Fall 12-19-2020

Analyzing Performance, Energy Consumption, and Reliability of Mobile Applications

Osama Barack
Southern Methodist University, obarack@smu.edu

ANALYZING PERFORMANCE, ENERGY CONSUMPTION,
AND RELIABILITY OF MOBILE APPLICATIONS

Approved by:

Dr. LiGuo Huang, Associate Professor

Dr. Jeff Tian, Professor

Dr. Jennifer Dworak, Associate Professor

Dr. Corey Clark, Assistant Professor

Dr. John Medellin

ANALYZING PERFORMANCE, ENERGY CONSUMPTION, AND RELIABILITY OF MOBILE APPLICATIONS

A Dissertation Presented to the Graduate Faculty of the

Lyle School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Doctor of Philosophy

with a

Major in Computer Science

by

Osama Barack

B.S., King Abdulaziz University, Jeddah, Saudi Arabia, 2001
M.S., Florida institute of Technology, Melbourne, FL 2008

December 19, 2020

ACKNOWLEDGMENTS

I would like to thank the Department of Computer Science and Engineering in the Lyle School of Engineering at Southern Methodist University. I am also very grateful to my Advisor, Dr. LiGuo Huang, for the support and guidance she gave me during my PhD journey. Dr. Huang is one of the smartest and most dedicated professors I met. She has been very supportive and her discussion provided me with guidance and insight in my research. Finally, I would like to appreciate and thank the members of my PhD Committee, Dr. Jeff Tian, Dr. Corey Clark, Dr. Jennifer Dworak, and Dr. John Medellin.

Barack, Osama B.S., King Abdulaziz University, Jeddah, Saudi Arabia, 2001
M.S., Florida institute of Technology, Melbourne, FL 2008

Analyzing Performance, Energy Consumption,
and Reliability of Mobile Applications

Advisor: Dr. LiGuo Huang, Associate Professor
Doctor of Philosophy degree conferred December 19, 2020
Dissertation completed September 09, 2020

Mobile applications have become a high priority for software developers. Researchers and practitioners are working toward improving and optimizing the energy efficiency and performance of mobile applications due to the capacity limitation of mobile device processors and batteries. In addition, mobile applications have become popular among end-users, developers have introduced a wide range of features that increase the complexity of application code.

To improve and enhance the maintainability, extensibility, and understandability of application code, refactoring techniques were introduced. However, implementing such techniques to mobile applications affects energy efficiency and performance. To evaluate and categorize software implementation and optimization efficiency, several metrics are introduced, such as the Greenup, Powerup, and Speedup (GPS-UP) metrics. The first contribution in my work is to quantitatively evaluate the impact of several refactoring

techniques on the energy efficiency and performance of Fowler's sample code in mobile environments. In addition, I introduce two new categories to the GPS-UP metrics to better categorize the impact of refactoring techniques on mobile applications. Moreover, I explain the interrelationship between energy efficiency and performance to provide more knowledge and insight for mobile application developers.

Hence Fowler's sample code is simple and does not reflect an accurate evaluation of the refactoring techniques, I extend my work through presenting a case study that evaluates and categorizes the impact of refactoring techniques when they are applied to open-source mobile applications. In addition, I provide a comparison of the effect of refactoring techniques between the results of Fowler's sample and open-source mobile applications. The results of this contribution will allow software engineers and developers to understand the trade-offs between performance, energy efficiency, and maintainability when implementing refactoring techniques.

The second contribution in my work is to modify the Orthogonal Defect Classification (ODC) model to accommodate defects of mobile applications. The ODC model enables developers to classify defects and track the process of inspection and testing. However, ODC was introduced to classify defects of traditional software. Mobile applications differ from traditional applications in many ways; they are susceptible to external factors, such as screen and network changes, notifications, and phone interruptions, which affect the applications' functioning. The adapted ODC model allows me to address newly introduced application defects found in the mobile domain, such as

energy, notification, and Graphical User Interface (GUI). In addition, based on the new model, I classify found defects of two well-known mobile applications. Moreover, I discuss one-way and two-way analyses. This contribution provides developers with a suitable defect analysis technique for mobile applications.

