provided by Bansiya to the metrics provided by the plugin MetricsReloaded. This was done due to the metrics provided by Bansiya were not available in the plugin, and also we changed the metrics for which we believed can indicate a better result. See the descriptions of the metrics in the Bansiya model in [table 3.4.3, p. 33]. The metrics from the plugin

are Java source code metrics, and we selected the metrics that are similar to the metrics in the Bansiya QMOOD model. A detailed description of the replacement metrics is in the table 3.6.1 right below.

Metric	Name	Domain	Description
NOC	Number of Classes	Package	This metric is a count of the total number of classes in the design.
DIT	Depth of Inheritance Tree	Class	This metric calculates the depth of the inheritance tree for each class. The depth is calculated as the number of inheritance steps between the class and object.
A	Abstractness	Package	This metric calculates the number of abstract classes and interfaces divided by all classes for each package.
AHF	Attribute Hiding Factor	Project	This metric calculates the degree of attribute encapsulation in a project. Essentially, it gives the ratio of how many classes an average field is from, other than the defining class.
I	Instability	Package	This metric calculates the instability of a package, defined as the package's efferent couplings divided by the sum of the package's afferent and efferent couplings. (Range 0 to 1) 0 indicates a maximally stable category, 1 indicates a maximally unstable category.
LCM	Lack of Cohesion of Methods	Class	This metric calculates on the degree of cohesiveness of a class. A variant of the LCOM metric designed by Hitz and Montazeri is used due to it being more appropriate for Java. The metric says that two methods of a class are related if they share a variable use, or one method calls another. It is then the count of the number of components of the method relation graph. (Range 0 to 1) 1 indicates a highly cohesive class, which can not easily be split into smaller classes.
CSA	Class Size (attributes)	Class	This metric calculates the total number of attributes or fields for each class. Static fields inherited from superclasses are not counted for purposes of this metric.
OMM	1 - number of overridden methods / number of methods	Class	This metric is the ratio of the number of methods inherited by a class to the total number of methods accessible by member methods of the class. (Range 0 to 1)
NOM	number of overridden methods or polymorphism factor	Class	This metric calculates the number of times each non-abstract method is overridden.
CSO	Class Size Operation	Class	This metric calculates the total number of operations (or methods) for each class. Static methods inherited from superclasses are not counted for purposes of this metric.
WMC	Weighted Method Complexity	Class	This metric calculates the total cyclomatic complexity of the methods in each class.

Table 3.6.1: Descriptions of our Adapted Design Metrics site MetricsReloaded

As we can see, most of our replacement metrics express similar information to the Bansiya metrics, and some of the metrics are alike but have different names. Some of the metrics like OOM, Abstractness, and Instability were selected following the Adapted QMOOD model of [18], because they too express similar information to the Bansiya metrics. The metrics which are noticeably different are the last metric of both tables, namely Number of Methods (NOM) and Weighted Method Complexity (WMC). Compared to NOM, WMC calculates the total cyclomatic complexity of all the methods in each class instead of counting all the methods. Cyclomatic Complexity is a metric, defined by McCabe[20], used for evaluating the complexity of a program or an algorithm in a method. Essentially, the metric counts the number of decisions in a method, and the more decisions a method has, the more complex it is. Generally, a method with low cyclomatic complexity is better, and easier to maintain and understand, and vice versa. The reason for the replacement was that we think WMC is a better indicator of the complexity as WMC sums up the cyclomatic complexity of all the class methods compared to NOM, which counts the number of class methods.

In our case, we want the domain of the metrics to be either package or class as we will first calculate the metrics of the design patterns one by one and then the project as a whole. Unfortunately, we could not find an alternative metric from the plugin for the Encapsulation where the domain is package or class. Our replacement metric AHF is adequate for measuring the Encapsulation, but the metric calculates the degree of attribute encapsulation of the whole project. As a result, we will use the value of this metric to assess Encapsulation, only when we are analyzing the project as a whole. When we analyze the design patterns one by one, the calculation will be neutral; thus, Encapsulation will have value 1.

3.6.1 The Mapping of L_{23}

The following table 3.6.2, shows the relation between the design properties and our replacement of design metrics. This process represents the mapping L_{23} .

Design Property	Derived Design Metric
Design Size	Number of Classes (C)
Hierarchies	Depth of Inheritance Tree (DIT)
Abstraction	Abstractness (A)
Encapsulation	Attribute Hiding Factor (AHF)
Coupling	Instability (I)
Cohesion	Lack of Cohesion of Methods (LCM)
Composition	Class Size Attributes (CSA)
Inheritance	1 - Number of Methods Overriden / Number of Methods (OMM)
Polymorphism	Number of Methods Overriden (NOOC)
Messaging	Number of Methods (METH)
Complexity	Weighted Method complexity (WMC)

Table 3.6.2: Design Metrics for Design Properties [3]

3.6.2 The Mapping of L_{12}

 Positive Influence
  Negative Influence


















	Reusability	Understandab	Effectiveness
Design Size			
Hierarchies			
Abstraction			
Encapsulation			
Coupling			
Cohesion			
Composition			
Inheritance			
Polymorphism			
Messaging			
Complexity			

Table 3.6.3: Design Properties connections with Quality Attributes[3].

As we can see from the table above, we changed the Hierarchies property to negatively effect the Effectiveness, compared to the positive influence in Bansiya’s model. The change was made due to the design property is now being measured by the metric Depth of Inheritance tree. According to [17], the longer the inheritance tree, the danger of having extra overhead increases, which can consume more resources then necessary, thus resulting a negative influence for the Effectiveness.

3.7 Measurement Process

In this section, we describe how we use the QMOOD model and the tools, plugin, and methods to evaluate our apps and its implemented design patterns towards our verification criteria from ISO 25010 and QMOOD model.

3.7.1 Our Measurement Process of QMOOD

In step L_{12} , Bansiya uses a computation formula for computing the value of the quality attributes. The formula uses weighted design properties in combination to build one quality attribute. As we can see from [table 3.7.1, p. 39], the weighing of the design properties can either positively or negatively influence the quality attribute. For example, the design size has a positive effect on the reusability as the more classes we have, the more

we can reuse. On the other hand, for understandability, the design size has a negative effect as the more classes we have, the size of the app increases. Thus it will be harder to maintain and understand the app. The initial value of the weighted design properties had, depending on if it has a positive or negative influence, ± 1 or ± 0.5 . In the end, the initial values of the weighted design properties are proportionally changed so that the sum of the new weighted design properties are equivalent to ± 1 . This ensured that the sum of weighted design properties influences the quality attribute ± 1 . This way of weighting was chosen by Bansiya to make it straightforward and easy to apply. The following table shows the computation formula for the quality attributes.

Quality Attribute	Index Computation
Reusability	$-0.25 * \text{Coupling} + 0.25 * \text{Cohesion} + 0.5 * \text{Messaging} + 0.5 * \text{Design Size}$
Understandability	$-0.33 * \text{Abstraction} + 0.33 * \text{Encapsulation} - 0.33 * \text{Coupling} + 0.33 * \text{Cohesion} - 0.33 * \text{Polymorphism} - 0.33 * \text{Complexity} - 0.33 * \text{Design Size}$
Effectiveness	$0.22 * \text{Abstraction} + 0.22 * \text{Encapsulation} + 0.22 * \text{Composition} + 0.22 * \text{Inheritance} + 0.22 * \text{Polymorphism} - 0.11 * \text{Hierarchies}$

Table 3.7.1: Computation Formulas for Quality Attributes [3]

In practice, the actual value of the metrics will have different ranges when calculating the sum of the different metric values. Hence why, we normalize the metric values with respect to their metric value of the first version, which is, in our case, the app that is implemented with the regular design patterns. The following example illustrated in the tables 3.7.2 and 3.7.3, shows how the actual metric values are replaced by their normalized values. The normalized values are calculated by dividing a metric value with its metric value in the first version of the app.

Metric	app 1.0	app 2.0	app 3.0
Design Size	18	35	42
Hierarchies	8	12	15
Abstraction	2	5	12
Encapsulation	0.23	0.38	0.54
Coupling	4.02	5.04	8.3
Cohesion	0.3	0.25	0.2
Composition	73	120	147
Inheritance	0.2	0.48	0.82
Polymorphism	13	34	59
Messaging	23	48	74
Complexity	57	72	81

Table 3.7.2: Example 1: Actual Metric Values for App

Metric	app 1.0	app 2.0	app 3.0
Design Size	1	1.94	2.33
Hierarchies	1	1.5	1.87
Abstraction	1	2.5	6
Encapsulation	1	1.65	2.34
Coupling	1	1.25	2.06
Cohesion	1	0.8	0.5
Composition	1	1.64	2.01
Inheritance	1	2.4	4.1
Polymorphism	1	2.6	4.53
Messaging	1	2.09	3.2
Complexity	1	1.26	1.42

Table 3.7.3: Example 1: Normalized Metric Values for App

After normalizing the metric values, we can calculate the values for the quality attributes. The next table 3.7.4 shows the computed value of the quality attributes based on the normalized values from the metrics. From the example results, we can see it indicates that the last version of the app has increased reusability and effectiveness. However, understandability has reduced, which is reasonable as the app size has increased.

Quality Value	app 1.0	app 2.0	app 3.0
Reusability	1	1.90	2.38
Understandability	-1	-2.34	-4.45
Effectiveness	1	2.31	3.90

Table 3.7.4: Example 1: Computed Quality Attribute Values for App

Metrics Reloaded

Metrics Reloaded is a plugin from the Android Marketplace that calculates and analyzes the source code towards a set of selected metrics. This plugin was used to measure the design metrics we have selected from the plugin [table 3.6.1, p. 36].

3.7.2 Our Measurement Process of ISO/IEC 25010

The app was tested towards our selected verification criteria from ISO/IEC 25010 with Android Profiler tools. Also, we measured the execution time of the design patterns.

Android Profiler

Android provides built-in profiler tools to provide real-time data about how one's app uses CPU, memory, energy, network, and battery resources [1]. The following are descriptions of two profilers we used in Android Studio and what selected sub-characteristics of ISO/IEC 25010 they evaluated.

- **CPU Profiler** allows us to inspect our app’s CPU usage in real-time while interacting with our app and inspect details in recorded method traces (Java), function traces (C/C++) and system traces. For evaluating Performance Efficiency, we record the CPU usage of our design patterns executed over a certain time.
- **Memory Profiler** will help us monitor the memory use of our app. It will help us to identify memory leaks and track memory allocations by capturing heap dumps. A heap dump is a snapshot that shows all the objects using memory at a particular time. Memory leaks can then be spotted by identifying objects still in memory that should not be. This can happen when the garbage collector is unable to reclaim a resource and return it to the heap. In Android, they can typically occur when a view, context, or activity reference is saved on a background thread. As we have been working with views, activities, and their context, we have to be aware of memory leaks for managing resources appropriately. We also have to be aware of unnecessary memory allocations, as we will allocate a high number of objects. Using memory profiler, we can then identify where in the code we allocate too many objects in a short amount of time or leaked objects. Avoiding an undesirable memory allocation can lead to better performance [1]. A good score in memory profiler will indicate a positive influence on the sub-characteristics Resource Utilization of Performance Efficiency.

Execution Time

The execution time of the design patterns and the app was measured by using the Java method `System.nanoTime()`, because it is accurate and straightforward. If a mobile design pattern has a better execution time than its regular version, then we can say that the mobile design pattern has improved in the sub-characteristics Time behavior of Performance Efficiency.

3.8 Summary

Our mobile design patterns will be evaluated towards two selected verification criteria or characteristics, namely performance efficiency and maintainability from the ISO model 25010. For performance efficiency, we evaluate three sub-characteristics: time behavior, resource utilization, and capacity. For our use case, we evaluate how the app performs with regards to the required performance and amount of resources needed. For maintainability, we focus only on the sub-characteristics modularity and reusability. We evaluate the degree to which the design patterns can be reused for other mobile apps and the ease of using the design patterns such that a change to one component should not affect another. To further assess the quality attributes, we use the QMOOD model to analyze and compare the source code metrics between the design patterns and link them further up to design properties and then quality attributes. Our goal of the evaluation is to see if the mobile design patterns compared to the regular version, have improved the design properties of the app, proving that the proposed mobile design patterns are more suited to mobile app development and enhance performance and maintainability of the app.

In the measurement process, we make use of Android Profiler Tools to profile and measure CPU usage and memory usage. In addition, we measure the execution time of the design patterns. For measuring the metrics in the QMOOD model, we use the plugin `MetricsReloaded`.

Chapter 4

Analysis

4.1 Introduction

In this paper, we create and implement different mobile design patterns. We first have to understand the differences between a mobile design pattern and a regular design pattern. Furthermore, we discuss the difference between implementing design patterns in a non-mobile and mobile environment as this can greatly affect the effectiveness of a mobile design pattern, and whether or not it is applicable to use. Lastly, we introduce some of the methods and concepts of how we create and implement mobile design patterns.

4.2 Mobile Environment vs. Non-Mobile Environment

Developing a mobile app in a mobile environment is different from developing a desktop app in a non-mobile environment. These differences can affect and introduce new challenges and requirements to design patterns. The following sections discuss the main differences between developing apps in the mobile environment versus non-mobile environments and how we believe it can affect a design pattern.

4.2.1 Data Persistence and Application Life Cycle

A user interacts with mobile apps in high frequency compared to desktop apps. The user also interacts usually with one app at a time. Hence why, mobile apps compared to desktop apps are often opened, closed, reopened, or hold on standby (to save battery life). This highlights the issue of data persistence for mobile apps [14]. For example, a running app currently in use may be interrupted by a phone call, leading to the app going into a paused state. Afterward, the app's latest changes can be lost if the user did not save them before the app was paused due to the app being killed in the background by the mobile system, especially if it runs out of memory. These state changes often happen in a mobile app and must be appropriately handled to guarantee data persistence. This is generally handled by an implemented application life cycle where the developer can define what the app does when its states changes. Most mobile platforms offer their life cycle, differing in syntax and semantics. In this paper, we introduce the application life cycle in Android. Android has four essential building blocks of an app or components: activities, services, broadcast receivers, and content providers [1]. The components serve their purpose and have distinct life cycles. The activity is an entry point for user interaction and represents the user interface for a single screen. The activity has callback methods that correspond to specific stages of its life cycle. These methods can be used to manage the life cycle of

the activity and the transitions between the states. The following sections describe the callback methods, and figure 4.2.1, on the next page, illustrates the activity's lifecycle.

OnCreate()

This is the first invoked method when an activity gets launched. This method must be implemented and initialize the essential elements of one's activity. Most of all, this is where one must define the layout of the activity's user interface.

OnStart()

This method is invoked after the `OnCreate()` method or when the activity gets restarted. This callback contains the final preparations for interacting with the user.

OnResume()

This callback is invoked just before the user begins interacting with the activity. Here, all user input is recorded, and the core functionality of the app implemented.

OnPause()

The `OnPause()` method is called when the activity gets interrupted and loses focus. In this method, the activity is still partially visible before it will, in most cases, go into Stopped state and no longer visible.

OnStop()

This callback is called when the activity is no longer visible to the user.

OnRestart()

If the activity is no longer visible but is returning to interact with the user again, `OnRestart()` is called. The callback will restore the state of the activity from when it was stopped.

OnDestroy()

If `OnRestart()` is not called, the next callback is `OnDestroy()`, which is the final callback method before the activity is completely destroyed.

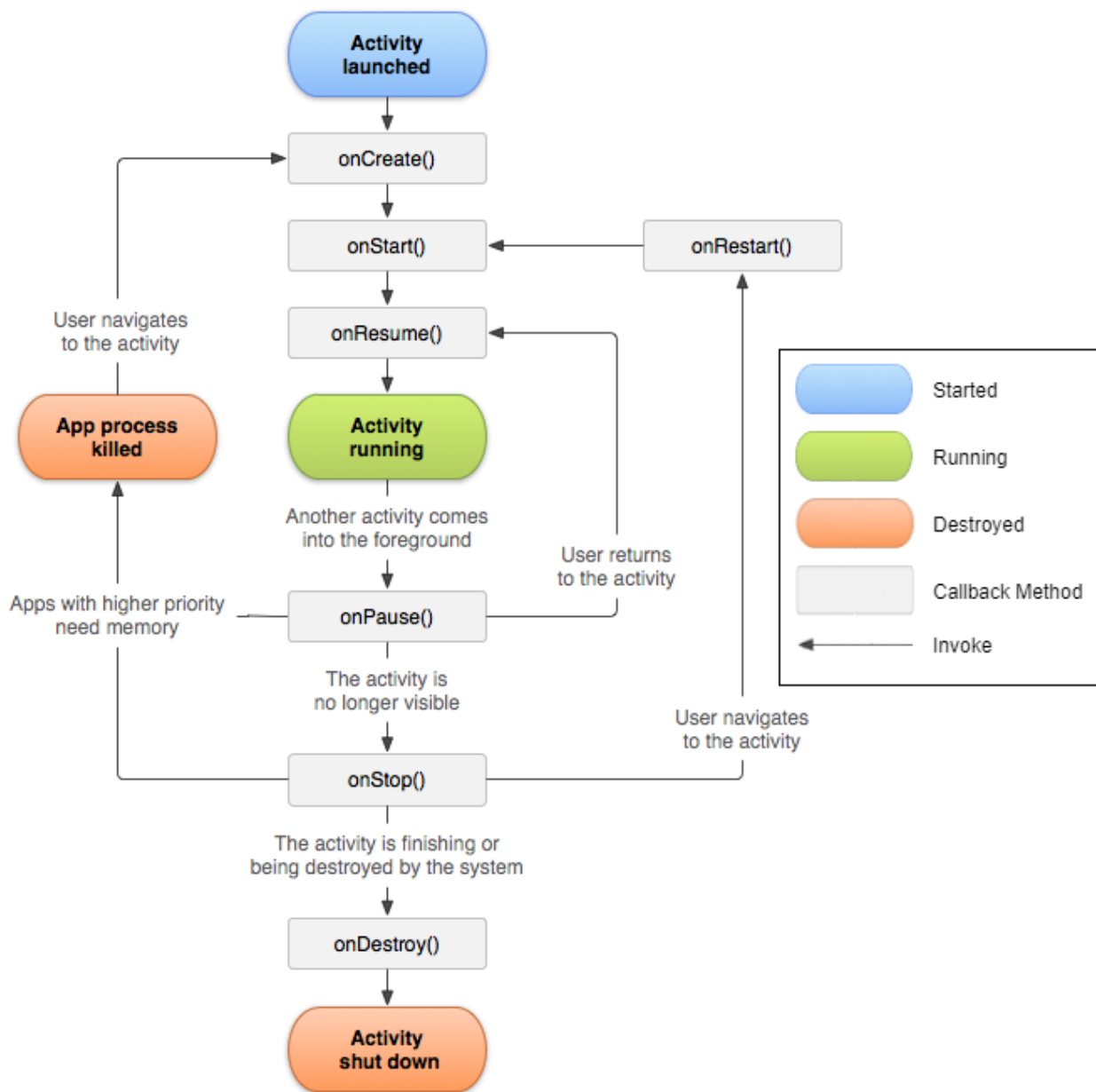


Figure 4.2.1: Illustration of the activity lifecycle [1]

When working with activities, it is also important to release all the activity resources before it is completely terminated. If not, memory leaks can happen. This was described more in detail in [chapter Evaluation Criteria, p. 41]. As we can see, when developing a mobile app in Android with design patterns, we have to properly handle its core components and its life cycles to guarantee data persistence and avoid other problems such as memory leaks.

4.2.2 Computing Perspective

Following is a list of the main issues with the development of apps in a mobile environment from a computing perspective and how we believe it can affect the design patterns.

- **Battery life and Computing Power** Due to battery life and limited computer

power, resources must be managed and used correctly and efficiently for mobile apps. For example, the brightness of the display affects how much battery power is used. For creating more power concerning design patterns, a solution might be to adapt the design patterns, using various programming techniques, efficient data types, and more, to make it more efficient. One has also to consider using heavy loading, which can make a mobile app more efficient by executing heavyweight functionality, functions that require a lot of processing power, on the server-side. This allows the client to access the Web Service API to invoke the functionality on the server [8].

- **Wireless Network and Communication** Mobile apps connected to the wireless network need to deal with a lack of reliable connectivity and how to operate when the app gets disconnected. There seems to be a lack of research for implementing design patterns for apps that requires reliable and excessive connectivity.
- **Portability** Due to limitations of storage capacity, mobile apps have to be designed to optimize data storage usage. Similar concept to heavy loading, a solution might be to store the data on the server and access it when needed. However, one must take into account that the information stored on the server will not be available while the mobile device is disconnected from the network.
- **Varying User Interfaces** Developing an interactive mobile app that requires multiple types of user interfaces can lead to unstructured design, such as code duplication, which can increase the size of the software and poor design. As mentioned in [4], the traditional MVC (model-view-controller) pattern is not sufficient, but one can modify and create a more efficient MVC pattern supporting more dynamic properties as shown in [8].

4.2.3 Software Development Perspective

The main issue we identified with developing apps in a mobile environment from a software development perspective is that there are different mobile platforms with their own defined architecture and components. For example, Android uses the Activity component to represent and manage the user interface, while Apple's iOS uses UIView, UIViewController, and more. Thus, a mobile design pattern for Android may not work or has to be tweaked for Apple's iOS. The platforms also use different programming languages. Android uses Java, while iOS uses mainly Swift, which is primarily written in C++. This can cause problems to mobile design patterns, e.g., a technique for memory saving used in design patterns can work differently for the two languages as Java uses garbage collector for allocating memory, while Swift uses Automatic Reference Counting (ARC). In conclusion, the difference in a programming language and mobile platform can affect the effectiveness of a mobile design pattern. This is further discussed in [section 4.4, p. 47].

4.3 Mobile Design Pattern vs. Design Pattern

As mentioned earlier in this paper, design patterns were originally targeted for computers and not mobile devices. Thus some patterns from the GoF book are not recommended when developing mobile apps [6]. There are design patterns that are more suitable than others, such as heavy loading, Flyweight, and Proxy [24]. Both heavy loading and Proxy can increase the efficiency of a mobile app. Heavy loading executes heavyweight

functionality on the server, While the Proxy pattern avoids duplication of objects by providing a substitute or a placeholder for another object.

The Flyweight pattern can reduce the computing power consumed by an app by instantiating smaller objects and sharing similar objects instead of instantiating new ones. For example, in this research paper [24], the results showed that the combination of both the Flyweight pattern and Proxy, provided that they are applied in the right context, improved the efficiency of a mobile app by decreasing both execution time and memory usage.

Design patterns that are not appropriate to use in a mobile environment where, e.g., power consumption and battery life are of concern, can be adapted to tackle these constraints, as demonstrated by author Kayun in his research paper [6]. We can conclude that mobile design patterns are either adapted versions of existing patterns or existing patterns that are suitable for a mobile environment such as Proxy and Flyweight, to overcome the limitations and challenges with mobile devices and the development of mobile apps.

4.4 Choosing the Right Platform for our Mobile Application

Platforms such as iOS, Android, and Microsoft have different programming languages and tools which can affect the effectiveness of the mobile design patterns and whether or not they are even profitable to use. Android Studio, which is the IDE for Google's Android, is written mainly in Java and some C and C++. However, the adapted Factory method suggested by Kayun [6] is most effective when it mixes the usage of struct and class. The Factory method was implemented in Microsoft's .NET framework, which is written in C#. In Android Studio, the datatype struct is not available for the developer, making the adapted Factory method less effective. We can try to replicate a struct in Java by creating a class where the methods and variables are all public. However, the main advantage of using a struct was that it is a value type, while class is a reference type [6].

4.5 Creating Mobile Design Patterns

We summarize the techniques and strategies for developing mobile design patterns in two methods: combining design patterns or using various optimization techniques and strategies. The methods are created and inspired by some research papers about optimization techniques and strategies for memory and energy consumption [7][17], and other papers that have developed various design patterns for mobile devices [8][6].

4.5.1 Method 1: Combining Design Patterns

Because design patterns are extensible and reusable solutions, we can combine the design patterns to solve a new problem or an existing problem, which is more complex. In [8], the author incorporated the design patterns Observer, Command, Composite, Mediator, and Strategy into the architectural pattern model-view-controller to solve the limitations and challenges of an interactive mobile app which can support dynamic properties and more.

4.5.2 Method 2: Using various Optimization Techniques and Strategies

In [17], the authors presented the following techniques and strategies for mobile apps to save memory space and lower memory consumption.

- **Combining several classes**

One can save memory quite dramatically if several classes containing methods are combined into one class. The paper showed two apps with two different structures where the first app was divided into 14 classes, and in the second, the same methods were in one class. The second alternative showed impressive 50% more savings.

- **Combining several images**

This is a similar memory-saving technique as the above, where we combine several images into one bigger image. Instead of having to store and draw individual images, one can extract and draw the desired part from a single image. The paper shows that this technique is especially effective for smaller images as the overhead of the image files is clearer. Also, if the images are similar in content, then the compression algorithm might achieve greater results.

- **Avoiding small classes**

Classes contain considerable overhead. One can save more memory by avoiding the creation of small classes or inner classes. Inner or nested classes are recognized as classes also, so they too contain overhead. Normally, inner classes contain small functionality, so having the extra overhead is relatively large in most cases.

- **Managing inheritance** Inheritance can take up more memory than necessary since all parent variables are present in its child objects, all though they may be unnecessary. Additionally, the parent class has to be loaded when creating a child object if it has not been loaded yet. Inheritance should be carefully used in cases where it is necessary and useful. Especially, one should avoid unnecessary methods or variables.

- **Avoiding dependencies** Similar to inheritance, one can save memory by avoiding unnecessary references between classes, as it saves at least one item from the classes constant pool and sometimes the loading of the other class.

In [7], the authors proposed optimized strategies for developing software in an energy concerned environment, a familiar environment for mobile devices, based on quantitative outcomes. In their report, they measured and compared the power consumption of significant usages in Object-oriented programming such as classes, prototypes, attributes, methods, and more. Some impressive results were the power consumption of using class or struct, and whether or not using the keyword static. In [6], the author realized that creating a struct rather than a class where the size did not exceed a certain amount was more efficient. The author also found out that certain combinations of static and non-static methods were also more efficient, and the static method consuming less CPU than the non-static method. In conclusion, we can use these research articles as references to mix usages of keywords and data types for developing a more energy concerned design pattern. However, it is not possible for us to use the datatype struct as the programming language of Android is Java. The following table is a summary of the results in [7].

Issue	Option 1	Option 2	Option 3	Recommendation
Group of attributes creation	class	struct		class
Class prototype	abstract class	interface		any
Class attribute	static	dynamic		static
Class method	static	dynamic	dynamic anonymous	dynamic
Dynamic local variable class	bare use	this		any
Attribute accessibility	private	public	protected	private or public
Method accessibility	private	public	protected	any

Table 4.5.1: Summary of Strategies for Power Conscious System [7]

4.6 Summary

We have discussed the main differences between the mobile environment and non-mobile environment, and how that may affect the implementation of design patterns and the considerations, we may have to take. Some key differences were that in mobile app development, we encounter the issue of data persistence and working with application life cycles. The lifecycle of an activity in Android can transition between different states. To guarantee data persistence, avoiding the loss of data, we have to properly handle the transition states, especially where the activity or the app itself gets destroyed. We have also to ensure that the resources of activity are successfully released so that we avoid memory leaks. From researching several papers about optimization techniques and strategies for memory and energy consumption, and other papers that have implemented mobile design patterns, we summarized the information into two methods for developing mobile design patterns: 1. combining design patterns or 2. using various optimization techniques and strategies. Some of the techniques introduced were having an optimized combination of data-types and keywords, converting multiple images into one, converting methods to static, and combining several classes. One of the techniques, however, used the keyword struct, which is not available in the Java language, which we are using in our use case. One solution was to replace the struct with a public class containing only public members, leading to the difference being that structure is a value type, while the public class is a reference type.

Chapter 5

Case

In this chapter, we introduce the case of our project: a mobile maze game for which our design patterns will be implemented. We illustrate an abstract view of our app to describe the overall architecture and flow, and how the game app will work in Android.

5.1 Overall Architecture

Our maze app is a continuation from the maze example in the GoF book [15]. As defined in the book, a maze consists of a set of rooms. The rooms consist of four sides (north, east, south, west) that can have a door or a wall. Entering an open door can then lead us to the next room. A key difference is that we created a maze app with several design patterns implemented for the same app, making it more of a complete game. In our continuation, the maze game consists of one player and a room, which consists of 5 sides where the additional side is the center of the room. In the center of the room, there can spawn coins which can be picked up by the player. An engine is also implemented and contains all the objects created in the maze app. As we can see from the flowchart of our overall architecture 5.1.1, the Engine is responsible for updating the app and adding the game objects into the View. The activity class `GameActivity` is responsible for processing incoming requests from the View. From the model-view-controller perspective, the Engine can be seen as the model as it contains all the game objects with its operations, while the game activity acts as a controller that interacts with the Engine to perform the user requests, provided through user inputs from the View. The flowchart can be regarded as a loop that will typically stop on the condition that the game is completed. If so, `GameActivity` will finish its activity.

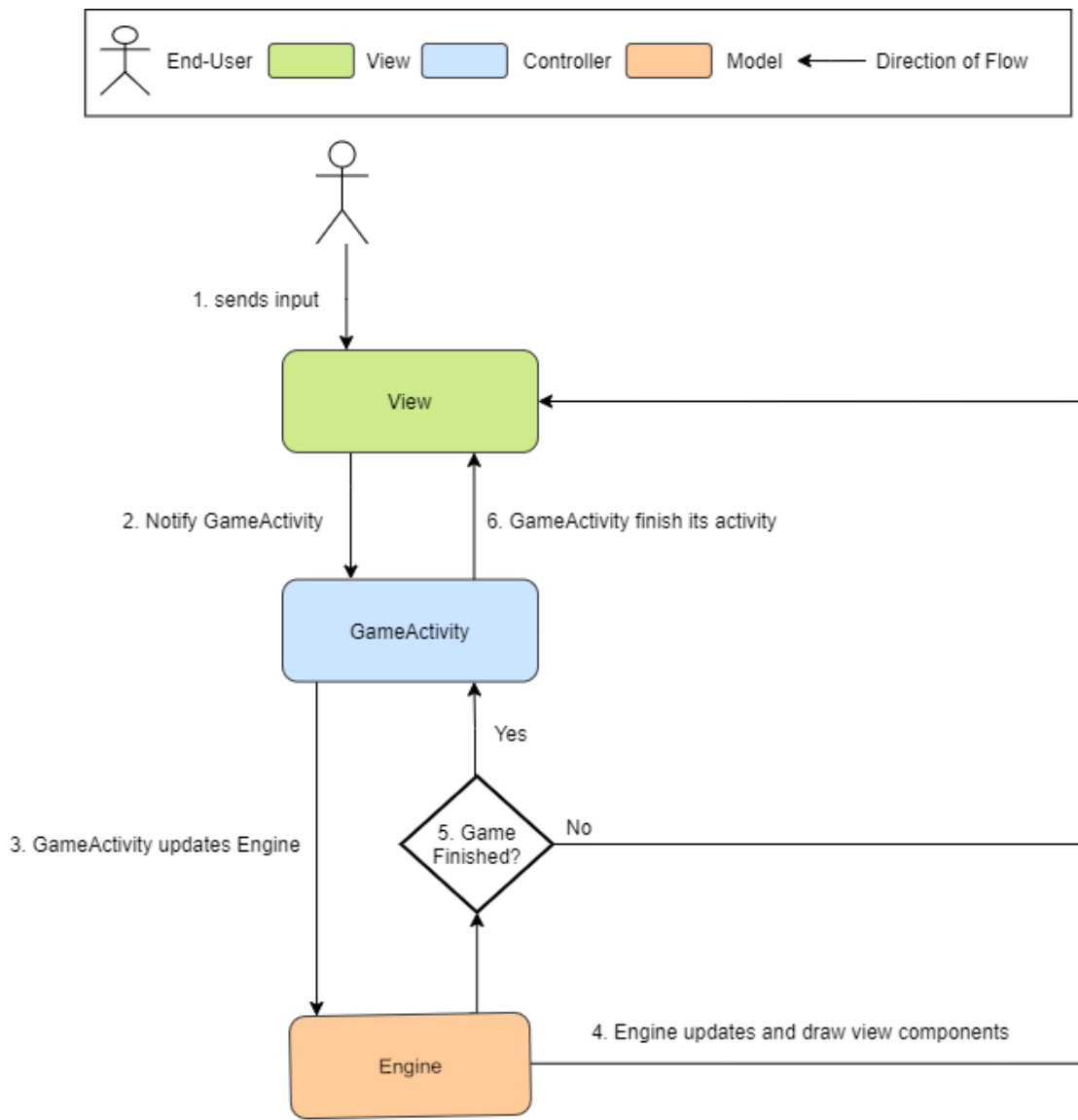


Figure 5.1.1: Flowchart of the Overall Architecture

5.2 The Main Components

Following is an overview of the main components of the maze app, and these components will be used and referenced in the implementation chapter of our design patterns.

- **Engine** contains all the game objects in the app. The Engine is responsible for constantly updating and drawing the game objects as long as the game is not finished. The update method updates the player and checks for collisions between the game objects, while the draw method adds the image view of the game objects to the user interface.
- **Game Activity** acts as an interface between the user interface components and the game objects. The activity listens and processes incoming requests from the View (UI). When processing the requests, data from the Engine and its game objects gets manipulated.
- **Player** is a character in the game that the user controls. A player object can move in four directions: north, south, east, or west, and open doors and pick up items from the ground.
- **Maze** contains rooms with doors and walls and picks up items like coins. A maze is created in the Engine.
- **Room** consists of a total of five sides. The four sides (north, east, south, and west) of a room can be delegated to a wall or a door, while the last side, center, can contain a pickup item.
- **Door** can either be locked or opened. If it is opened, then the player can open the door to enter the next room. If a player tries to enter a locked door or a wall, the player will end up hitting the door or the wall.
- **Pickup** is items that can be picked up by the player. In this case, a pickup item can be a handful coin of bronze, silver, or gold.

The implemented design patterns in our maze app were Abstract Factory, Command, Visitor, Strategy, and Singleton. Each pattern had a specific role or responsibility for implementing the maze app. Following is a list of what the implemented design patterns were used for.

Abstract Factory was used for creating different mazes depending on which factory (BombedMazeFactory or StandardMazeFactory) is chosen.

Command was used for sending and handling different commands from the player. For example, the player can command to enter a door or a wall. The command pattern will then encapsulate this request (as an object) and handle it.

Visitor was used for performing different operations when a player visits or collides with an object. For example, if the player collides with a pickup item, then the visitor pattern will calculate the value of the pickup item.

Strategy defined different algorithms for taking a screenshot of the maze app and saving the image in different file formats like PDF and JPEG.

Singleton was used for ensuring that the Engine class had only one instance and provided a global access point to it, and the context of the GameActivity.

5.3 Summary

For developing our mobile app and design patterns, we continue the maze example from the GoF book [15]. Our two main components, Engine and GameActivity can be seen as the model and controller of the model-view-controller pattern. The Engine contains all the game objects, updates the objects and draws them on the View. The GameActivity is responsible for interacting with the Engine to process incoming user requests from the View. Abstract Factory was used for creating different types of mazes and their objects. The Command pattern was responsible for handling the different commands from the player, and the Visitor was used to perform the different operations of the maze objects. Strategy defines different algorithms for saving a screenshot of the game in various file formats. Singleton was used to ensuring there is only one Engine instance and provided a global access point to the game activity context.

Chapter 6

Implementation

In this chapter, we analyze and compare the implementation of our regular and mobile design patterns. We discuss the changes we made for developing mobile design patterns and issues and challenges which we encountered for developing a mobile app in Android Studio. We start by analyzing the first design pattern implemented, then the second, and so forth. For the full implementation details of the projects, please see [Appendix A, p. 93].

6.1 Abstract Factory vs. Mobile Abstract Factory

The regular abstract factory was easy to implement, and we followed the general UML class diagram of the pattern. As mentioned, the pattern was responsible for creating the different mazes and the player. Each sub maze factory implemented a maze factory interface with factory methods for creating the maze objects (door, wall, room, and pick up item) and the player object.

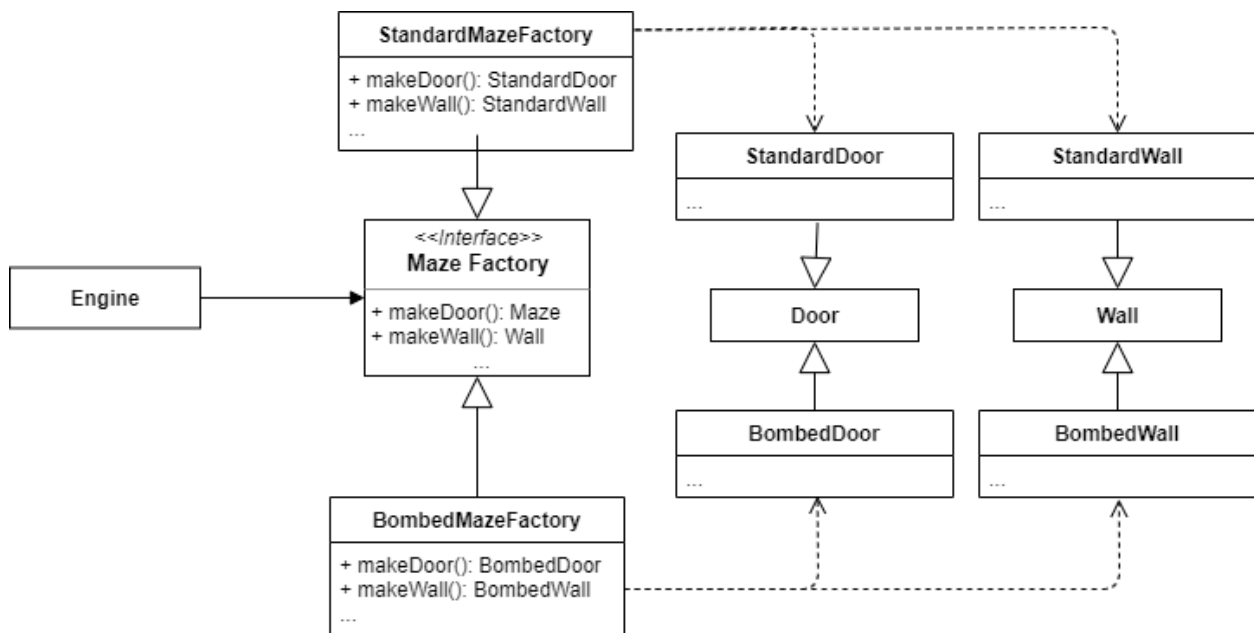


Figure 6.1.1: Our Regular Abstract Factory Implementation in UML Class Diagram

```

public interface MazeFactory {
    Maze makeMaze();
    Wall makeWall(Direction d, Context c);
    Room makeRoom(int roomNumber);
    Door makeDoor(Room r1, Room r2, Direction d, Context c);
    Pickup makeRandomPickUp(Context c);
}
public class StandardMazeFactory implements MazeFactory {
    @Override
    public Door makeDoor(Room r1, Room r2, Direction direction, Context
        context) {
        return new StandardDoor(r1, r2, direction, context);
    }
    @Override
    public Maze makeMaze() {
        return new Maze();
    }
    @Override
    public Room makeRoom(int roomNumber) {
        return new Room(roomNumber);
    }
    @Override
    public Wall makeWall(Direction direction, Context context) {
        return new StandardWall(direction, context);
    }
    @Override
    public Pickup makeRandomPickUp(Context c) {
        Random random = new Random();
        int randWeight = random.nextInt(100);
        switch (random.nextInt(3)){
            case 0: return new Bronze(c, randWeight);
            case 1: return new Silver(c, randWeight);
            case 2: return new Gold(c, randWeight);
            default: return new Bronze(c, randWeight);
        }
    }
}
}

```

Code Snippet 6.1: Sample Code of Abstract Factory

However, as the Abstract Factory pattern was not meant for mobile devices, we tried to adapt it by using the techniques discussed in chapter 4. For developing the mobile Abstract Factory pattern, we converted all the factory methods that create and return game objects to static methods. Static methods can prove to be more cost-efficient than non-static if the size of the objects. However, making the methods static made it impossible for the sub-factories to override the interface methods in MazeFactory. Our proposed solution was then to make MazeFactory a public class, instead of an interface, that contains a FactoryType enum that defines which sub-factory is to be used. We do not need to know which sub-factories are used, because we will only need to create a MazeFactory instance and not instances of specific sub-factories. That is delegated to the MazeFactory instance with its FactoryType enum. The following figures illustrate these implementation differences between our mobile Abstract Factory and regular Abstract Factory.