Software reliability is an important quality attribute, and software reliability models are frequently used to measure and predict software maturity. The nature of mobile environments differs from that of PC and server environments due to many factors, such as the network, energy, battery, and compatibility. Evaluating and predicting mobile application reliability are real challenges because of the diversity of the mobile environments in which the applications are used, and the lack of publicly available defect data. In addition, bug reports are optionally submitted by end-users. In the third contribution of my dissertation, I propose assessing and predicting the reliability of a mobile application using known software reliability growth models (SRGMs). Four software reliability models are used to evaluate the reliability of an open-source mobile application through analyzing bug reports. The results of my work enable software developers and testers to assess and predict the reliability of mobile software applications.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I dedicate this dissertation to my family who always supports me, believes in me, and encourages me, to be successful and achieve my goals.

"PERSISTENCE"

Chapter 1

INTRODUCTION

As known, the contemporary world is dependant on technology and information. Software engineering is the approach of studying the design, development, testing, and maintenance of software. It ensures that the software is built correctly with satisfying all requirements [48], [12], [11]. In addition, software quality is concerned with measuring and addressing the quality level of software projects [32].

The quality model ISO 25010 [58] is introduced to evaluate the quality of a software product or system. The quality model decides which quality attributes will be considered when evaluating the properties of a software product.

Due to the limitation of mobile device hardware capacity, the mobile application code style is considered as a main factor, which affects the performance, energy efficiency, and reliability of mobile applications. Thus, in this work, three critical attributes are selected form the quality model (maintainability, performance, and reliability) to be evaluated and analyzed through the mobile application code.

Maintainability measures the level of effectiveness and efficiency when modifying or improving a software product or system. In addition, performance measures the utilization of the device resources that are used by a software product. Furthermore, reliability assesses a software product performing specified functions under specified conditions for a specified period of time.

Injecting code refactoring techniques to the code as a part of maintaining the mobile application will affect the performance and energy efficiency. In addition, new mobile application defects have surfaced due to the nature of mobile environments. These new defects need to be classified to improve in-process feedback. Moreover, based on the new type of defects, reliability of mobile applications should be assessed and evaluated to determine product maturity, availability, fault tolerance, and recoverability.

In this dissertation, I present three approaches (code refactoring, defect analysis, and reliability) to improve the quality of mobile applications.

1.1. Code Refactoring

Since Apple's and Android's first smartphones were released, the evolution of mobile applications has increased rapidly and continuously. As a result, daily moderate or heavy use of mobile devices led to a decrease in the batteries' charge. Thus, researchers and practitioners have been improving and optimizing the energy efficiency and performance of smartphone applications to extend the battery life.

One of the quality attributes of software is maintainability. In order to achieve maintainability, the software code has to be extensible, readable, and understandable. Fowler Martin [21] introduced refactoring to make existing software maintainable and defined refactoring as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written". However, refactoring also affects performance and energy efficiency for mobile applications by adding, deleting, or moving a number of code lines. Thus, knowing the trade-off between performance and energy efficiency when implementing refactoring techniques is very important for mobile application developers.

Abdulsalam et al. [1] proposed Greenup, Powerup, and Speedup (GPS-UP) metrics to show the interrelationships between energy, performance, and power. The GPS-UP metrics translate the effect of software optimization on the attributes (energy, performance, and power) into a category on the GPS-UP application energy graph. However, to better categorize the impact of refactoring techniques on mobile applications and in response to the experiment results, I introduce two new categories to the GPS-UP metrics.