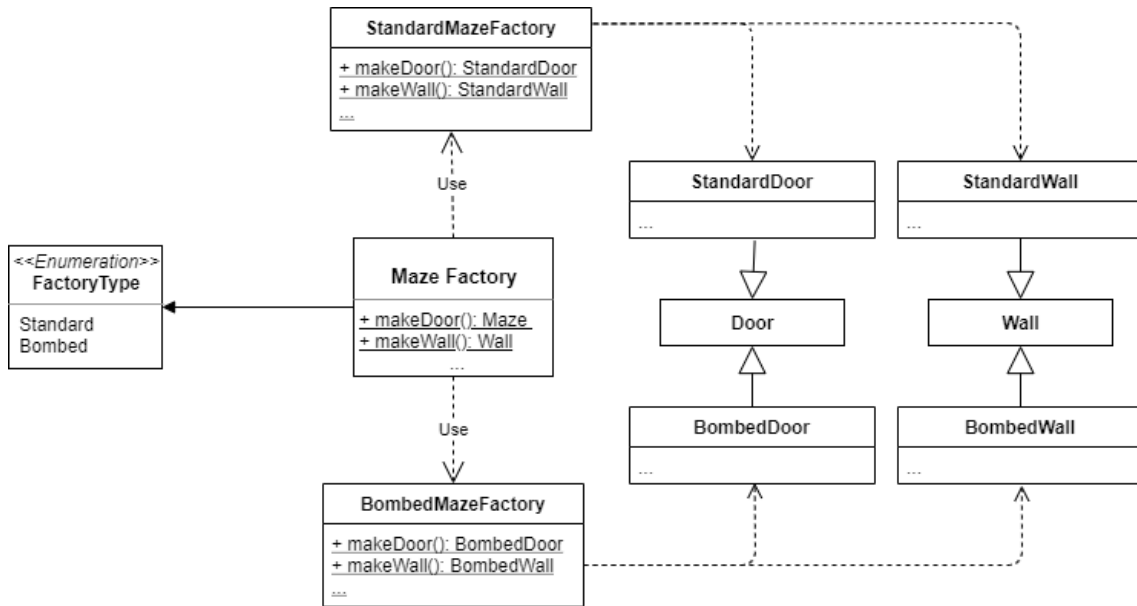


Figure 6.1.2: UML Class Diagram of Mobile Abstract Factory

```

public class MazeFactory {
    public static Maze makeMaze() {
        Maze maze = null;
        switch (Setting.factoryType) {
            case Bombed:
                maze = BombedMazeFactory.makeMaze();
                break;
            default:
                maze = StandardMazeFactory.makeMaze();
        }
        return maze;
    }
    public static Wall makeWall(Direction d, Context c) {
        Wall wall = null;
        switch (Setting.factoryType) {
            case Bombed:
                wall = BombedMazeFactory.makeWall(d, c);
                break;
            default:
                wall = StandardMazeFactory.makeWall(d, c);
        }
        return wall;
    }
    public static Door makeDoor(Room r1, Room r2, Direction d, Context c)
    {...}
    public static Room makeRoom(int roomNumber) {...}
    public static Pickup makeRandomPickUp(Context c, double weight) {...}
}

public class StandardMazeFactory {
    public static Maze makeMaze() {
        return new Maze();
    }
    public static Door makeDoor(Room r1, Room r2, Direction direction,
        Context context) {

```

```
        return new StandardDoor(r1, r2, direction, context);
    }
    public static Room makeRoom(int roomNumber) {
        return new Room(roomNumber);
    }
    public static Wall makeWall(Direction direction, Context context) {
        return new StandardWall(direction, context);
    }
    public static Pickup makeRandomPickUp(Context c, double weight) {...}
}
```

Code Snippet 6.2: Sample Code of Mobile Abstract Factory

In mobile abstract factory, we also use the memory saving technique by converting the images of the game objects into a single image. For example, in regular abstract factory, when our player is changing directions, individual images for a specific direction east, west, south or north is drawn. But in mobile abstract factory, the different images of a player for moving left, up, down or right are combined into a single image player.png, resulting in a sprite sheet. Now, instead of drawing individual images, we can extract and draw the desired part from a single image.



Figure 6.1.3: Sprite Sheet of Player

6.2 Command vs Mobile Command

In our regular Command pattern, we have two sub-classes EnterDoorCommand and EnterWallCommand of the Command interface. The EnterDoorCommand checks if the current game object collided with the player is a door and if so, enter the door to the next room if it is unlocked. Similar to the EnterWallCommand, this command will check if it is a wall and if so, the player will not be able to enter it.

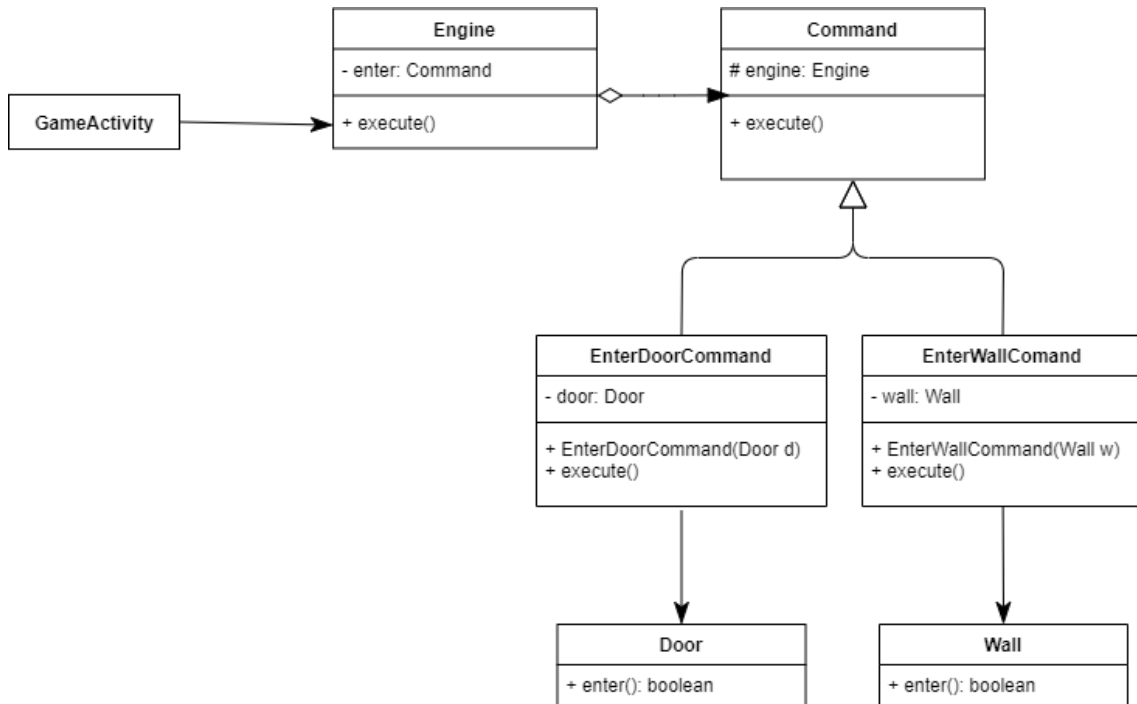


Figure 6.2.1: UML Class Diagram of Command

```

public abstract class Command {
    protected Engine engine = Engine.getInstance();
    public abstract void execute();
}
public class EnterWallCommand extends Command {
    Wall wall;
    public EnterWallCommand(Wall wall) {...}

    @Override
    public void execute() {
        if(wall == null){
            return;
        }
        if(wall.intersects(engine.getPlayerSprite())) {
            wall.enter();
        }
    }
}
public class EnterDoorCommand extends Command {
    Door door;
    public EnterDoorCommand(Door door) {...}

    @Override
    public void execute() {
        if(door == null){
            return;
        }
        if(door.intersects(engine.getPlayerSprite())){
            if(door.enter()){
                engine.movePlayerToRoom(door.otherSideFrom(engine.getCurrentRoom()));
            }
        }
    }
}
  
```

```

    }
}
}

```

Code Snippet 6.3: Sample Code of Command

As we can see from the sample code 6.3, the sub-classes and their commands are similar, simple and small. Our proposed solution for the mobile command pattern was then to combine these small commands into one command, namely EnterCommand. The command will now instead check if there is a StaticSite collided with the player and if the object is possible to enter. If so, the command assumes it is a unlockable door and moves the player to the next room of the door.

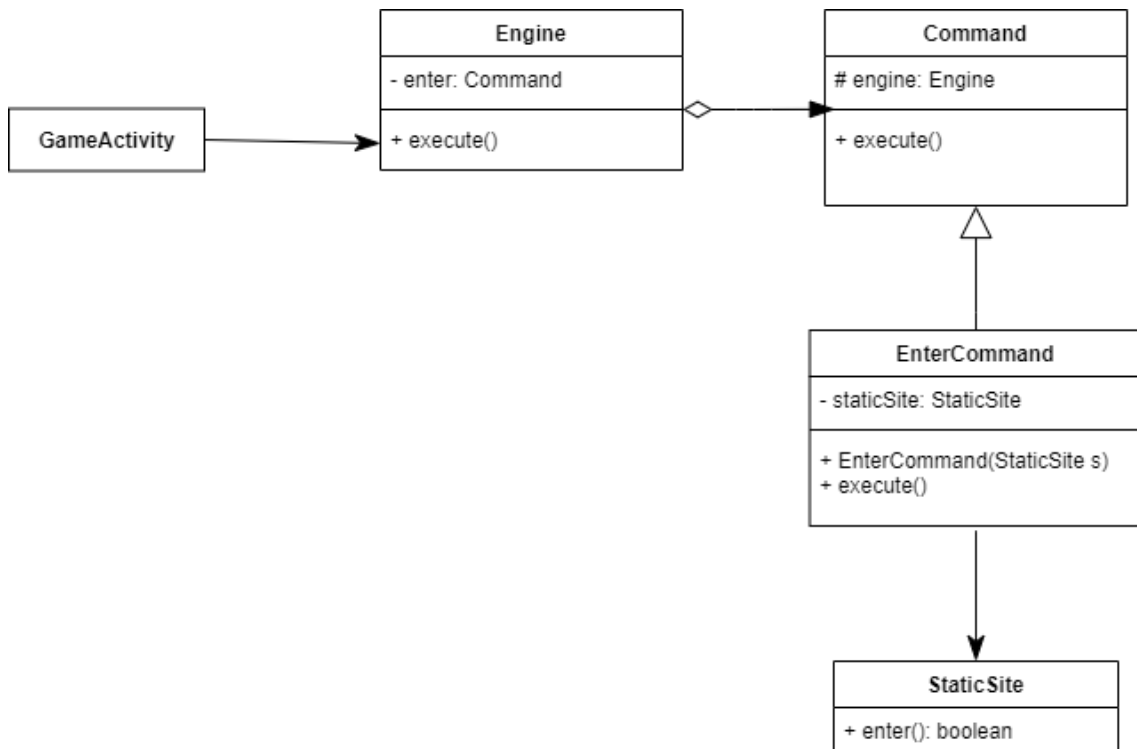


Figure 6.2.2: UML Class Diagram of Mobile Command

```

public abstract class Command {
    public Engine engine = Engine.getInstance();
    public abstract void execute();
}
public class EnterCommand extends Command {
    StaticSite staticSite;
    public EnterCommand(StaticSite sS) {...}

    @Override
    public void execute(){
        if(this.staticSite == null){
            return;
        }
        if(staticSite.intersects(engine.getPlayerSprite()) && staticSite.
            enter() == true) {
            Door door = (Door) staticSite;

```

```

engine . movePlayerToRoom ( door . otherSideFrom ( engine . getCurrentRoom ( ) ) )
;
}
}
}
}

```

Code Snippet 6.4: Sample Code of Mobile Command

6.3 Visitor vs. Mobile Visitor

In our visitor pattern we perform different operations when the player visits or collides with game objects. For example, when the player collides with either bronze, silver or gold objects, the objects gets visited and a operation for calculating their value is performed. In the regular visitor pattern, we have specific visit operation methods for calculating each type (bronze, silver and gold) of pick up item. The visit operation method calculates the value of a pick up item by multiplying the weight and value of the item.

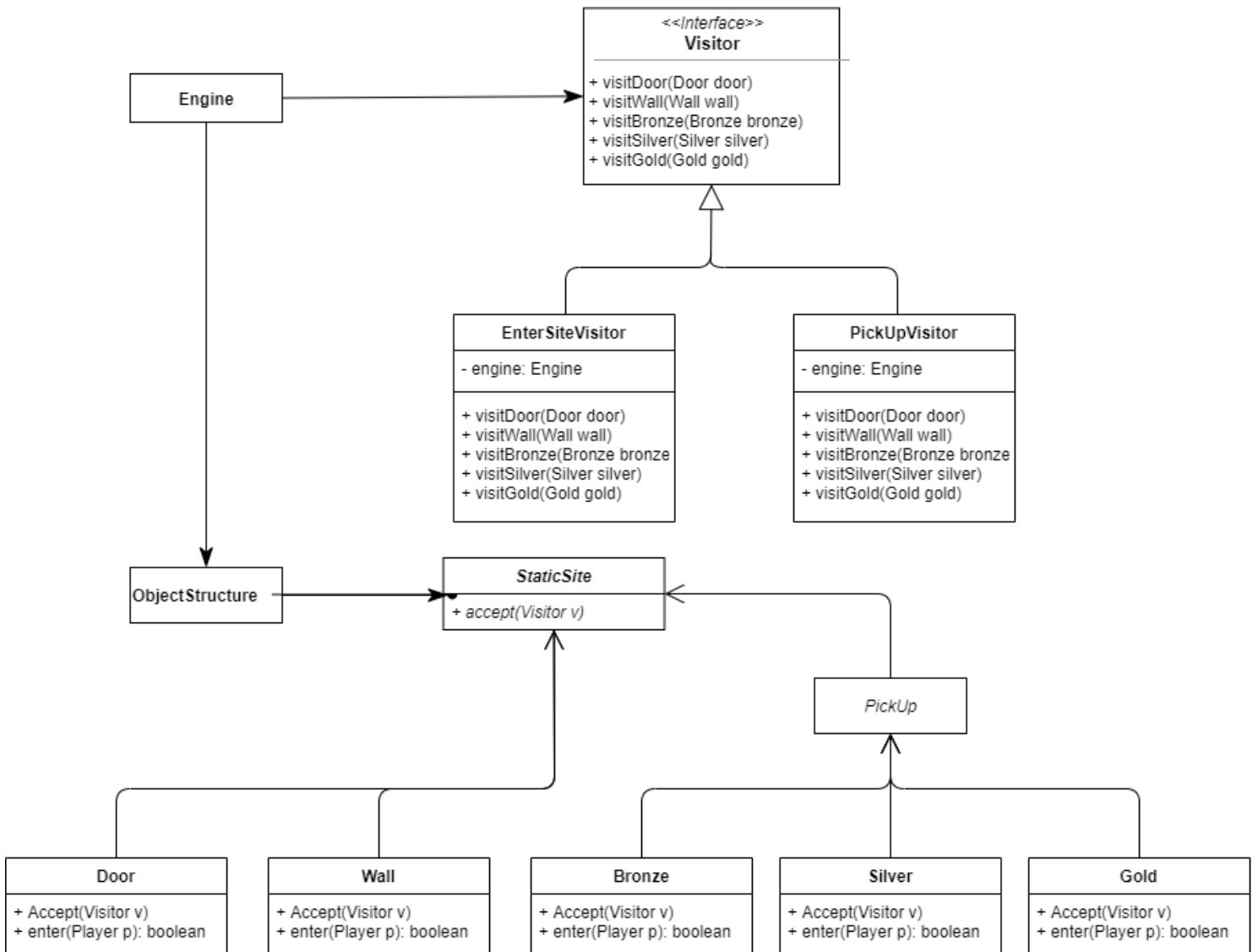


Figure 6.3.1: UML Class Diagram of Visitor

```

public class PickupVisitor implements Visitor {
    @Override
    public void visitDoor(Door door) {

    }
    @Override
    public void visitWall(Wall wall) {

    }
    @Override
    public void visitBronze(Bronze bronze) {
        double value = bronze.getWeight() * bronze.getValue();
        Engine.getInstance().incrementPlayerMoney(value);
    }
    @Override
    public void visitGold(Gold gold) {
        double value = gold.getWeight() * gold.getValue();
        Engine.getInstance().incrementPlayerMoney(value);
    }
    @Override
    public void visitSilver(Silver silver) {
        double value = silver.getWeight() * silver.getValue();
        Engine.getInstance().incrementPlayerMoney(value);
    }
}

```

Code Snippet 6.5: Sample Code of Visitor

In the visitor operation methods, we saw a possible improvement in efficiency by taking advantage of the polymorphism concept. Hence, the proposed solution when developing the mobile visitor pattern was to combine the visit operations methods for the specific pick up items and the static sites to a single method. For instance, instead of calling the weight and value method of the child class of the parent Pickup class, we will call the necessary methods from the parent itself. Due to polymorphism, the program will know which specific method to call when a child class like gold is visited.

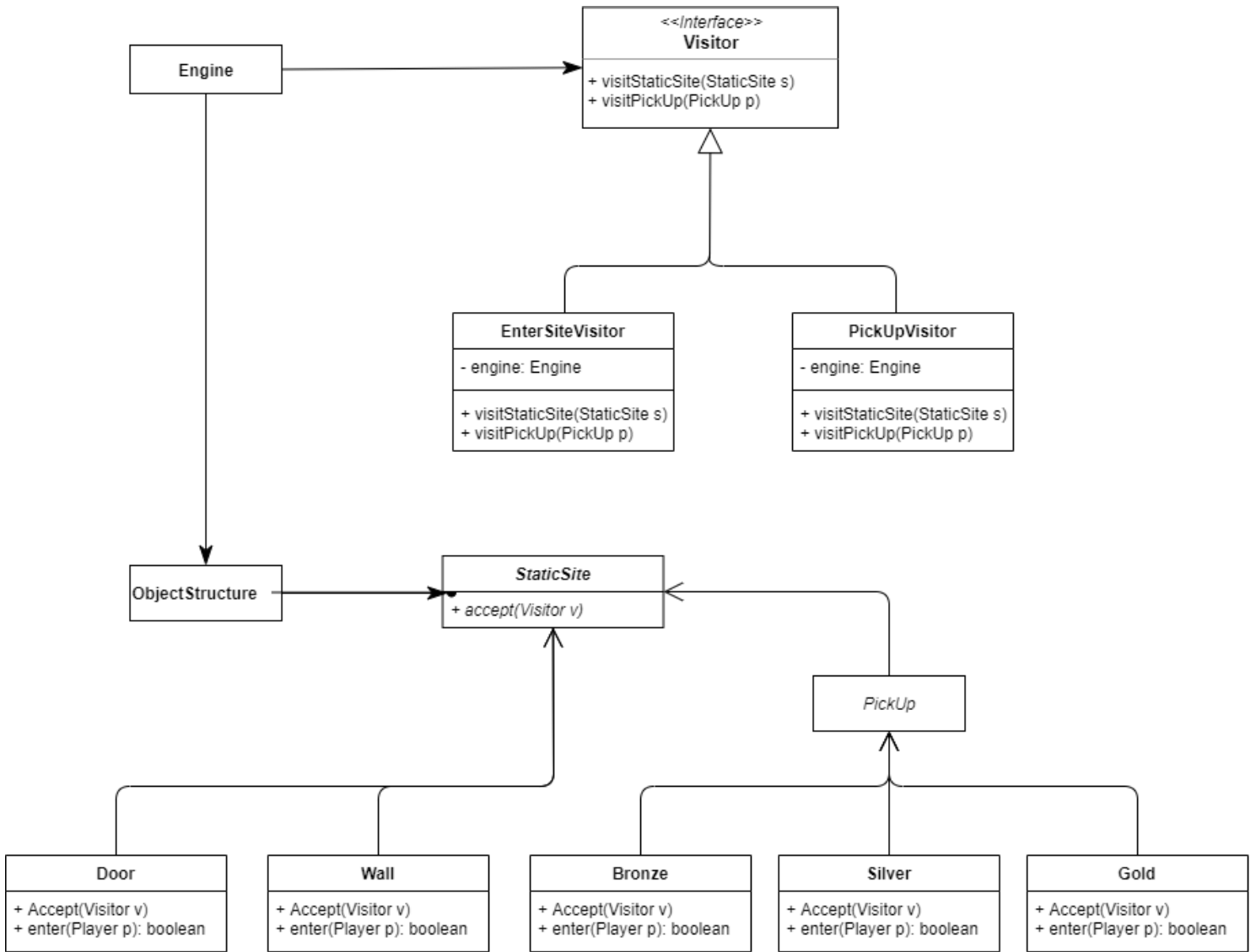


Figure 6.3.2: UML Class Diagram of Mobile Visitor

```

public class PickupVisitor implements Visitor {
    private Engine engine = Engine.getInstance();

    @Override
    public void visitStaticSite(StaticSite staticSite) {}
    @Override
    public void visitPickUp(PickUp pickUp) {
        double value = pickUp.getWeight() * pickUp.getValue();
        engine.incrementPlayerMoney(value);
    }
}

```

Code Snippet 6.6: Sample Code of Mobile Visitor

6.4 Strategy vs Mobile Strategy

In our Strategy pattern, the `GameActivity` contains a specific Strategy, either `JPEGStrategy` or `PNGStrategy`, for taking a screenshot. Our implementation of taking a screenshot on Android could not have been completed without the recommended solution from the author at Stack Overflow [25]. The strategies has to take in a activity as a parameter for it to be able to take a screenshot of the activity screen, save it and then view it. As we can see, from the method `setOnClickListener` in `GameActivity`, it sends its activity.

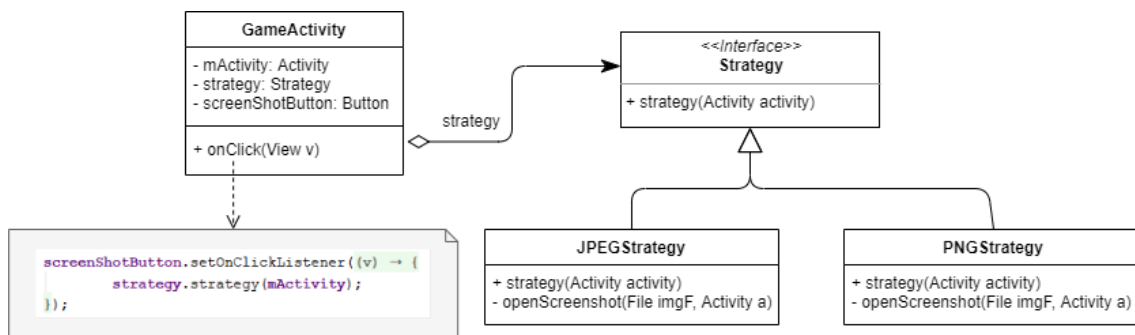


Figure 6.4.1: UML Class Diagram of Strategy

When calling a specific strategy, in our case, whether `JPEGStrategy` or `PNGStrategy`, one has to create an instance of the class. We felt this was an unnecessary operation as the strategy method does not need to be bound to a specific object. As a result, our proposed solution for our mobile strategy pattern was to make the strategy methods static. As we can see from the class diagram 6.4.2, we can call the strategy method without having to create an instance. For specifying the file format of the screenshot, an additional parameter `fileFormat` is created.

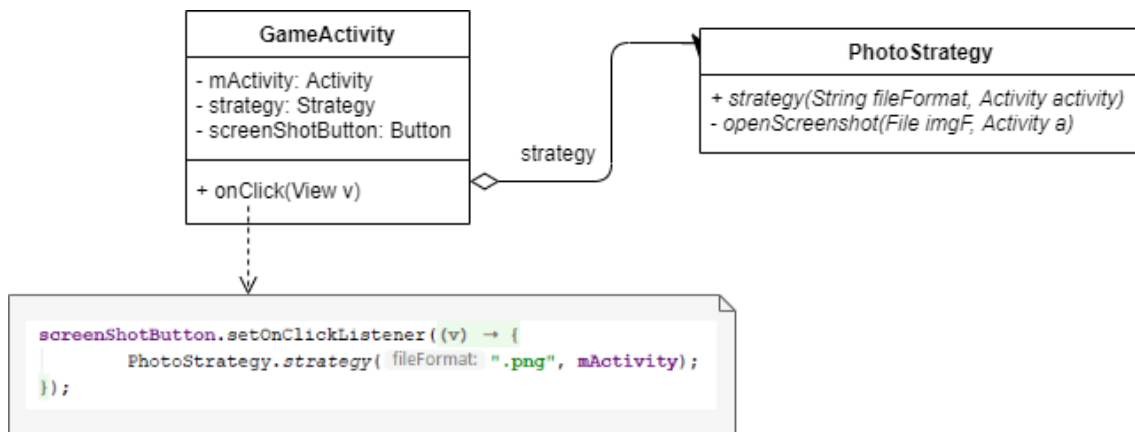


Figure 6.4.2: UML Class Diagram of Mobile Strategy

6.5 Singleton vs. Mobile Singleton

In this paper, we did not develop a mobile singleton pattern, because it was difficult to change it as there is not much that can be changed. The pattern can be combined with other patterns and used in other cases where it might solve some problems. We encountered one problem getting the context of the game activity for which our game

objects were displayed. The context of the activity was needed to create the image views of the game objects. We used Singleton to solve this problem by providing a global access point for the activity's context to be saved and attainable. The solution consisted of making the Engine a singleton, which makes sense as there should be only one Engine running the app. The global Engine instance was then accessed in the GameActivity class, and we passed the context of the activity to the Engine. Now we had a global access point to our necessary context, and the Engine's game objects could be easily created by passing down the context.

```
public class GameActivity extends AppCompatActivity {
    // Getting a global instance of Engine using Singleton
    private Engine engine = Engine.getInstance();

    @Override
    public void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_game);

        // Passing down the context of the activity to the Engine, so the
        // game can be started
        engine.startGame(getApplicationContext());
    }
}
```

Code Snippet 6.7: Sample Code of Providing Global Access Point for the Context of a Activity using Singleton

6.6 Summary

In summary, the mobile abstract factory was implemented with both combining the multiple images of each game object to one and converting the factory methods to static. The mobile command pattern was created by combining similar sub-command classes to one. The mobile visitor pattern was created by combining the visitor methods that visited the same element to one method by taking advantage of polymorphism. This was possible due to the elements having the same parent class. The mobile strategy was created by converting the strategy method to static and combining similar strategy classes to one.

Chapter 7

Results

This chapter shows the results from our measurement process of the apps implemented with regular and mobile design patterns towards our evaluation criteria. The tests were performed on a powerful ASUS gaming laptop with the following device specifications.

- **Processor** Intel Core i7-7700HQ CPU @ 2.80 GHz
- **Graphics Card** Geforce GTX 1070
- **Installed RAM** 16GB
- **Operating System** Windows 10 Home, 64-bit

7.1 QMOOD Results

In this section, the results of the design properties and its normalization from our QMOOD analysis are first shown. Based on these results and the computation formula, a quality attribute graph is created. The graph displays the computed values of the quality attributes of our regular and mobile design patterns.

7.1.1 Abstract Factory vs.. Mobile Abstract Factory

The class and package metrics of the regular abstract factory and mobile abstract factory were calculated. The following tables show the actual metric values and the normalized values of both patterns.

Metric	Abstract Factory	Mobile Abstract Factory
Design Size	3	3
Hierarchies	1	1
Abstraction	0.33	0
Encapsulation	1	1
Coupling	0.79	0.77
Cohesion	6	4.47
Composition	0	0
Inheritance	0.33	1
Polymorphism	12	0
Messaging	18	19
Complexity	10.5	12.33

Table 7.1.1: Actual Metric Values of Abstract Factory vs. Mobile Abstract Factory

Metric	Abstract Factory	Mobile Abstract Factory
Design Size	1	1
Hierarchies	1	1
Abstraction	1	0
Encapsulation	1	1
Coupling	1	0.97
Cohesion	1	0.78
Composition	0	0
Inheritance	1	3
Polymorphism	1	0
Messaging	1	1.06
Complexity	1	1.17

Table 7.1.2: Normalized Metric Values of Abstract Factory vs. Mobile Abstract Factory

After normalizing the metric values, the following quality attribute graph was created.

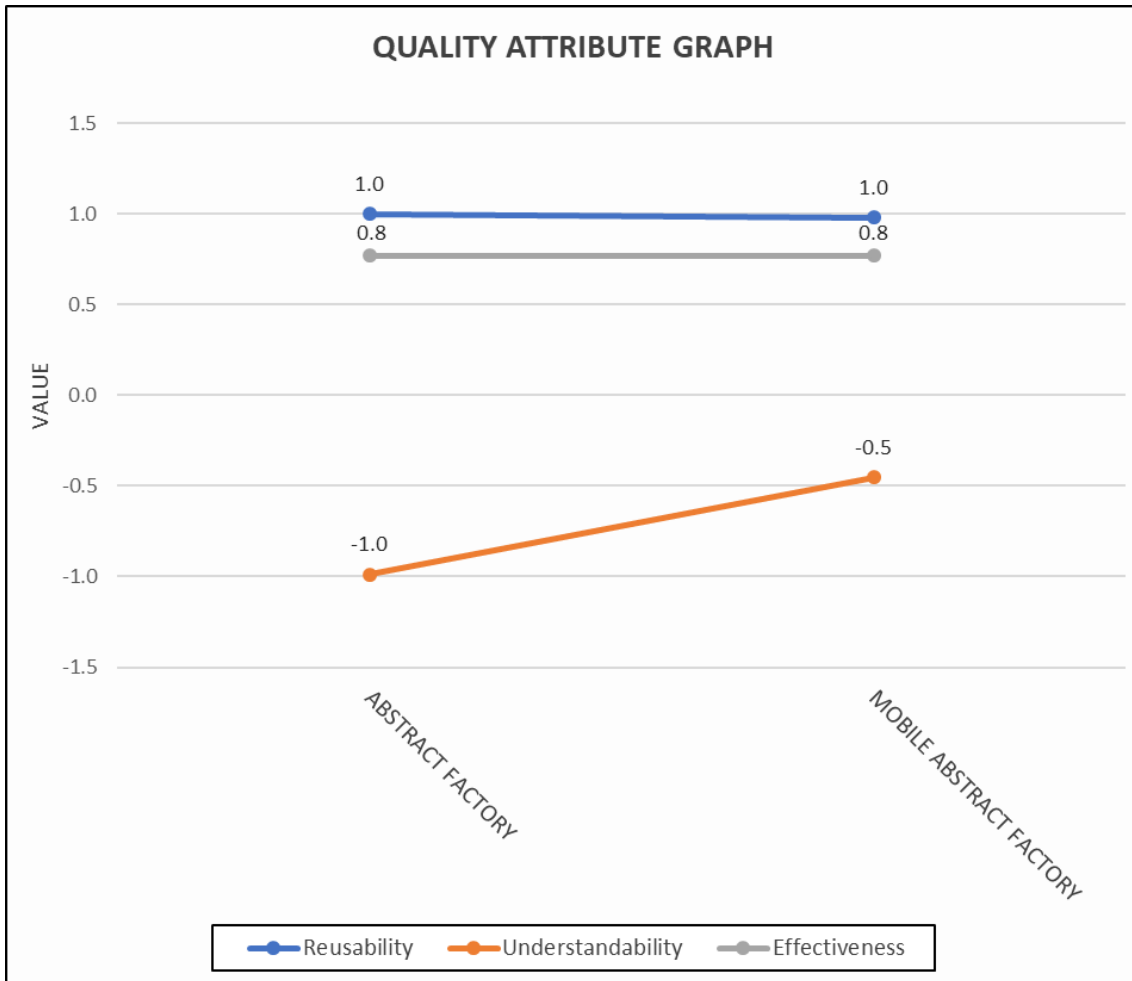


Figure 7.1.1: Quality Attribute Graph of Abstract Factory vs. Mobile Abstract Factory

From the graph 7.1.1, we can see that both the reusability and effectiveness of the mobile abstract factory has not changed compared to the regular version. However, the understandability of the abstract factory has increased from -1 to -0.50.

7.1.2 Command vs. Mobile Command

The following tables show the actual metric values and its normalized values of our Command and Mobile Command implementation.

Metric	Command	Mobile Command
Design Size	3	2
Hierarchies	1.67	1.50
Abstraction	0.33	0.50
Encapsulation	1	1
Coupling	0.64	0.57
Cohesion	1	1
Composition	5	3
Inheritance	0.67	0.75
Polymorphism	2	1
Messaging	6	4
Complexity	3.33	2.5

Table 7.1.3: Actual Metric Values of Command vs. Mobile Command

Metric	Command	Mobile Command
Design Size	1	0.67
Hierarchies	1	0.90
Abstraction	1	1.52
Encapsulation	1	1
Coupling	1	0.89
Cohesion	1	0.33
Composition	1	0.60
Inheritance	1	1.13
Polymorphism	1	0.50
Messaging	1	0.67
Complexity	1	0.75

Table 7.1.4: Normalized Metric Values of Command vs.. Mobile Command

Then the following quality attribute graph was created.

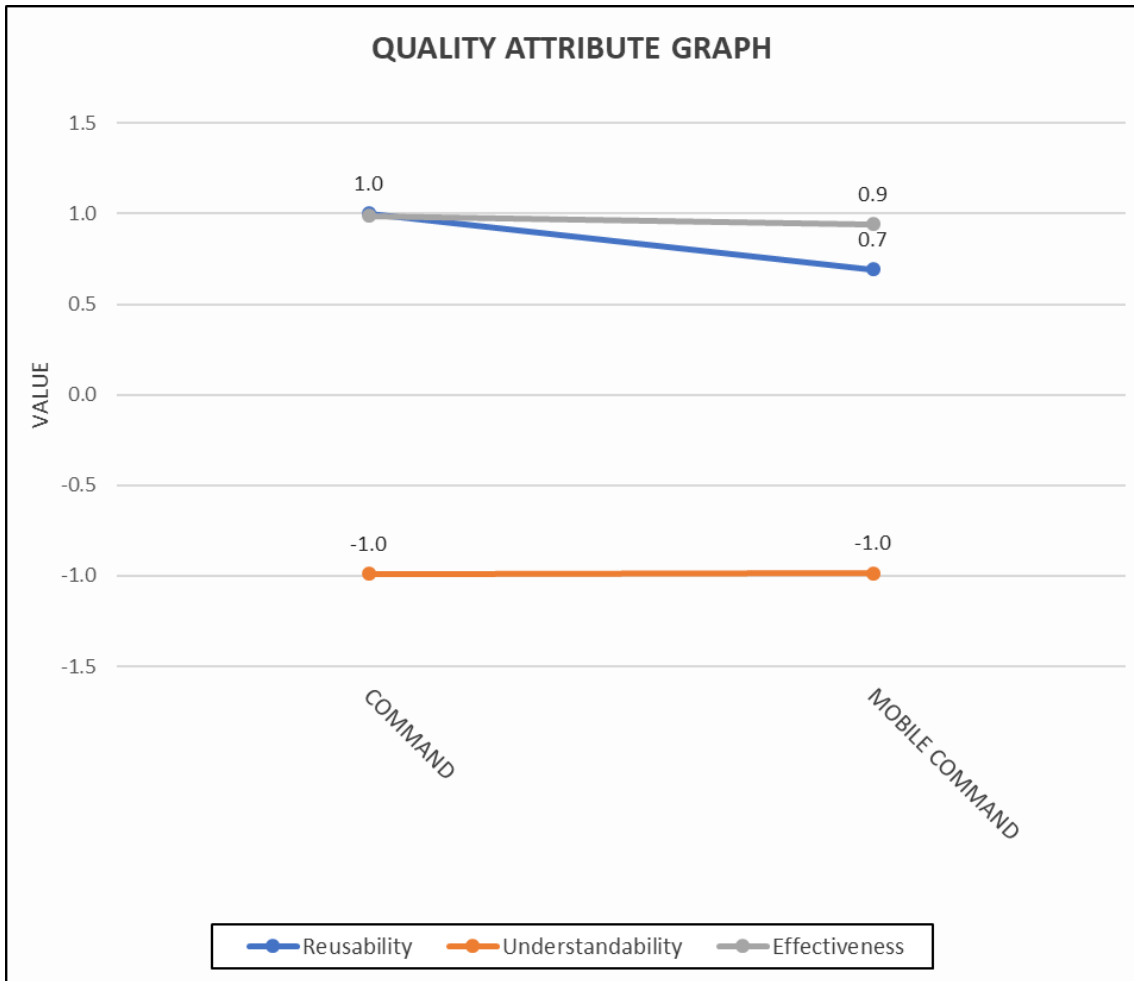


Figure 7.1.2: Quality Attribute Graph of Command vs.. Mobile Command

From the graph 7.1.2, we can see that both the reusability and effectiveness of the mobile command version decreased from 1 to 0.7 and 0.9. While the understandability of both patterns remained the same.

7.1.3 Visitor vs.. Mobile Visitor

The following tables show the metric values and the normalized values of our Visitor and Mobile Visitor implementations.

Metric	Visitor	Mobile Visitor
Design Size	4	3
Hierarchies	1	1
Abstraction	0.33	0.33
Encapsulation	1	1
Coupling	0.67	0.67
Cohesion	2	2
Composition	2	1
Inheritance	0.33	0.33
Polymorphism	10	4
Messaging	15	6
Complexity	5	2

Table 7.1.5: Actual Metric Values of Visitor vs.. Mobile Visitor

Metric	Visitor	Mobile Visitor
Design Size	1	0.75
Hierarchies	1	1
Abstraction	1	1
Encapsulation	1	1
Coupling	1	1
Cohesion	1	1
Composition	1	0.50
Inheritance	1	1
Polymorphism	1	0.40
Messaging	1	0.40
Complexity	1	0.40

Table 7.1.6: Normalized Metric Values of Visitor vs.. Mobile Visitor

Then the following quality attribute graph was created.

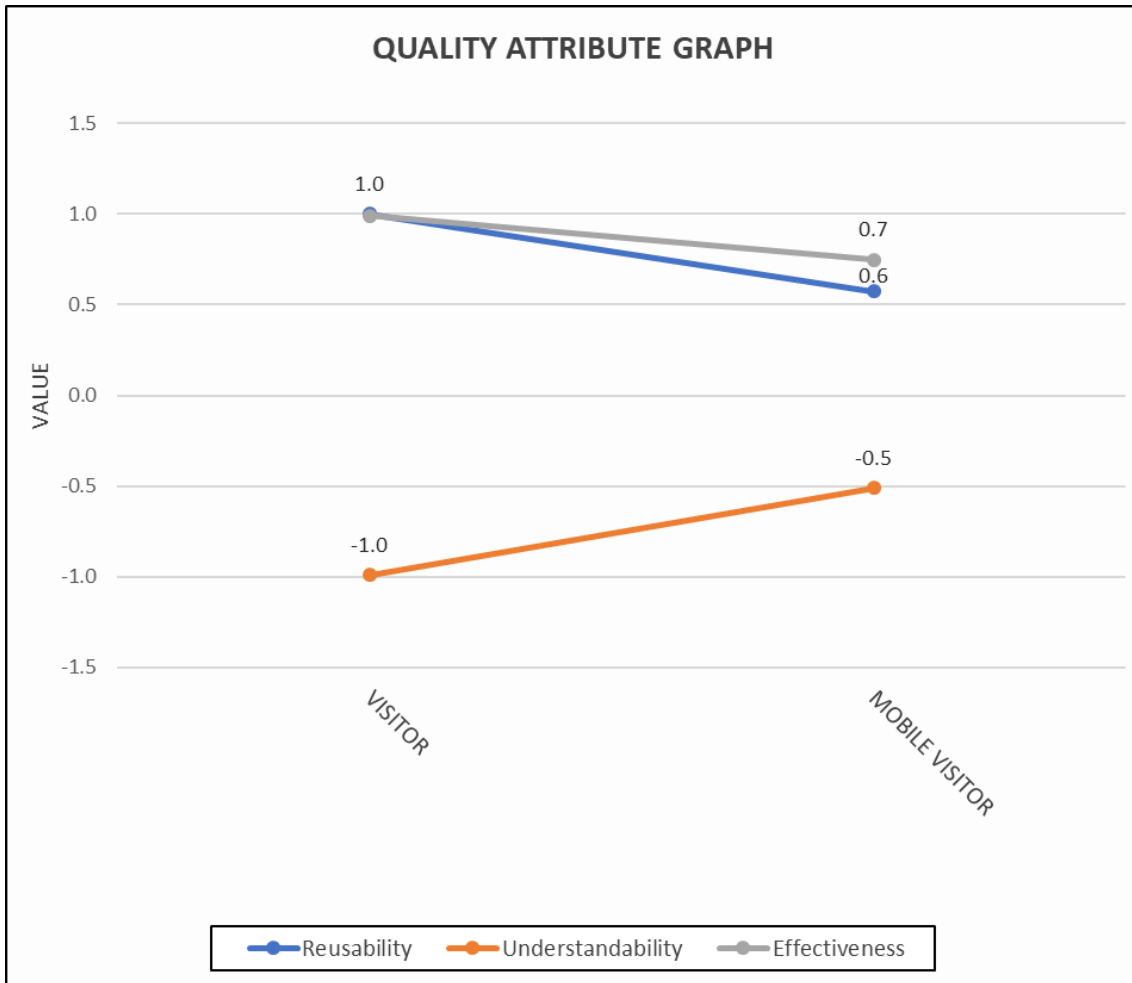


Figure 7.1.3: Quality Attribute Graph of Visitor vs. Mobile Visitor

From the graph 7.1.3, similar to the quality attribute graph of Abstract Factory, we can see that both reusability and effectiveness of the mobile visitor version have decreased from 1 to 0.6 and 0.5. While the understandability of the Visitor pattern has increased significantly from -1 to -0.2.