Code refactoring techniques are considered software application optimization tools. Therefore, I propose a study to investigate code refactoring techniques by using the GPS-UP metrics in a mobile software system. Power is the rate of the energy used; therefore, evaluating only Greenup (the ratio of the total energy consumption of the non-refactored code to the total energy consumption of the refactored code) is not sufficient to discover the reason behind the positive or negative impact on energy efficiency. The solution is to use Powerup which is the measure of the effect of refactoring techniques on the power usage inside the mobile device components. Using Greenup with Powerup enables mobile application developers to understand the advantages and disadvantages of each refactoring technique and how it affects the average power consumption of mobile device components.

In chapter 3, I quantitatively evaluate the impact of several refactoring techniques on the energy efficiency and performance of Fowler's sample code in a mobile application. In addition, I extend this work through evaluating the impact of refactoring techniques that are applied to the software code of mobile applications that contain common algorithms (quick sort and binary search), and data structures (linked list). Moreover, I evaluate the influence of refactoring techniques on open-source mobile applications (Simple Calculator and AnotherMonitor). Finally, I provide a detailed comparison between the results of all case studies.

1.2. Defect Analysis

Due to the importance and popularity of services that mobile applications provide, mobile applications require a short and accurate cycle of defect classification that is adapted to the nature of mobile environments. Ram Chillarege [16] introduced the concept of Orthogonal Defect Classification (ODC) for traditional software and defined it as "a concept that enables in-process feedback to developers by extracting signatures on the development process from defects". ODC was originally introduced to classify defects of traditional software, whereas the nature of mobile applications differs somewhat. In mobile environments, new functionalities and features were developed, which introduced additional factors such as energy, network, incompatibility, Graphical User Interface (GUI), interruption, and notification. Defects originating from these factors need to be classified for better defect resolution. Therefore, to benefit from the ODC concept, ODC needs to be adapted to attend to these new factors.

In chapter 4, I adapt ODC to mobile applications by considering the characteristics of the mobile environment. This includes adding the new factors to the ODC framework to classify defects in order to improve the reliability of mobile applications after release. In addition, I provide one-way and two-way analyses to accommodate the results of the classification process. The provided classification process and analyses are based on defect reports of two mobile applications, Tomdroid and Telegram. Tomdroid [9] is a note-taking application that has a unique wiki-style display in the Android platform, and Telegram [26] is a cloud-based instant messaging and voice over IP service.

1.3. Reliability

Software reliability is a measure for controlling and maintaining the processes of the software development life cycle (SDLC) to develop reliable software. This measure is used during the testing process until the process's exit criteria are met. In addition, software reliability helps to maintain and predict the correctness of the software [56]. Software reliability engineering was introduced to aid in analyzing and measuring the quality of software applications. It presents the quality of the software running without producing defects [47] [46]. Researchers and practitioners have been improving software reliability models to assess the reliability of different types of software.

Measuring and predicting the reliability of a mobile application are real challenges due to the following reasons. First, the nature of mobile environments is different from that of PC and server environments. Second, in mobile environments, new functionalities and features are introduced, such as energy, network, incompatibility, modified and limited Graphical User Interface (GUI), interruption, and notification, which produce new types of defects [7]. Third, mobile operating systems and devices are divers. Fourth, the high demand for mobile applications from users has made the development process fast and the functionality of mobile application more complex [61]. Finally, after a mobile application is released, failures occur in mobile devices. In addition to testing, software developers partially rely on bug reports, which are optionally submitted by end-users.

To assess software reliability in mobile applications, researchers are required to spend more time and effort to evaluate the efficacy of software reliability. Considering the characteristics of mobile applications while measuring their software reliability will produce more accurate results and analyses.

Predicting mobile application failures is as important to software developers as to companies and research organizations. Therefore, I propose measuring the reliability of mobile applications and producing more accurate results based on defects that are extracted from bug reports.

The remainder of this dissertation is structured as follows: Chapter 2 presents related work. Chapter 3 explains the impact of code refactoring techniques for energy efficiency in mobile environments. Chapter 4 explains the adaptation of the ODC concept for mobile application defect analysis. Chapter 5 presents an assessment and prediction of software reliability in mobile applications. Chapter 6 contains the conclusions and perspectives.