7.1.4 Strategy vs. Mobile Strategy

The following tables show the metric values and the normalized values of our Strategy and Mobile Strategy implementations.

Metric	Strategy	Mobile Strategy
Design Size	3	1
Hierarchies	1	1
Abstraction	0.33	0
Encapsulation	1	1
Coupling	0.4	0
Cohesion	1	1
Composition	0	0
Inheritance	0.6	1
Polymorphism	2	0
Messaging	5	2
Complexity	2	3

Table 7.1.7: Actual Metric Values of Strategy vs.. Mobile Strategy

Metric	Strategy	Mobile Strategy
Design Size	1	0.33
Hierarchies	1	1
Abstraction	1	1
Encapsulation	1	1
Coupling	1	0
Cohesion	1	1
Composition	0	0
Inheritance	1	1.67
Polymorphism	1	0
Messaging	1	0.40
Complexity	1	1.50

Table 7.1.8: Normalized Metric Values of Strategy vs. Mobile Strategy

Then its following quality attribute graph was created.

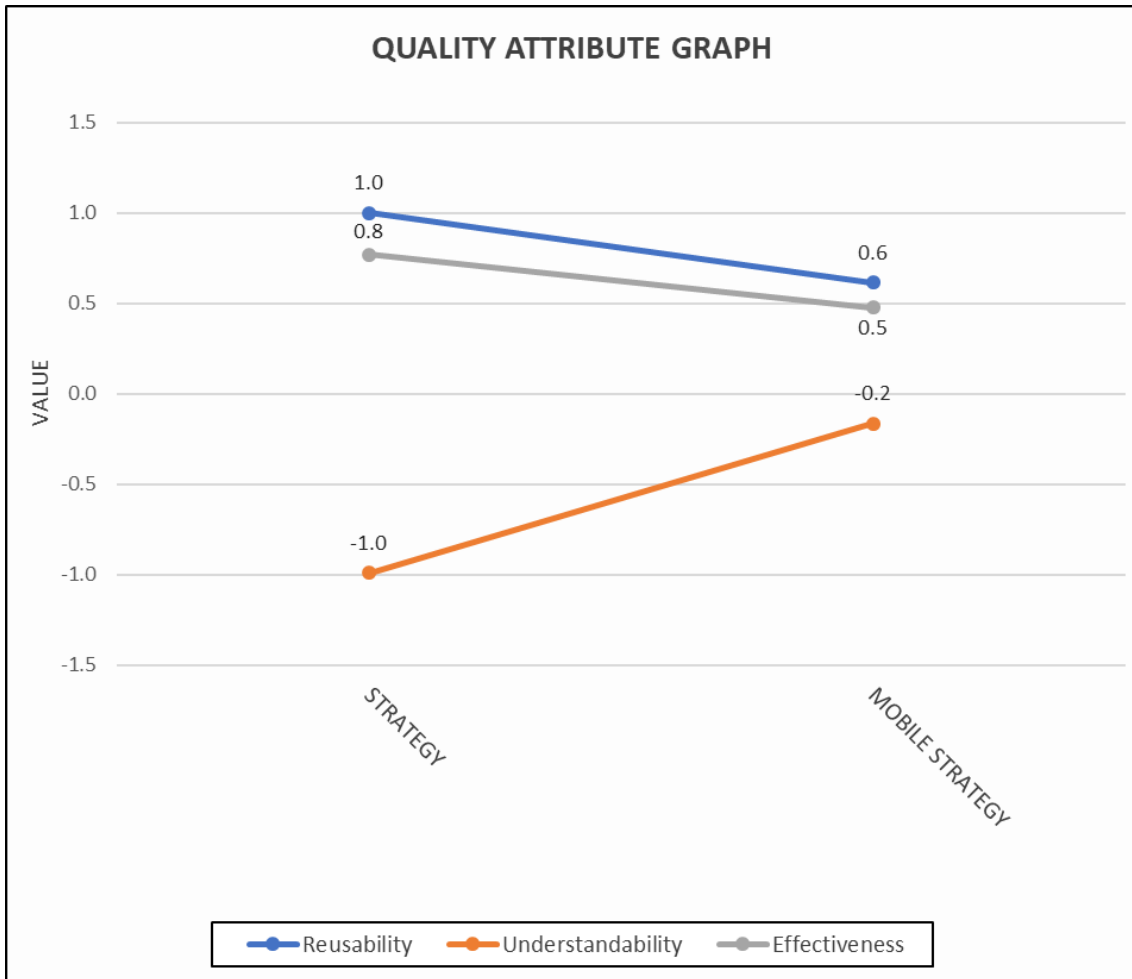


Figure 7.1.4: Quality Attribute Graph of Strategy vs.. Mobile Strategy

The quality attribute graph of Strategy 7.1.4, shows also that the reusability and effectiveness of the mobile strategy version have decreased to 0.6 and 0.5, while understandability has increased significantly from -1 to -0.2.

7.1.5 Project vs.. Mobile Project

When measuring the design properties of the project with regular design patterns and the mobile project with mobile design patterns, we summed all the values from the metric values of its design patterns that we have shown from above. Following tables these summed metric values and the normalized values of our project and mobile project.

Metric	Project Factory	Mobile Project
Design Size	37	35
Hierarchies	2.36	2.22
Abstraction	0.24	0.19
Encapsulation	0.70	0.67
Coupling	0.55	0.56
Cohesion	2.34	2.48
Composition	183	183
Inheritance	0.73	0.82
Polymorphism	46	28
Messaging	168	159
Complexity	8	8.1

Table 7.1.9: Actual Metric Values of Project vs.. Mobile Project

Metric	Project	Mobile Project
Design Size	1	0.95
Hierarchies	1	0.94
Abstraction	1	0.79
Encapsulation	1	0.96
Coupling	1	1.02
Cohesion	1	1.06
Composition	1	1
Inheritance	1	1.13
Polymorphism	1	0.61
Messaging	1	0.95
Complexity	1	1.01

Table 7.1.10: Normalized Metric Values of Project vs.. Mobile Project

After normalizing the metric values, the following quality attribute graph was created.

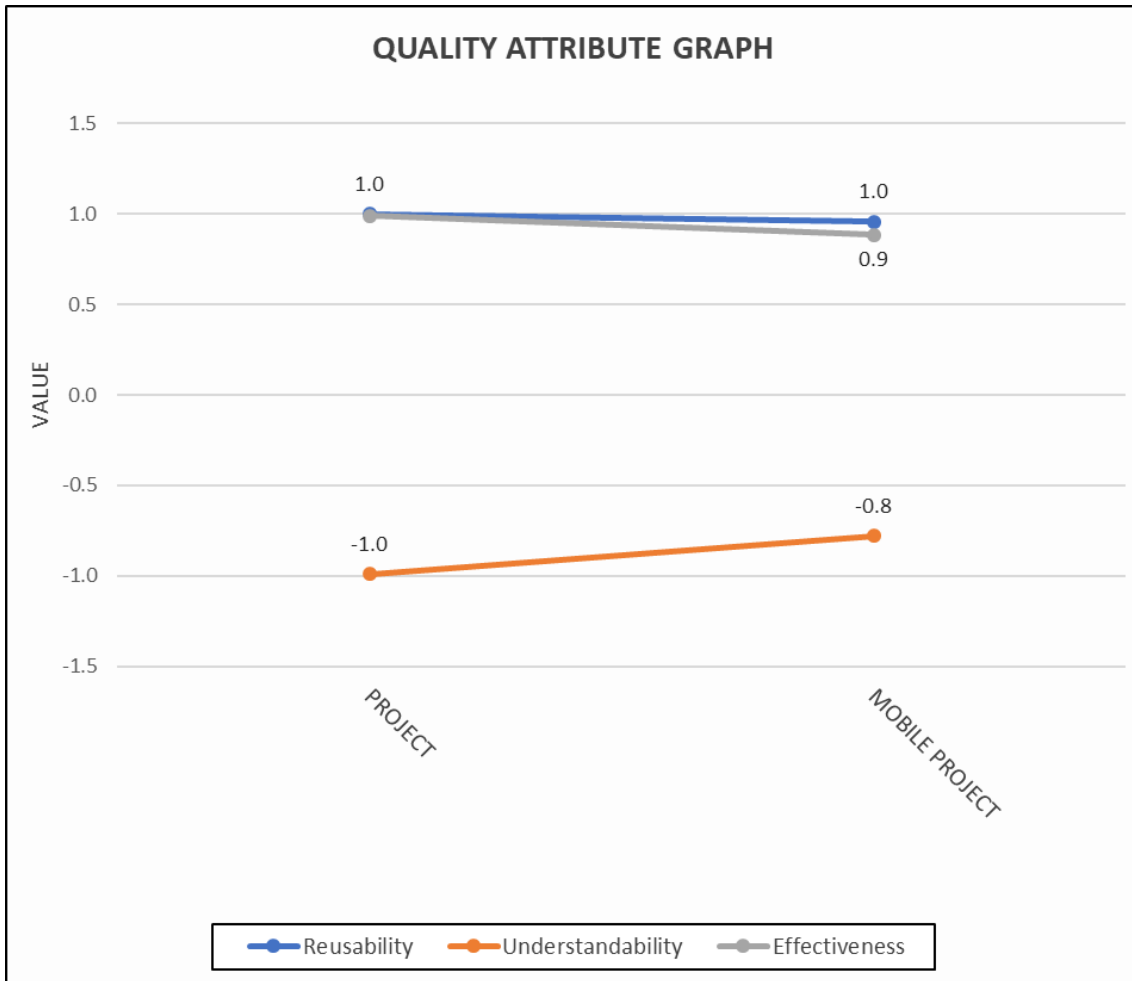


Figure 7.1.5: Quality Attribute Graph of Project vs. Mobile Project

The results from the graph 7.1.5 shows that the reusability of the mobile project has remained the same. The effectiveness has decreased slightly from 1 to 0.9, and the project’s understandability has increased from -1 to -0.8.

7.1.6 Summary

In summary, the QMOOD results show that most of the mobile design patterns improved in the quality attribute understandability, while the reusability and effectiveness of the pattern got reduced, especially the effectiveness. Notably, none of the mobile design patterns had improved in reusability and effectiveness, and none had decreased in understandability.

7.2 Android Profiler Results

When profiling each design pattern, we looked at the average CPU usage of the entire app and its memory usage, and then overall memory usage of the design pattern itself. At the end, the improvement of the mobile design patterns at each of these points is summarized compared to the usual design patterns.

7.2.1 Abstract Factory vs. Mobile Abstract Factory

The mobile app with mobile abstract factory had the same CPU usage as the regular app with abstract factory of 70%. But the mobile app used 100MB more than the regular app and the mobile design pattern itself used 40KB more.

Design Pattern	CPU Usage	Memory Footprint	Memory Usage of Design Pattern
Abstract Factory	70%	800MB	212KB
Mobile Abstract Factory	70%	900MB	252KB

Table 7.2.1: Profiling of Abstract Factory

7.2.2 Command vs. Mobile Command

For the mobile command, the app used 65% CPU usage compared to the regular command at 60%, and in addition 30MB more. The mobile command pattern itself was allocated 37KB, while the regular pattern was allocated 34KB.

Design Pattern	CPU Usage	Memory Footprint	Memory Usage of Design Pattern
Command	60%	90MB	34KB
Mobile Command	65%	120MB	37KB

Table 7.2.2: Profiling of Command

7.2.3 Visitor vs. Mobile Visitor

The mobile app with mobile visitor used 5% lesser CPU usage compared to the regular app with Visitor. The mobile app used only 22MB, while the regular app used 38MB. However, the memory usage of the mobile design pattern was around the same as the regular version despite the positive results from the previous two points.

Design Pattern	CPU Usage	Memory Footprint	Memory Usage of Design Pattern
Visitor	50%	38MB	49KB
Mobile Visitor	45%	22MB	51KB

Table 7.2.3: Profiling of Visitor

7.2.4 Strategy vs. Mobile Strategy

The regular app with Strategy had virtually the same CPU usage and total memory usage as the mobile version. But the mobile strategy pattern itself used 22KB, while the regular pattern used 31KB.

Design Pattern	CPU Usage	Memory Footprint	Memory Usage of Design Pattern
Strategy	27%	50MB	31KB
Mobile Strategy	26%	50MB	22KB

Table 7.2.4: Profiling of Strategy

7.2.5 Project vs. Mobile Project

When measuring the first field of the projects, we measured the average CPU usage based on its results from its design patterns above, while the last two fields are the sum of all the values from its design patterns.

Both projects used around, on average, the same CPU usage. However, the total memory usage of the mobile project and its design patterns was 1092MB and 362KB compared to the regular project of 1042MB and 326KB.

Design Pattern	CPU Usage	Memory Footprint	Memory Usage of Design Pattern
Project	52%	1042MB	326KB
Mobile Project	51%	1092MB	362KB

Table 7.2.5: Profiling of Project

7.2.6 Summary

In summary, the profiling shows that most of the mobile design patterns compared to the regular design patterns improved negatively, especially in the memory usage of the design pattern. Noticeably, the Total Memory Usage of the mobile command increased 33%. The mobile visitor had 10% better CPU Usage and 42% better Total Memory Usage. While the mobile strategy pattern had 32% better Memory Usage of Design Pattern. Overall, the mobile project had 2% better CPU Usage, 5% poorer memory usage and 11% poorer memory usage of the design patterns themselves.

 Positive Improvement  Negative Improvement

Summary	CPU Usage	Memory Footprint	Memory Usage of Design Pattern
Mobile Abstract Factory	0%	-12%	19%
Mobile Command	-8%	-33%	-9%
Mobile Visitor	10%	42%	-4%
Mobile Strategy	4%	0%	32%
Mobile Project	2%	-5%	-11%

Table 7.2.6: Summary of the Total Improvement

7.3 Execution Time Results

When measuring the execution time, we executed the implementation of the design patterns ten thousand times and then one hundred thousand times. The results displayed is the average time of execution time for 10 iterations. At the end, we summarize the results.

7.3.1 Abstract Factory vs. Mobile Abstract Factory

The mobile abstract factory had a better execution time of 1.22 seconds in ten thousand executions and 21.26 seconds in one hundred thousand executions, while the abstract factory had a execution time of 2.02 seconds and 31.14 seconds.

Design Pattern	Execution Time x 10k	Execution Time x 100k
Abstract Factory	2.02	31.14
Mobile Abstract Factory	1.22	21.26

Table 7.3.1: Execution Time of Abstract Factory

7.3.2 Command vs. Mobile Command

Mobile Command resulted in a execution time of 0.23 seconds and 2.06 seconds. Meanwhile, the regular Command had a execution time of 0.32 seconds and 3.00 seconds.

Design Pattern	Execution Time x 10k	Execution Time x 100k
Command	0.32	3.00
Mobile Command	0.23	2.06

Table 7.3.2: Execution Time of Command

7.3.3 Visitor vs. Mobile Visitor

When executing itself ten thousand times, both Visitor patterns had approximately the same execution time of 0.70 seconds. However, when executing one hundred times the regular visitor pattern had a better execution time of 6.57 seconds, while the mobile version had 6.13 seconds.

Design Pattern	Execution Time x 10k	Execution Time x 100k
Visitor	0.70	6.57
Mobile Visitor	0.71	6.13

Table 7.3.3: Execution Time of Visitor

7.3.4 Strategy vs. Mobile Strategy

Mobile Strategy had just about a better execution time of 1.76 seconds and 17.23 seconds, compared to the regular Strategy that had 1.79 seconds and 17.66 seconds.

Design Pattern	Execution Time x 10k	Execution Time x 100k
Strategy	1.79	17.66
Mobile Strategy	1.76	17.23

Table 7.3.4: Execution Time of Strategy

7.3.5 Project vs. Mobile Project


The execution time of the projects are measured by adding up the execution time of its design patterns. As we can see in the following table, the mobile project, containing the mobile design patterns, had a better execution time in both fields of 3.86 seconds and 51.05 seconds, compared to the regular project's 6.04 seconds and 64.96 seconds.

Design Pattern	Execution Time x 10k	Execution Time x 100k
Project	6.04	64.96
Mobile Project	3.86	51.05

Table 7.3.5: Execution Time of Project

7.3.6 Summary

In summary, the results shows that the mobile design patterns generally had a better execution time compared to the regular design patterns. Noticeably, we can see in table 7.3.6 that the mobile abstract factory had 36% better execution time then the regular abstract factory. Similarly, the mobile command had 29% better execution time then its regular version. While both mobile visitor and strategy had small improvements of 3% and 2%. Overall, the mobile project improved 17% in its execution time of its implemented design patterns.

 Positive Improvement

Summary	Total Improvement
Mobile Abstract Factory	36%
Mobile Command	29%
Mobile Visitor	3%
Mobile Strategy	2%
Mobile Project	17%

Table 7.3.6: Summary of the Total Improvement

7.4 Weakness and Mistakes of our Measurement Process

A weakness in our QMOOD analysis was that we compared only two versions to the regular and mobile design patterns. We analyzed at the end when we finished applying all the changes we made to develop the mobile design patterns and the project. This made it difficult to pinpoint which changes had the least and most impact on the quality attributes.

Better QMOOD analysis for the future would have been to have several more versions of our mobile design patterns whenever an update is made. Furthermore, we should have changed the model's influence and weight following the techniques and strategies we used to develop the mobile design patterns. For instance, according to [17], the design property Design Size that described the number of classes should have had a negative influence on the quality attribute Effectiveness. These changes could have led to results that better indicate how much of the techniques and strategies we have used.

Another weakness of our study was that only one component, namely Activity, of the Android architecture was used in the development of our app. Other app components from Android, such as Broadcast receivers, Services, and Content providers, were not included in our app or code. These components can potentially impose new requirements for design patterns. Due to adequate time, the activity component with the design patterns was also not thoroughly tested for other problems such as memory leaks or other potential problems like using multiple activities in the Observer pattern.

One critical mistake was that we started the measurement process after we had implemented all the changes and updates to the mobile design patterns, that is when we had finished the development phase. This made it difficult to pinpoint which changes had the least and most impact in the measurement process.

Finally, the thesis was about design patterns for mobile devices. However, we tested only the design patterns for one mobile device or emulator. Testing the design patterns with several different mobile devices would have reflected the different characteristics of a mobile device better and lead to a better validation of the results.

7.5 Summary

The QMOOD results show none of the mobile design patterns improving in reusability and effectiveness, but most of the patterns improved in understandability.

Results of the Android profiling shows that most of the mobile design patterns improved positively in CPU Usage, but negatively in the memory footprint and memory usage of the design pattern themselves. Notably, the mobile command's memory footprint increased 33%, while the mobile visitor had 10% better CPU Usage and 42% better Total Memory Usage. Overall, the mobile project had 2% better CPU Usage, 5% poorer memory footprint, and 11% poorer memory usage of the design patterns themselves.

In general, the execution time of the mobile design patterns had better execution time than the regular design patterns. Most impressively, the mobile abstract factory had an improvement of 36%, and the mobile command had 29% better execution time than its regular version. Overall, the execution time of the mobile project improved 17% compared to the regular project.

Chapter 8

Discussion

In this chapter, we discuss the results of our regular and mobile design patterns from our measurement process, including QMOOD, android profiling, and execution times. We discuss the improvements in our mobile design patterns and what we could have done differently. In the end, we discuss the requirements we believe were imposed on the design patterns by Android and its mobile devices.

8.1 QMOOD

From our QMOOD analysis, none of our mobile design patterns improved in efficiency or reusability compared to its regular design pattern. However, an expected improvement between the patterns was in understandability due to the changes made for the mobile design patterns reduced the overall size of the pattern: small classes combined to one, similar methods combined to one method, and converting the instance methods to static resulted in removing the inheritance mechanism of the classes, and thus the overridden methods. For QMOOD, reducing the overall size meant that the metric values for the design properties also got reduced. This will, as expected, by seeing the computation formula for calculating the quality attributes, give us lower efficiency and reusability. For those two quality attributes, most of the design properties influence the attribute positive, which means that the more methods, classes, and use of inheritance, the better the results. The positive influence is contradictory to the methods we used from [17] for developing mobile design patterns such as merging small classes into a bigger one for saving static space and the avoidance of using inheritance when possible. While for the quality attribute understandability, the methods implemented from the [17] had a positive effect. As we can see, our QMOOD's influence and weight did not follow the techniques and strategies we used to develop the mobile design patterns. Not regarding it, explains why none of the mobile design patterns could improve in reusability and efficiency, but only improve in understandability.

8.2 Improved Execution Times

Mobile Abstract Factory had a significantly better execution time than its predecessor, with an improvement of 36%. The mobile abstract factory was, on average, 10 seconds faster than the Abstract Factory. The improvements can be understood by first explaining the difference in memory allocation. Allocating objects in the heap is known to be costly, hence why one should avoid it and rather allocate memory to the stack when possible.

We accomplished this in our design patterns, where we converted the instance methods to static methods. By doing so, we no longer had to create an instance of the class when calling the factory method. By avoiding creating an instance, we avoid unnecessary allocation of memory to the heap. Avoiding unnecessary allocations can improve the execution time and save memory, especially in the worst-case scenario where we have to create an instance for each time we call the instance method. For further description, please see [table 8.2.1, p. 85]. However, this drastic improvement does not apply when we do not create a new object for each time we call the instance method. By the test from 8.2.1, we concluded that there is no noticeable difference in the execution time of a static method and an instance method when called from a single class reference or object. Nevertheless, the advantage of the static method is that we can avoid allocating memory in the heap (further explained in the discussion of Android profiling), and in the worst-case scenario, we have to create an instance for each time we call the instance method. One should, if possible and convenient, use a static method.

Despite the small changes in combining two sub-command classes into one command, the mobile Command pattern also had a better execution time than the regular Command pattern with an improvement of 29%. The only change we made was combining the two commands into one command EnterCommand. According to [17], combining small classes into one class can reduce memory from the heap as the overhead of the small classes is removed and no longer needed. However, we combined only two classes, so we had to test if this was the case for the improvement of 29%. It turned out it was not, because we implemented and tested the execution time of the regular Command pattern in the mobile project, and its average execution time was no different to the mobile design pattern. The real difference between the two patterns was the game objects it was working with. In the mobile project, the images of the game objects such as Player, Wall, and Door, were combined. The image of the sprites was selected by extracting the desired part from the combined image. These desired parts were preallocated in memory in the ImageManager class, making the instruction of setting a desired image to the sprite efficient, as the program just had to access the preallocated image. In conclusion, it was the more efficient game objects, combining small images into one, which improved the mobile Command pattern and not the change we made directly to the Command pattern, which was combining the small command classes. However, combining small command classes into one should not be excluded as a possible improvement, as we tested combining only two classes into one and not numerous classes.

The visitor patterns had approximately the same execution time when they executed ten thousand times, and a difference of 0.44 seconds when executed one hundred times. The change we made directly to the mobile visitor pattern was combining multiple visitor methods that visited elements of the same parent class into one visitor method, and it seemed like the method itself had no real effect. In theory, more methods a class has, the more static memory is allocated in the heap as information about each method to a class is saved. However, in practice, the memory saved by combining methods into one was not noticeable. Noteworthy, it also seemed like the mobile game objects which combined their images into one had no real effect of improving the visitor pattern. This is likely due to the visitor methods not allocating new game objects or changing existing game objects such that a new image of the sprite was selected.

The mobile and regular strategy pattern also had around the execution time. The change made to the mobile Strategy pattern of making the strategy algorithms of PNGStrategy and JPEGStrategy to static methods had no significant effect. The difference in execution time from static and instance strategy methods was hard to see

because the strategy patterns were tested with ten iterations and one hundred iterations. This low amount of iterations was due to the strategy algorithm's high cost, which took a screenshot. As a result, we did another test where the strategy methods only did some basic arithmetic operations, so we could execute the methods numerous times to see if there is a difference. The results were clear; a static method invoked from a class reference had the same execution time as an instance method invoked from one object reference. Both cases had an execution time of around 0.21 seconds. However, in the worst-case scenario, when the instance method got invoked for each time an object is created, it had an execution time of 1.01 seconds. The inefficient result is due to the constant memory allocation of creating an object for every method call. In conclusion, a mobile strategy pattern with static strategy methods can be very beneficial if one has to create a new strategy object for every strategy call.

Method Call	Execution Time
Invoke Static Method from Class Reference	0.21s
Invoke Instance Method from one Object Reference	0.21s
Invoke Instance Method for each Object Created	1.01s

Table 8.2.1: Execution Time of Static Method vs Instance Method B.1

8.3 Android Profiling

The profiling of our design patterns showed an interesting mix of results. Both mobile abstract factory and command, which had a better execution time, used more resources of CPU, memory footprint, and the patterns itself also used more memory. The mobile Visitor pattern, which improved slightly in execution time, used less CPU, and overall memory. The strategy patterns had the same CPU usage and total memory usage, but the mobile strategy pattern itself used 9 KB less.

In conclusion, the mobile project used more memory than the regular project, most likely, due to the extra class and code, we implemented from the technique of combining multiple images into one. Our implementation consisted of an ImageManager class that took the use of the singleton pattern. The class took the use of the Bitmap and Drawable classes to retrieve all the desired parts from our sprite sheets. This change resulted in faster execution time for the mobile abstract factory and Command pattern, but it cost an extra in both memory and CPU usage.

We used the same test to analyze the memory and CPU usage as we discussed the execution time of static methods versus instance methods with a simple test. As we can see from table 8.3.1, the results were clear. The worst-case scenario where calling the instance method for each object created used almost double amount of CPU then the best scenario where we call the instance method from just one object. The worst-case scenario added 22 MB to the app while the best scenario added 0,2 MB. When calling the instance method from only one object, there was no real difference in CPU usage and total memory usage. However, when looking at the number of allocations and shallow size of Employee class, the best scenario had one allocation of 16KB, whereas the static method had zero allocation and thus zero in shallow size. Even though the difference is not noticeable when using the app, it is noteworthy that using a static method; we avoid allocating at least one object. In conclusion, one should use static methods, if possible, to avoid allocating an unnecessary amount of memory and using unnecessary CPU.

Method Call	CPU Usage	Total Memory Usage
Invoke Static Method from Class Reference	25%	10MB
Invoke Instance Method from one Object Reference	25%	10MB
Invoke Instance Method for each Object Created	40%	22MB

Table 8.3.1: CPU and Memory Usage of Static Method vs Instance Method B.1

8.4 Did Mobile Devices Impose new Requirements for Design Patterns?

During the measurement process of our game app, the memory for our mobile device was noticeably limited. When allocating a high number of objects, it did not take long before there was little to no memory in the heap. When there was no free memory of the heap, the app froze and had to wait for the garbage collector to finish up, freeing up enough memory for the app to begin starting again. With limited memory space, memory saving techniques, and proper management of memory are necessary for developing more quality and faster mobile app. This was proved with the mobile Abstract Factory as we converted the factory methods to static and developed more efficient products of the factory by combining their images into one image, and preallocated the desired parts of the image. With the right techniques and strategies applied, we proved that a regular design pattern could be translated into a more mobile-friendly pattern. In conclusion, performance and managing memory in a more efficient way is a clear requirement for design patterns.

We came across some problems with the activity component of Android architecture when implementing the design patterns for our maze app using Android. For instance, the regular implementation of the Command pattern and Abstract Factory were not sufficient for running the maze game. The Abstract Factory had the task of creating game objects, and the Command pattern could receive a game object as a parameter from a command request and execute some code. The problem for both patterns was that the context of the Activity was required for performing their task. For instance, the game objects needed the context to define their ImageView when created. As a result, the Abstract Factory could not create new products without getting the context of the Activity for which the game objects were going to be displayed. Activity is one of the components of the Android architecture. The architecture and components differ from different mobile platforms like Android and Apple's iOS. In conclusion, it seems there is a requirement for design patterns to consider if it is necessary to incorporate the components from the mobile platform's architecture and adequately handle them.

8.5 Summary

We identified two requirements imposed by Android and their mobile devices during the development of our mobile maze game in Android with design patterns. Requirement one was for the design patterns to consider incorporating components of the mobile platform architecture, in our case Android. Requirement two was for the design patterns to manage limited memory and CPU in a more optimized way for leading to a more efficient and faster app.

Requirement one was first imposed on the Abstract Factory as the game objects that were going to display on the UI of an activity, needed its context when created. By creating

an extra parameter context in the factory methods and passing down the context from the activity to the Engine where we use the Abstract Factory, the requirement solved.

Requirement two imposed due to the limited computing power and resources of the mobile device. As the regular design patterns were originally not created for such an environment, we implemented and tested different memory and power optimization techniques and strategies in the development of the mobile design patterns. Improving all three execution time, memory, and CPU usage of the mobile patterns turned out to be challenging. The results show that noticeably improved execution time uses more memory and the same amount or more CPU. The following table shows the techniques and strategies applied to the mobile design pattern, their improvements compared to the regular design patterns, and which techniques and strategies had a successful and unsuccessful impact. Noticeably, the methods that had a successful impact was combining multiple images to one and using static methods.

 Positive Improvement  Negative Improvement

	Successful Methods	Unsuccessful Methods	Exec. Time	CPU Usage	Memory Footprint
Abstract Factory	combining multiple images to one, using static methods		36%	0%	-12%
Command	combining multiple images to one	combining small classes to one	29%	-8%	-33%
Visitor	combining similar methods to one		3%	10%	42%
Strategy	using static methods		2%	4%	0%

Figure 8.5.1: Effect of Techniques and Strategies on Mobile Design Patterns

The drawbacks of the techniques and strategies we used to create mobile design patterns were that it could remove the inheritance of the classes by converting the methods to static and reduce the number of classes and methods. The drawback emerged in our quality attribute graphs from our QMOOD results, showing that most mobile design patterns had reduced their reusability attribute. Low reusability can lead to a longer development process, and this is not optimal in short to time market of mobile apps. However, the understandability of the mobile design patterns improved, which, according to the QMOOD definition, the design patterns are easier to comprehend and less complex. Better understandability can be one of the advantages of techniques and strategies for optimizing power and memory consumption.

Finally, one key lesson we learned throughout our thesis and believed it is important to understand for developing more efficient design patterns is memory allocation. For example, through our testing and research of memory allocation and using static methods [6][7], we learned it is more CPU and memory efficient than instance methods. A critical difference between the two methods is that a static method requires only one CPU instruction when called from the stack, while an instance method must first allocate an object in the heap before it can be called. The difference is illustrated in the following figure.

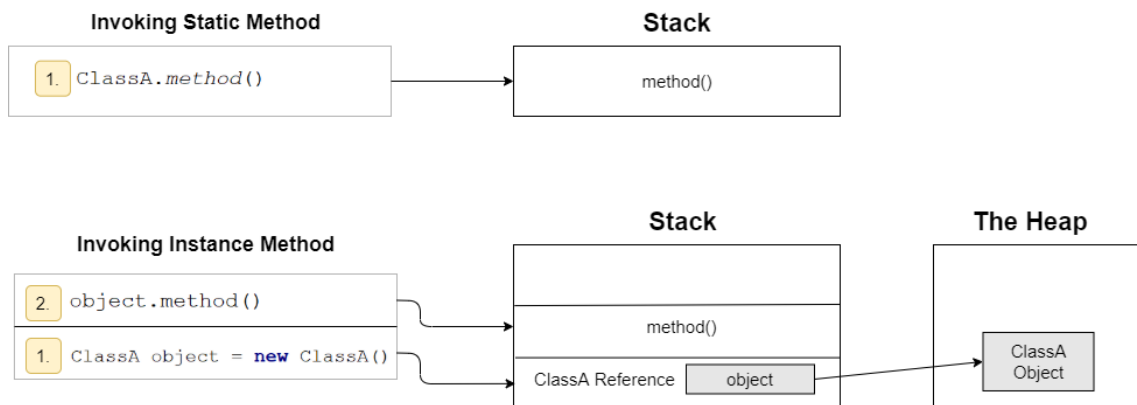


Figure 8.5.2: Invoking Static Method Versus Instance Method

Chapter 9

Conclusion

Developing a mobile game in Android imposed two requirements for the software design patterns. Requirement one was for the design patterns to consider incorporating the components of the mobile platform architecture. In our case, the context of the Activity component of the Android architecture was, e.g., needed in the Abstract Factory pattern for creating its products. The second requirement was for the design patterns to be more optimized or efficient by managing the memory more suitable to the limiting processing power and memory of the mobile device, as the design patterns were initially not designed for such an environment.

In conclusion, the results of the comparative study showed that mobile design patterns improved the performance efficiency of the mobile app and the understandability of the patterns at the expense of their reusability. Optimization techniques and strategies for energy and memory consumption, such as converting the instance methods to static, could increase the execution time of the design patterns, but it could also remove its inheritance mechanism. According to our QMOOD results, the reusability of the pattern would decrease and instead increase the understandability.

Notable, the combination of converting instance methods to static and combining multiple images to one, improved the execution time of Abstract Factory 36%, while just combining multiple images to one, improved the execution time of Command 29%. One important lesson we learned and believe is key for understanding in developing more efficient design patterns is memory allocation, to avoid allocating unnecessary memory, especially to the heap, and managing memory in a more optimized way.

Appendices

Appendix A

Project Implementation

In this appendix, we present the implementation of the regular and mobile project and its design patterns. Note that this is more of a refined version of our projects, in an attempt to make it more readable and easier to understand.

A.1 Project with Regular Design Patterns

A.1.1 Package; abstractFactory

```
public class BombedMazeFactory implements MazeFactory {
    @Override
    public Door makeDoor(Room r1, Room r2, Direction direction, Context
        context) {
        return new BombedDoor(r1, r2, direction, context);
    }
    @Override
    public Maze makeMaze() {
        return new Maze();
    }
    @Override
    public Room makeRoom(int roomNumber) {
        return new Room(roomNumber);
    }
    @Override
    public Wall makeWall(Direction direction, Context context) {
        return new BombedWall(direction, context);
    }
    @Override
    public Pickup makeRandomPickUp(Context c) {
        Random random = new Random();
        int randWeight = random.nextInt(100);
        switch (random.nextInt(3)){
            case 0: return new Bronze(c, randWeight);
            case 1: return new Silver(c, randWeight);
            case 2: return new Gold(c, randWeight);
            default: return new Bronze(c, randWeight);
        }
    }
}
public interface MazeFactory {
    Maze makeMaze();
    Wall makeWall(Direction d, Context c);
    Room makeRoom(int roomNumber);
}
```

```

    Door makeDoor(Room r1, Room r2, Direction d, Context c);
    Pickup makeRandomPickUp(Context c);
}
public class StandardMazeFactory implements MazeFactory {
    @Override
    public Door makeDoor(Room r1, Room r2, Direction direction, Context
        context) {
        return new StandardDoor(r1, r2, direction, context);
    }
    @Override
    public Maze makeMaze() {
        return new Maze();
    }
    @Override
    public Room makeRoom(int roomNumber) {
        return new Room(roomNumber);
    }
    @Override
    public Wall makeWall(Direction direction, Context context) {
        return new StandardWall(direction, context);
    }
    @Override
    public Pickup makeRandomPickUp(Context c) {
        Random random = new Random();
        int randWeight = random.nextInt(100);
        switch (random.nextInt(3)){
            case 0: return new Bronze(c, randWeight);
            case 1: return new Silver(c, randWeight);
            case 2: return new Gold(c, randWeight);
            default: return new Bronze(c, randWeight);
        }
    }
}
}

```

A.1.2 Package; activity

```

public class GameActivity extends AppCompatActivity {

    private Engine engine = Engine.getInstance();

    private Strategy strategy = new PNGStrategy();
    private Activity mActivity;

    private Button screenShotButton;

    public GameActivity() {}

    @Override
    public void onCreate(final Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_game);

        ActivityCompat.requestPermissions(this,
            new String[]{Manifest.permission.WRITE_EXTERNAL_STORAGE},
            1);

        mActivity = this;
        screenShotButton = (Button) findViewById(R.id.screenShotButton);
    }
}

```



```

engine.moneyText = (TextView) findViewById(R.id.moneyText);
engine.enterDoorButton = (Button) findViewById(R.id.actionButton);
engine.gridLayout = (FrameLayout) findViewById(R.id.gridLayout);
engine.startGame(getApplicationContext());
engine.enterDoorButton.setOnClickListener(new View.OnClickListener()
{
    @Override
    public void onClick(View v) {
        engine.executeEnter();
    }
});

screenShotButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        strategy.strategy(mActivity);
    }
});

}

@Override
public void onWindowFocusChanged(boolean hasFocus) {
    super.onWindowFocusChanged(hasFocus);
    if(hasFocus){
        engine.positionEachRoomFromMaze(engine.maze);
    }
}

@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    switch (keyCode){
        case KeyEvent.KEYCODE_D:{
            if(engine.currentRoom.getStaticSite(East) != null){
                if(! (engine.currentRoom.getStaticSite(East).intersects(
                    engine.player.getSprite()))){
                    engine.player.moveRight();
                    engine.intersectsFlag = false;
                }
            } else{
                engine.player.moveRight();
            }
            return true;
        }
        case KeyEvent.KEYCODE_A:{
            if(engine.currentRoom.getStaticSite(West) != null){
                if(! (engine.currentRoom.getStaticSite(West).intersects(
                    engine.player.getSprite()))){
                    engine.player.moveLeft();
                    engine.intersectsFlag = false;
                }
            } else{
                engine.player.moveLeft();
            }
            return true;
        }
        case KeyEvent.KEYCODE_S:{
            if(engine.currentRoom.getStaticSite(South) != null){
                if(! (engine.currentRoom.getStaticSite(South).intersects(

```



```

    }
    @Override
    public void visitWall(Wall wall) {
        Engine.getInstance().setEnter(new EnterWallCommand(wall));
    }
    @Override
    public void visitBronze(Bronze bronze) {
        PickupVisitor pickUpVisitor = new PickupVisitor();
        bronze.accept(pickUpVisitor);
        Engine.getInstance().gridLayout.removeView(bronze.getSprite());
    }
    @Override
    public void visitGold(Gold gold) {
        PickupVisitor pickUpVisitor = new PickupVisitor();
        gold.accept(pickUpVisitor);
        Engine.getInstance().gridLayout.removeView(gold.getSprite());
    }
    @Override
    public void visitSilver(Silver silver) {
        PickupVisitor pickUpVisitor = new PickupVisitor();
        silver.accept(pickUpVisitor);
        Engine.getInstance().gridLayout.removeView(silver.getSprite());
    }
}

```

A.1.5 Package; strategy

```

public interface Strategy {
    void strategy(Activity activity);
}
public class PNGStrategy implements Strategy {
    @Override
    public void strategy(Activity activity) {
        Date now = new Date();
        String fileName = "screenshot"
            + android.text.format.DateFormat.format("yyyy-MM-dd_hh:mm:ss", now);
        String fileFormat = ".png";

        try {
            String path = Environment.getExternalStorageDirectory().toString()
                + "/" + fileName + fileFormat;

            // Screen capture
            View view = activity.getWindow().getDecorView().getRootView();
            view.setDrawingCacheEnabled(true);
            Bitmap bitmap = Bitmap.createBitmap(view.getDrawingCache());
            view.setDrawingCacheEnabled(false);

            // Save Screenshot
            File file = new File(path);
            FileOutputStream outputStream = new FileOutputStream(file);
            bitmap.compress(Bitmap.CompressFormat.PNG, 100, outputStream);
            outputStream.flush();
            outputStream.close();
            openScreenshot(file, activity);
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
}

```

```

    }
}
private void openScreenshot(File imageFile, Activity activity){
    Intent intent = new Intent();
    intent.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
    intent.setAction(Intent.ACTION_VIEW);
    Uri uri = Uri.fromFile(imageFile);
    intent.setDataAndType(uri, "image/*");
    activity.startActivity(intent);
}
}
public class JPEGStrategy implements Strategy{
    @Override
    public void strategy(Activity activity) {
        Date now = new Date();
        String fileName = "screenshot"
            + android.text.format.DateFormat.format("yyyy-MM-dd_hh:mm:ss", now);
        String fileFormat = ".jpg";

        try {
            String path = Environment.getExternalStorageDirectory().toString()
                + "/" +
                fileName + fileFormat;

            // Screen capture
            View view = activity.getWindow().getDecorView().getRootView();
            view.setDrawingCacheEnabled(true);
            Bitmap bitmap = Bitmap.createBitmap(view.getDrawingCache());
            view.setDrawingCacheEnabled(false);

            // Save Screenshot
            File file = new File(path);
            FileOutputStream outputStream = new FileOutputStream(file);
            bitmap.compress(Bitmap.CompressFormat.JPEG, 100, outputStream);
            outputStream.flush();
            outputStream.close();
            openScreenshot(file, activity);
        } catch (Throwable e){
            e.printStackTrace();
        }
    }
private void openScreenshot(File imageFile, Activity activity){
    Intent intent = new Intent();
    intent.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
    intent.setAction(Intent.ACTION_VIEW);
    Uri uri = Uri.fromFile(imageFile);
    intent.setDataAndType(uri, "image/*");
    activity.startActivity(intent);
}
}
}

```

A.1.6 Package; enums

```

public enum Direction{
    North,
    South,
    East,
    West,
}

```

```
}
    Center;
}
```

A.1.7 Package; gameObjects

```
public class Door extends StaticSite {

    // Room
    protected Room room1;
    protected Room room2;

    // Boolean
    protected boolean isOpen;

    // Class
    protected Setting setting = Setting.getInstance();

    public Door(Direction d, Context c) {
        super(c, d);
        isOpen = false;

        if (d == Direction.North || d == Direction.South) {
            width = setting.getImgWidthMap().get(North);
            height = setting.getImgHeightMap().get(North);
            sprite.setLayoutParams(new FrameLayout.LayoutParams(width,
                height));
            sprite.setScaleType(ImageView.ScaleType.FIT_XY);
            sprite.setImageResource(R.drawable.dooreast);
        } else {
            width = setting.getImgWidthMap().get(East);
            height = setting.getImgHeightMap().get(East);
            sprite.setLayoutParams(new FrameLayout.LayoutParams(width,
                height));
            sprite.setScaleType(ImageView.ScaleType.FIT_XY);
            sprite.setImageResource(R.drawable.doornorth);
        }
    }
}

public interface MapSite {
    boolean enter();
}

public abstract class StaticSite implements MapSite {
    // Context
    protected Context context;
    // Sprite
    protected ImageView sprite;

    // Default width & height
    protected int width = 100;
    protected int height = 100;
    // Direction
    protected Direction direction;

    public StaticSite(Context c, Direction d){
        this.direction = d;
        this.context = c;
        this.sprite = new ImageView(this.context);
    }
}
```

```

public abstract void accept(Visitor v);

public boolean intersects(ImageView view2) {
    final int[] location = new int[2];

    sprite.getLocationInWindow(location);
    Rect rect1 = new Rect(location[0], location[1], location[0] + sprite.
        getWidth(), location[1] + sprite.getHeight());

    view2.getLocationInWindow(location);
    Rect rect2 = new Rect(location[0], location[1], location[0] + view2.
        getWidth(), location[1] + view2.getHeight());

    return rect1.intersect(rect2);
}
// Getters and setters
}
public Door(Room r1, Room r2, Direction d, Context c) {
    super(c, d);
    this.room1 = r1;
    this.room2 = r2;
    this.isOpen = true;

    if (d == Direction.North || d == Direction.South) {
        width = setting.getImgWidthMap().get(North);
        height = setting.getImgHeightMap().get(North);
        sprite.setImageResource(R.drawable.dooreast);
    } else {
        width = setting.getImgWidthMap().get(East);
        height = setting.getImgHeightMap().get(East);
        sprite.setImageResource(R.drawable.doornorth);
    }
    sprite.setLayoutParams(new FrameLayout.LayoutParams(width, height));
    sprite.setScaleType(ImageView.ScaleType.FIT_XY);
}
public Room otherSideFrom(Room r1) {
    if (r1.equals(this.room1)) {
        return room2;
    } else if (r1.equals(this.room2)) {
        return room1;
    } else {
        return null;
    }
}
@Override
public boolean enter() {
    if (!isOpen) {
        return false;
    } else {
        return true;
    }
}
@Override
public void accept(Visitor v) {
    v.visitDoor(this);
}
}
public class BombedDoor extends Door {
    public BombedDoor(Room r1, Room r2, Direction d, Context c){
        super(r1, r2, d, c);
    }
}

```

```

    }
    public BombedDoor(Direction d, Context c){
        super(d,c);
    }
    @Override
    public boolean enter() {
        // Door exploded!
        return false;
    }
}
}
public class StandardDoor extends Door {
    public StandardDoor(Room r1, Room r2, Direction d, Context c){
        super(r1,r2,d,c);
    }
    public StandardDoor(Direction d, Context c){
        super(d,c);
    }
}
public class Wall extends StaticSite {
    // Setting
    protected Setting setting = Setting.getInstance();

    public Wall(Direction d, Context c){
        super(c,d);

        if(d == North || d == South){
            width = setting.getImgWidthMap().get(North);
            height = setting.getImgHeightMap().get(North);
            sprite.setImageResource(R.drawable.walleast);
        }
        else {
            width = setting.getImgWidthMap().get(East);
            height = setting.getImgHeightMap().get(East);
            sprite.setImageResource(R.drawable.wallnorth);
        }
        sprite.setLayoutParams(new FrameLayout.LayoutParams(width, height));
        sprite.setScaleType(ImageView.ScaleType.FIT_XY);
    }
    @Override
    public boolean enter() {
        return false;
    }
    @Override
    public void accept(Visitor v) {
        v.visitWall(this);
    }
}
public class BombedWall extends Wall {
    public BombedWall(Direction d, Context c) {
        super(d, c);
    }
    @Override
    public boolean enter() {
        // Wall exploded!
        return false;
    }
}
}
public class StandardWall extends Wall {
    public StandardWall(Direction d, Context c) {
        super(d, c);
    }
}

```



```

    }
}
public class Player extends MovableObject {

    private HashMap<Direction , Integer> imgManagerL = new HashMap<>();
    private HashMap<Direction , Integer> imgManagerR = new HashMap<>();

    public Player(Context c, Direction d){
        super(c, d);

        imgManagerL.put(North , R.drawable.upl);
        imgManagerL.put(South , R.drawable.downl);
        imgManagerL.put(East , R.drawable.right);
        imgManagerL.put(West , R.drawable.left);

        imgManagerR.put(North , R.drawable.upr);
        imgManagerR.put(South , R.drawable.downr);

        sprite.setLayoutParams(new FrameLayout.LayoutParams(121, 108));
        sprite.setScaleType(ImageView.ScaleType.FIT_XY);
        this.sprite.setImageResource(imgManagerL.get(d));

        this.velocity = new Point(20,20);
    }

    @Override
    public void update(float elapsedTime) {
        // update player
    }
    @Override
    public ImageView getSprite() {
        return this.sprite;
    }
    public void moveRight() {
        if(lastDirection != East){
            changeDirection(East);
        }
        position.x += velocity.x;
        this.sprite.setX(position.x);
    }
    public void moveLeft() {
        if(lastDirection != West){
            changeDirection(West);
        }
        position.x -= velocity.x;
        this.sprite.setX(position.x);
    }
    public void moveUp() {
        if(lastDirection != North){
            changeDirection(North);
        }
        position.y -= velocity.y;
        this.sprite.setY(position.y);
    }
    public void moveDown() {
        if(lastDirection != South){
            changeDirection(South);
        }
        position.y += velocity.y;
        this.sprite.setY(position.y);
    }
}

```

```

public void changeDirection(Direction d){
    if(d == South || d == North){
        if(lastDirection == West){
            sprite.setImageResource(imgManagerL.get(d));
        }else{
            sprite.setImageResource(imgManagerR.get(d));
        }
    }else{
        sprite.setImageResource(imgManagerL.get(d));
    }
    lastDirection = d;
}
// Getters and setters
}
public abstract class Pickup extends StaticSite {
    protected double weight;
    public Pickup(Context c, double weight){
        super(c, North);
        this.weight = weight;
        sprite.setLayoutParams(new FrameLayout.LayoutParams(width, height));
        sprite.setScaleType(ImageView.ScaleType.FIT_XY);
    }
    public double getWeight() {
        return weight;
    }
}
public class Bronze extends Pickup {

    public String rareType = "Bronze";
    private double value;

    public Bronze(Context c, double weight){
        super(c, weight);
        sprite.setImageResource(R.drawable.bronze);
        value = 0.01;
    }
    public String getRareType() {
        return rareType;
    }
    @Override
    public void accept(Visitor v) {
        v.visitBronze(this);
    }
    @Override
    public boolean enter() {
        return false;
    }
    public double getValue() {
        return value;
    }
}
public class Silver extends Pickup {

    public String rareType = "Silver";
    private double value;

    public Silver(Context c, double weight){
        super(c, weight);
        sprite.setImageResource(R.drawable.silver);
        value = 0.1;
    }
}

```

```

    }
    public String getRareType() {
        return rareType;
    }
    @Override
    public void accept(Visitor v) {
        v.visitSilver(this);
    }
    @Override
    public boolean enter() {
        return false;
    }
    public double getValue() {
        return value;
    }
}
public class Gold extends Pickup {
    public String rareType = "Gold";
    private double value;

    public Gold(Context c, double weight){
        super(c, weight);
        sprite.setImageResource(R.drawable.gold);
        value = 1;
    }
    public String getRareType() {
        return rareType;
    }
    @Override
    public void accept(Visitor v) {
        v.visitGold(this);
    }
    @Override
    public boolean enter() {
        return false;
    }
    public double getValue() {
        return value;
    }
}
public class Room implements MapSite {
    private Map<Direction, StaticSite> staticSiteMap = new HashMap<>();
    private Point roomPosition = new Point(0,0);
    private Context c;

    // Room number
    private int roomNumber;

    private int roomWidthCenter = Setting.getInstance().getRoomWidth();
    private int roomHeightCenter = Setting.getInstance().getRoomHeight();

    // Constructor
    public Room(int roomNo){
        this.roomNumber = roomNo;
    }

    // Add MapSite in Direction [North, South, East, West]
    public void addSite(StaticSite staticSite){
        staticSiteMap.put(staticSite.getDirection(), staticSite);
    }
    // Add MapSite in Direction [North, South, East, West]

```

```

public void addSite(Direction direction , StaticSite staticSite){
    staticSiteMap.put(direction , staticSite);
}

public void positionAllSites() {
    if(staticSiteMap.isEmpty()) {
        return;
    }
    for (Map.Entry<Direction , StaticSite> entry : staticSiteMap.entrySet
        ()) {
        switch (entry.getKey()) {
            case North:
                entry.getValue().set(roomPosition.x, roomPosition.y -
                    entry.getValue().getHeight());
                break;
            case East:
                entry.getValue().set(roomPosition.x + roomWidthCenter -
                    entry.getValue().getWidth(), roomPosition.y);
                break;
            case South:
                entry.getValue().set(roomPosition.x, roomPosition.y +
                    roomHeightCenter);
                break;
            case West:
                entry.getValue().set(roomPosition.x, roomPosition.y);
                break;
            case Center:
                entry.getValue().set(roomPosition.x + roomWidthCenter /
                    3, roomPosition.y + roomHeightCenter / 2);
                break;

            default:
        }
    }
}

@Override
public boolean enter() {
    // enter room
}

// Getter and Setters...
public List<StaticSite> getAllSites() {
    List<StaticSite> listMapSite = new ArrayList<>();
    for (Map.Entry<Direction , StaticSite> entry : staticSiteMap.entrySet
        ()) {
        listMapSite.add(entry.getValue());
    }
    return listMapSite;
}

public StaticSite removeStaticSite(Direction d) {
    return staticSiteMap.remove(d);
}

public StaticSite getStaticSite(Direction d) {
    return staticSiteMap.get(d);
}
}

```

A.1.8 Package; project

```

public class Engine {

```

```

private static Engine engine = null;
private Context context;
// View
public FrameLayout gridLayout;
public Button enterDoorButton;
public TextView moneyText;
public int frameHeight;
public int frameWidth;

public Timer timer;
public Handler handler = new Handler();
public Maze maze;
public Room currentRoom;
private HashMap<Direction, StaticSite> staticSiteMap;
// Player
public Player player;
private double playerMoney = 0;

public StandardDoor currentDoor;
public boolean intersectsFlag;

public Command enter;
private EnterSiteVisitor enterSiteVisitor = new EnterSiteVisitor();

private Engine() {}
// Singleton
public static Engine getInstance() {
    if(engine == null){
        engine = new Engine();
    }
    return engine;
}

public void startGame(Context context) {
    this.context = context;
    if(frameHeight == 0) {
        frameHeight = gridLayout.getHeight();
        frameWidth = gridLayout.getWidth();
    }
    player = new Player(this.context, East);
    maze = createMaze(new StandardMazeFactory());
    display(maze);
    movePlayerToRoom(maze.getRoomList().get(0));

    this.timer = new Timer();
    timer.schedule(
        new TimerTask(){
            @Override
            public void run() {
                handler.post(new Runnable() {
                    @Override
                    public void run() {
                        update();
                    }
                });
            }
        }, 0, 20);
}

public Maze createMaze(MazeFactory factory) {
    Maze aMaze = factory.makeMaze();
}

```

```

Room r1 = factory.makeRoom(1);
Room r2 = factory.makeRoom(2);
Room r2_2 = factory.makeRoom(22);
Room r3 = factory.makeRoom(3);
Room r4 = factory.makeRoom(4);

Door locked = factory.makeDoor(r1, null, North, context);
locked.setIsOpen(false);
Door door1 = factory.makeDoor(r1,r2, East, context);
Door door2 = factory.makeDoor(r2,r3, North, context);
Door door2_2 = factory.makeDoor(r2,r2_2, East, context);
Door door3 = factory.makeDoor(r3,r4, South, context);

r1.addSite(locked);
r1.addSite(East, door1);
r1.addSite(factory.makeWall(South, context));
r1.addSite(factory.makeWall(West, context));

r2.addSite(East, door2_2);
r2.addSite(factory.makeWall(North, context));
r2.addSite(West, door1);
r2.addSite(Center, factory.makePickUp(context));
r2.addSite(South, door2);

r2_2.addSite(factory.makeWall(North, context));
r2_2.addSite(West, door2_2);
r2_2.addSite(factory.makeWall(East, context));
r2_2.addSite(Center, factory.makePickUp(context));
r2_2.addSite(factory.makeWall(South, context));

r3.addSite(factory.makeWall(West, context));
r3.addSite(factory.makeWall(East, context));
r3.addSite(Center, factory.makePickUp(context));
r3.addSite(South, door3);
r3.addSite(North, door2);

r4.addSite(factory.makeWall(East, context));
r4.addSite(factory.makeWall(West, context));
r4.addSite(North, door3);
r4.addSite(Center, factory.makePickUp(context));

aMaze.AddRoom(r1, East);
aMaze.AddRoom(r2, East);
aMaze.AddRoom(r2_2, East);
aMaze.AddRoom(null, West);
aMaze.AddRoom(r3, South);
aMaze.AddRoom(r4, South);

return aMaze;
}
// Called every frame
private void update(){
    if(!(staticSiteMap.isEmpty())){
        for(Map.Entry<Direction, StaticSite> entry : staticSiteMap.
            entrySet()){
            if(entry.getValue().intersects(player.getSprite())) {
                if(!intersectsFlag) {
                    entry.getValue().accept(enterSiteVisitor);
                }
            }
        }
    }
}

```

```

        if (!(entry.getKey() == Center)) {
            intersectsFlag = true;
        }
    }
}
player.update();
}
public void movePlayerToRoom(Room room){
    if(room == null) return;
    currentRoom = room;
    currentDoor = null;
    player.setPosition(room.getRoomPosition().x + room.
        getRoomWidthCenter()/3, room.getRoomPosition().y + room.
        getRoomHeightCenter()/4);
    staticSiteMap = (HashMap) currentRoom.getStaticSiteMap();
    intersectsFlag = false;
}
public void display(Maze maze){
    GridLayout.addView(player.getSprite());
    displayEachRoomFromMaze(maze);
}
public void displayEachRoomFromMaze(Maze maze){
    List<Room> roomList = maze.getRoomList();
    for (Room room : roomList) {
        for (StaticSite staticSite : room.getAllSites()) {
            if(GridLayout.indexOfChild(staticSite.getSprite()) == -1){
                GridLayout.addView(staticSite.getSprite());
            }
        }
    }
}
public void positionEachRoomFromMaze(Maze maze){
    maze.positionAllRooms();
    for (Room room : maze.getRoomList()) {
        room.positionAllSites();
    }
}
// Execute Command
public void executeEnter(){
    if(this.enter != null){
        enter.execute();
    }
}
public void updateMoneyText() {
    DecimalFormat df = new DecimalFormat();
    df.setMaximumFractionDigits(4);
    this.moneyText.setText("Money: □" + df.format(playerMoney) + "g");
}
public void incrementPlayerMoney(double playerMoney) {
    this.playerMoney += playerMoney;
    updateMoneyText();
}
// Getters and setters...
public ImageView getPlayerSprite(){
    return this.player.getSprite();
}
public void setEnter(Command enter) {
    this.enter = enter;
}
}

```

```

}
public class Maze {
    private List<Room> roomList = new ArrayList<>();

    private int startMazeX = Setting.getInstance().getStartMazeX();
    private int startMazeY = Setting.getInstance().getStartMazeY();
    private int nextRoomY = Setting.getInstance().getRoomHeight();
    private int nextRoomX = Setting.getInstance().getRoomWidth();
    Point nextRoomCenter = new Point(startMazeX, startMazeY);

    public Maze() {}
    public void AddRoom(Room room, Direction direction){
        if(roomList.isEmpty()){
            if(room != null) {
                roomList.add(room);
                room.setRoomPosition(nextRoomCenter);
            }
        }else {
            switch (direction) {
                case North:
                    nextRoomCenter = new Point(nextRoomCenter.x,
                        nextRoomCenter.y - nextRoomY);
                    break;
                case East:
                    nextRoomCenter = new Point(nextRoomCenter.x + nextRoomX,
                        nextRoomCenter.y);
                    break;
                case South:
                    nextRoomCenter = new Point(nextRoomCenter.x,
                        nextRoomCenter.y + nextRoomY); //+ offsetY
                    break;
                case West:
                    nextRoomCenter = new Point(nextRoomCenter.x - nextRoomX,
                        nextRoomCenter.y);
                    break;
                default:
                    break;
            }
            if (room != null) {
                roomList.add(room);
                room.setRoomPosition(nextRoomCenter);
            }
        }
    }
    //Getters and setters...
    public Room getRoomNu(int roomNu) {
        for(int i = 0; i < roomList.size(); i++){
            if(roomList.get(i).getRoomNumber() == roomNu){
                return roomList.get(i);
            }
        }
        return null;
    }
    public List<Room> getRoomList() {
        return roomList;
    }
}
public class Setting {

    private static Setting instance = null;

```



```

private int startMazeX;
private int startMazeY;
.
.
.
private HashMap<Direction , Integer> imgWidthMap = new HashMap<>();
private HashMap<Direction , Integer> imgHeightMap = new HashMap<>();

private Setting(){
    int number = 300;
    roomWidth = number;
    roomHeight = number;

    imgWidthEast = 40;
    imgHeightEast = number;
    imgWidthNorth = number;
    imgHeightNorth = 40;
    startMazeX = 100;
    startMazeY = 400;

    imgWidthMap.put(North , imgWidthNorth);
    imgHeightMap.put(North , imgHeightNorth);
    imgWidthMap.put(East , imgWidthEast);
    imgHeightMap.put(East , imgHeightEast);

}
public static Setting getInstance() {
    if(instance == null) {
        instance = new Setting();
    }
    return instance;
}
//Getters and setters...
public HashMap<Direction , Integer> getImgHeightMap() {
    return imgHeightMap;
}
.
.
.
}

```

A.2 Project with Mobile Design Patterns

We will not present and repeat all of the classes we have seen from the regular project above when presenting the implementation of the mobile project because they are pretty much alike. We present only the mobile design patterns and main differences. The main differences aside from the mobile design patterns were that we added an ImageManager class which represents the technique of combining multiple images to one, and a factory enum which decides whether we use a bombed maze factory or the standard.

A.2.1 Package; mobileAbstractFactory

```

public class MazeFactory {
    public static Maze makeMaze() {
        Maze maze = null;
        switch(Setting.factoryType){

```

```

        case Bombed:
            maze = BombedMazeFactory.makeMaze();
            break;
        default:
            maze = StandardMazeFactory.makeMaze();
    }
    return maze;
}
}
public static Wall makeWall(Direction d, Context c){
    Wall wall = null;
    switch(Setting.factoryType){
        case Bombed:
            wall = BombedMazeFactory.makeWall(d, c);
            break;
        default:
            wall = StandardMazeFactory.makeWall(d, c);
    }
    return wall;
}
}
public static Door makeDoor(Room r1, Room r2, Direction d, Context c){
    Door door = null;
    switch(Setting.factoryType){
        case Bombed:
            door = BombedMazeFactory.makeDoor(r1, r2, d, c);
            break;
        default:
            door = StandardMazeFactory.makeDoor(r1, r2, d, c);
    }
    return door;
}
}
public static Room makeRoom(int roomNumber){
    Room room = null;
    switch(Setting.factoryType){
        case Bombed:
            room = BombedMazeFactory.makeRoom(roomNumber);
            break;
        default:
            room = StandardMazeFactory.makeRoom(roomNumber);
    }
    return room;
}
}
public static Pickup makeRandomPickUp(Context c, double weight){
    Pickup pickUp = null;
    switch(Setting.factoryType){
        case Bombed:
            pickUp = BombedMazeFactory.makeRandomPickUp(c, weight);
            break;
        default:
            pickUp = StandardMazeFactory.makeRandomPickUp(c, weight);
    }
    return pickUp;
}
}
}
public class BombedMazeFactory {
    public static Maze makeMaze(){
        return new Maze();
    }
}
public static Door makeDoor(Room r1, Room r2, Direction direction,
    Context context) {
    return new BombedDoor(r1, r2, direction, context);
}
}

```

```

public static Room makeRoom(int roomNumber) {
    return new Room(roomNumber);
}
public static Wall makeWall(Direction direction, Context context) {
    return new BombedWall(direction, context);
}
public static Pickup makeRandomPickUp(Context c, double weight) {
    return new Bronze(c, weight);
}
}
public class StandardMazeFactory {
    public static Maze makeMaze(){
        return new Maze();
    }
    public static Door makeDoor(Room r1, Room r2, Direction direction,
        Context context) {
        return new StandardDoor(r1, r2, direction, context);
    }
    public static Room makeRoom(int roomNumber) {
        return new Room(roomNumber);
    }
    public static Wall makeWall(Direction direction, Context context) {
        return new StandardWall(direction, context);
    }
    public static Pickup makeRandomPickUp(Context c, double weight) {
        Random random = new Random();
        int randWeight = random.nextInt(100);
        switch (random.nextInt(3)){
            case 0: return new Bronze(c, randWeight);
            case 1: return new Silver(c, randWeight);
            case 2: return new Gold(c, randWeight);
            default: return new Bronze(c, randWeight);
        }
    }
}
}

```

A.2.2 Package; mobileCommand

```

public abstract class Command {
    public Engine engine = Engine.getInstance();
    public Command(){}
    public abstract void execute();
}
public class EnterCommand extends Command {
    StaticSite staticSite;
    public EnterCommand(StaticSite s){
        super();
        this.staticSite = s;
    }
    @Override
    public void execute(){
        if(this.staticSite == null){
            return;
        }
        if(staticSite.intersects(engine.getPlayerSprite()) && staticSite.
            enter()) {
            if (staticSite.enter()) {
                Door door = (Door) staticSite;
            }
        }
    }
}

```

```

        engine.movePlayerToRoom(door.otherSideFrom(engine.
            getCurrentRoom()));
    }
}
}
}

```

A.2.3 Package; mobileVisitor

```

public interface Visitor {
    void visitStaticSite(StaticSite staticSite);
    void visitPickUp(PickUp pickUp);
}
public class PickupVisitor implements Visitor {
    @Override
    public void visitStaticSite(StaticSite staticSite) {}
    @Override
    public void visitPickUp(PickUp pickUp) {
        double value = pickUp.getWeight() * pickUp.getValue();
        Engine.getInstance().incrementPlayerMoney(value);
    }
}
public class EnterSiteVisitor implements Visitor {
    @Override
    public void visitStaticSite(StaticSite staticSite) {
        Engine.getInstance().setEnter(new EnterCommand(staticSite));
    }
    @Override
    public void visitPickUp(PickUp pickUp) {
        PickupVisitor pickUpVisitor = new PickupVisitor();
        pickUp.accept(pickUpVisitor);
        Engine.getInstance().removeView(pickUp.getSprite());
    }
}
}

```

A.2.4 Package; mobileStrategy

```

public class PhotoStrategy {
    public static void strategy(String fileFormat, Activity activity) {
        Date now = new Date();
        String fileName = "screenshot"
            + android.text.format.DateFormat.format("yyyy-MM-dd_hh:mm:ss", now);

        try {
            String path = Environment.getExternalStorageDirectory().toString()
                + "/" +
                fileName + fileFormat;

            // Screen capture
            View view = activity.getWindow().getDecorView().getRootView();
            view.setDrawingCacheEnabled(true);
            Bitmap bitmap = Bitmap.createBitmap(view.getDrawingCache());
            view.setDrawingCacheEnabled(false);

            // Save Screenshot
            File file = new File(path);
            FileOutputStream outputStream = new FileOutputStream(file);

```

```

        if(fileFormat == ".png") {
            bitmap.compress(Bitmap.CompressFormat.PNG, 100, outputStream
                ); }
        else {
            bitmap.compress(Bitmap.CompressFormat.JPEG, 100,
                outputStream);
        }
        outputStream.flush();
        outputStream.close();
        openScreenshot(file, activity);
    } catch(Throwable e){
        e.printStackTrace();
    }
}
private static void openScreenshot(File imageFile, Activity activity){
    Intent intent = new Intent();
    intent.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
    intent.setAction(Intent.ACTION_VIEW);
    Uri uri = Uri.fromFile(imageFile);
    intent.setDataAndType(uri, "image/*");
    activity.startActivity(intent);
}
}
}

```

A.2.5 Package; enums

```

public enum FactoryType {
    Standard,
    Bombed;
}

```

A.2.6 Package; imageManager

```

public class ImageManager {
    private static ImageManager instance = null;
    private HashMap<Direction, Drawable> doorImage;
    private HashMap<Direction, Drawable> wallImage;
    private HashMap<String, Drawable> moneyImage;
    private HashMap<Direction, Drawable> playerImageL;
    private HashMap<Direction, Drawable> playerImageR;

    private Context context;
    private ImageManager(Context c){
        this.context = c;
        doorImage = new HashMap<>();
        wallImage = new HashMap<>();
        moneyImage = new HashMap<>();
        playerImageL = new HashMap<>();
        playerImageR = new HashMap<>();

        Bitmap sourcePlayer = Bitmap.createScaledBitmap(BitmapFactory.
            decodeResource(context.getResources(), R.drawable.player),
                700, 121, false);
        Bitmap upRBm = Bitmap.createBitmap(sourcePlayer, 363, 0, 121, 108);
        Bitmap upLBm = Bitmap.createBitmap(sourcePlayer, 242, 0, 121, 108);
        Bitmap downLBm = Bitmap.createBitmap(sourcePlayer, 0, 0, 121, 108);
    }
}

```

```

Bitmap downRbm = Bitmap.createBitmap(sourcePlayer, 121, 0, 121, 108)
;
Bitmap rightBm = Bitmap.createBitmap(sourcePlayer, 592, 0, 108, 121)
;
Bitmap leftBm = Bitmap.createBitmap(sourcePlayer, 484, 0, 108, 121);

Drawable upRbmD = new BitmapDrawable(context.getResources(), upRbm);
Drawable upLbmD = new BitmapDrawable(context.getResources(), upLbm);
Drawable downLbmD = new BitmapDrawable(context.getResources(),
    downLbm);
Drawable downRbmD = new BitmapDrawable(context.getResources(),
    downRbm);
Drawable rightBmD = new BitmapDrawable(context.getResources(),
    rightBm);
Drawable leftBmD = new BitmapDrawable(context.getResources(), leftBm
);

playerImageL.put(North, upLbmD);
playerImageL.put(South, downLbmD);
playerImageL.put(East, rightBmD);
playerImageL.put(West, leftBmD);

playerImageR.put(North, upRbmD);
playerImageR.put(South, downRbmD);

Bitmap sourceDoor = Bitmap.createScaledBitmap(BitmapFactory.
    decodeResource(context.getResources(), R.drawable.door),
    42,169, false);
Bitmap doorNorth = Bitmap.createBitmap(sourceDoor, 0, 0, 21, 40);
Bitmap doorEast = Bitmap.createBitmap(sourceDoor, 21, 0, 21, 169);
Drawable dN = new BitmapDrawable(context.getResources(), doorNorth);
Drawable dE = new BitmapDrawable(context.getResources(), doorEast);

doorImage.put(North, dN);
doorImage.put(East, dE);

Bitmap sourceWall = Bitmap.createScaledBitmap(BitmapFactory.
    decodeResource(context.getResources(), R.drawable.wall),
    190,169, false);
Bitmap wallNorth = Bitmap.createBitmap(sourceWall, 0, 0, 169, 50);
Bitmap wallEast = Bitmap.createBitmap(sourceWall, 169, 0, 21, 169);
Drawable wN = new BitmapDrawable(context.getResources(), wallNorth);
Drawable wE = new BitmapDrawable(context.getResources(), wallEast);
wallImage.put(North, wN);
wallImage.put(East, wE);

Bitmap sourceMoney = Bitmap.createScaledBitmap(BitmapFactory.
    decodeResource(context.getResources(), R.drawable.money),
    627,170, false);
Bitmap gold = Bitmap.createBitmap(sourceMoney, 418, 0, 209, 170);
Bitmap silver = Bitmap.createBitmap(sourceMoney, 209, 0, 209, 170);
Bitmap bronze = Bitmap.createBitmap(sourceMoney, 0, 0, 209, 170);
Drawable g = new BitmapDrawable(context.getResources(), gold);
Drawable s = new BitmapDrawable(context.getResources(), silver);
Drawable b = new BitmapDrawable(context.getResources(), bronze);

moneyImage.put("gold", g);
moneyImage.put("silver", s);
moneyImage.put("bronze", b);
}

```

```

// Singleton
public static ImageManager getInstance(Context c) {
    if(instance == null) {
        instance = new ImageManager(c);
    }
    return instance;
}
// Getters and setters...
public HashMap<Direction, Drawable> getDoorImage() {
    return doorImage;
}
.
.
.
public HashMap<Direction, Drawable> getPlayerImageR() {
    return playerImageR;
}
}

```

A.2.7 Package; mobileGameObjects

```

public class Player extends MovableObject {

    private HashMap<Direction, Drawable> imgManagerL = new HashMap<>();
    private HashMap<Direction, Drawable> imgManagerR = new HashMap<>();

    public Player(Context c, Direction d){
        super(c, d);

        // Using imageManager to get the desired image parts of the player's
        // spritesheet
        ImageManager imageManager = ImageManager.getInstance(c);

        Drawable upLbm = imageManager.getPlayerImageL().get(North);
        Drawable downLbm = imageManager.getPlayerImageL().get(South);
        Drawable rightBm = imageManager.getPlayerImageL().get(East);
        Drawable leftBm = imageManager.getPlayerImageL().get(West);

        Drawable upRbm = imageManager.getPlayerImageR().get(North);
        Drawable downRbm = imageManager.getPlayerImageR().get(South);

        imgManagerL.put(North, upLbm);
        imgManagerL.put(South, downLbm);
        imgManagerL.put(East, rightBm);
        imgManagerL.put(West, leftBm);

        imgManagerR.put(North, upRbm);
        imgManagerR.put(South, downRbm);

        this.sprite.setImageDrawable(rightBm);
        sprite.setLayoutParams(new FrameLayout.LayoutParams(121, 108));
        sprite.setScaleType(ImageView.ScaleType.FIT_XY);
    }

    public void moveRight() {
        if(lastDirection != East){
            changeDirection(East);
        }
    }
}

```

```

    }
    position.x += velocity.x;
    this.sprite.setX(position.x);
}
.
.
public void changeDirection(Direction d){
    if(d == South || d == North){
        if(lastDirection == West){
            sprite.setImageDrawable(imgManagerL.get(d));
        }else{
            sprite.setImageDrawable(imgManagerR.get(d));
        }
    }else{
        sprite.setImageDrawable(imgManagerL.get(d));
    }
    lastDirection = d;
}
.
.
}
public class Door extends StaticSite {
    .
    .
    public Door(Direction d, Context c) {
        super(c, d);
        isOpen = false;

        // Using imageManager to get the desired image parts of the Door's
        // spritesheet
        ImageManager imageManager = ImageManager.getInstance(context);
        Drawable part;

        if (d == North || d == Direction.South) {
            part = imageManager.getDoorImage().get(North);
            width = setting.getImgWidthMap().get(North);
            height = setting.getImgHeightMap().get(North);
        } else {
            part = imageManager.getDoorImage().get(East);
            width = setting.getImgWidthMap().get(East);
            height = setting.getImgHeightMap().get(East);
        }
        sprite.setLayoutParams(new FrameLayout.LayoutParams(width, height));
        sprite.setScaleType(ImageView.ScaleType.FIT_XY);
        sprite.setImageDrawable(part);
    }
    .
    .
}

```


Appendix B

Static Method vs Instance Method

In this appendix, we present the implementation of the static method versus the instance method test, where we measured both the execution time, CPU usage, and memory of both methods.

```
public class Employee {
    public void heavyOperation() {
        int a = 1;
        int b = 1;
        int operation = a+b;
    }
}
public class StaticEmployee {
    public static void staticHeavyOperation() {
        int a = 1;
        int b = 1;
        int operation = a+b;
    }
}
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void invokeTest(View view){
        double time = 0;
        for(int i = 0; i<10; i++){
            //time+=invokeInstanceMethodForEachObject();
            //time+=invokeInstanceMethodFromOneObject();
            time+=invokeStaticMethodFromClassReference();
        }
        System.out.println("Average time: " + (time/10));
    }

    public double invokeInstanceMethodForEachObject(){
        double time = 0;
        long startTime = System.nanoTime();
        for(int i = 0; i < 1E7; i++){
            Employee e = new Employee();
            e.heavyOperation();
        }
    }
}
```

```

    long elapsedTime2 = System.nanoTime() - startTime;
    time = (double) elapsedTime2/1E9;
    System.out.println("Invoke Instance Method for each time a object is
        created:" + time + "s");
    return time;
}
public double invokeInstanceMethodFromOneObject(){
    double time = 0;
    long startTime = System.nanoTime();
    Employee e = new Employee();
    for (int i = 0; i < 1E7; i++) {
        e.heavyOperation();
    }
    long elapsedTime5 = System.nanoTime() - startTime;
    time = (double) elapsedTime5 / 1E9;
    System.out.println("Invoke Instance Method from one Object reference
        ::" + time + "s");
    return time;
}
public double invokeStaticMethodFromClassReference(){
    double time = 0;
    long startTime = System.nanoTime();
    for (int i = 0; i < 1E7; i++) {
        StaticEmployee.staticHeavyOperation();
    }
    long elapsedTime5 = System.nanoTime() - startTime;
    time = (double) elapsedTime5 / 1E9;
    System.out.println("Invoke Static Method from Class reference::" +
        time + "s");
    return time;
}
}
}

```

Code Snippet B.1: Code of our Static Method versus Instance Method Test

References

- [1] Android profile. <https://developer.android.com/studio/profile>, Februar 2020. Accessed on 2020-02-27.
- [2] Christopher Alexander. *A pattern language: towns, buildings, construction*. Oxford university press, 1977.
- [3] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on software engineering*, 28(1):4–17, 2002.
- [4] Reza B'Far. *Mobile computing principles : designing and developing mobile applications with uml and xml*, 2005.
- [5] Barry W Boehm, John R Brown, Hans Kaspar, M Lipow, and G MacLeod. Merritt.: Characteristics of software quality, 1978.
- [6] Kayun Chantarasathaporn and Chonawat Srisa-an. Energy conscious factory method design pattern for mobile devices with c# and intermediate language. In *Proceedings of the 3rd International Conference on Mobile Technology, Applications & Systems*, Mobility '06, New York, NY, USA, 2006. ACM.
- [7] Kayun Chantarasathaporn and Chonawat Srisa-an. Object-oriented programming strategies in c# for power conscious system. *International Journal of Computer Science {Online}*, 1(1), 2006.
- [8] Chua Fang-Fang. *Design patterns for developing high efficiency mobile application*. 2013.
- [9] International Organization for Standardization. Iso/iec 9126. <https://www.iso.org/home.html>, 2001. Accessed on 2020-02-11.
- [10] International Organization for Standardization. Iso/iec 9126, software engineering - product quality - part 1: Quality model. <https://www.iso.org/standard/22749.html>, 2001. Accessed on 2020-02-11.
- [11] International Organization for Standardization. Iso/iec 25010, systems and software quality requirements and evaluation (square) - system and software quality models. <https://www.iso.org/standard/35733.html>, 2011. Accessed on 2020-02-11.
- [12] G. H. Forman and J. Zahorjan. The challenges of mobile computing. *Computer*, 27(4):38–47, April 1994.
- [13] Dominik Franke, Stefan Kowalewski, and Carsten Weise. A mobile software quality model. In *2012 12th International Conference on Quality Software*, pages 154–157. IEEE, 2012.

- [14] Dominik Franke and Carsten Weise. Providing a software quality framework for testing of mobile applications. In *2011 fourth IEEE international conference on software testing, verification and validation*, pages 431–434. IEEE, 2011.
- [15] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.
- [16] Object Management Group. Uml 2.0. <https://www.omg.org/spec/UML/2.0>, 2005. Accessed on 2020-05-13.
- [17] V-M Hartikainen, Pasi P Liimatainen, and Tommi Mikkonen. On mobile java memory consumption. In *14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)*, pages 7–pp. IEEE, 2006.
- [18] Andreas Jetter, Harald Gall, Martin Pinzger, Patrick Knab, and Andreas Jetter. Assessing software quality attributes with source code metrics, 2006.
- [19] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 15–24, Oct 2013.
- [20] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.
- [21] JA McCall, PK Richards, and GF Walters. Factors in software quality. vol. 1, 2, and 3. *Nat'l Tech. Information Service, Springfield, USA*, 1977.
- [22] Microsoft Patterns. *Microsoft Application Architecture Guide*. Microsoft Press, USA, 2nd edition, 2009.
- [23] A. G. Peker and T. Can. A design goal and design pattern based approach for development of game engines for mobile platforms. In *2011 16th International Conference on Computer Games (CGAMES)*, pages 114–120, July 2011.
- [24] Muhammad Ehsan Rana, Wan Nurhayati Wan Ab. Rahman, Masrah Azrifah Azmi Murad, and Rodziah Atan. The impact of flyweight and proxy design patterns on software efficiency: An empirical evaluation. 2019.
- [25] taraloca. How to programmatically take a screenshot on android. <https://stackoverflow.com/questions/2661536/how-to-programmatically-take-a-screenshot-on-android/5651242#5651242>, 2018. Accessed on 2020-05-13.
- [26] Techopedia. Mobile device. <https://www.techopedia.com/definition/23586/mobile-device>, May 2019. Accessed on 2019-05-08.
- [27] Wikipedia. Mobile device. https://en.wikipedia.org/wiki/Mobile_device, May 2019. Accessed on 2019-05-08.
- [28] Wikipedia. Software design pattern. https://en.wikipedia.org/wiki/Software_design_pattern, April 2019. Accessed on 2019-04-12.