Chapter 2

RELATED WORK

In this chapter, I explore the related work to code refactoring analysis, defect analysis, and reliability analysis in mobile environments.

2.1. Code Refactoring

Researchers and practitioners have presented many studies that investigated the relationship between optimizing the software code, performance, and energy efficiency. Their methodologies have different approaches, such as offloading to solve the limitation of mobile device components, code refactoring techniques to make the code maintainable, and different algorithm complexity and data structures to find which one is more effective for improving performance and energy consumption.

Metri et al. [43] proposed a case study that shows the comparisons between hand-held devices' platforms and applications for energy consumption. The comparisons provided useful information about the providers of major platforms, energy consumption of application design, and common practices of application developers.

Medellin et al. [41] modularized a mobile device application by utilizing SOA/BPEL/services design principles and proposed a service that decides which part of the application is migrated to the cloud to be computed. This study proposed a new technique to exceed the limitation of mobile computation and batteries.

Fekete et al. [19] presented a technique called "offloading" and proposed a tool that automates recognizing heavy computational tasks in mobile device systems and sends them to web services. Using outsource machines to compute those tasks decreases energy consumption and makes mobile device batteries last longer.

Ramirez et al. [52] presented a study that measured the energy consumption of multithreading android application executing only java application and compared it to the Android executing complex part of code in C programming language using JNI. The study results help developers to find the cause behind increasing mobile application energy consumption and improve application development strategies aimed at increasing energy efficiency.

Mittal et al. [44] presented an energy emulation tool that allows developers to estimate the energy use for their mobile applications during the developing process on the developers' workstation. Their energy emulation tool has the ability of scaling the emulated resources, including the processing speed and network characteristics to imitate the application behavior on a physical mobile device. In addition, the authors presented a prototype implementation of this tool and evaluated it through comparing real device energy measurements.

implementing refactoring techniques improve understandability, maintainability, and extensibility of the software code. However, the impact of energy efficiency of each refactoring technique is not shown in the automated support of refactoring in IDEs. Sahin et al. [54] presented an empirical study to explore the impact of energy efficiency for 197 application of 6 refactoring techniques. Their experiment results showed the refactoring impact on energy consumption and the capability of increasing and decreasing energy consumption. In addition, the authors showed metrics that correlated with energy consumption in order to predict the impact of implementing refactoring techniques.

Park et al. [50] explored whether code refactoring techniques increase or decrease energy consumption. Park's experiment used XEEMU which is a software power estimation tool that supports C/C++ based estimation to analyze and evaluate Fowler's refactoring techniques.

Silva et al. [18] presented a case study that combined object-oriented languages with code refactoring techniques in embedded software to evaluate the positive and negative effectiveness. The case study results showed helpful information related to energy efficiency and CPU performance.

Morales et al. [45] proposed EARMO(An Energy-aware Refactoring Approach for Mobile Apps), a novel anti-pattern, accounts energy consumption when refactoring mobile anti-patterns. The authors analyzed the impact of eight types of anti-patterns on a testbed of 20 android applications. EARMO has been evaluated by testing three multiobjective search-based algorithms. Their experiment results show that EARMO can generate refactoring recommendations in less than a minute and remove a median of 84% of anti-patterns. In addition, EARMO extended the battery life of a mobile phone by up to 29 minutes, and 68% of EARMO refactoring suggestions were found relevant by developers.

Bunse et al. [14] proposed a case study that showed the interrelationship between sorting algorithm complexity time and its energy efficiency in mobile and embedded devices. However, the experiment showed that there is no direct interrelationship between them.

Zecena et al. [65] explored and analyzed three parallelized sorting algorithms (Odd-Even Sort, ShellSort, and QuickSort) by executing them on multicore computers. The results showed that better algorithm performance leads to more energy savings.

Comito and Talia [17] presented an experimental study that evaluated the energy consumption of mobile devices when executing data mining algorithms. Moreover, they proposed a learning machine that predicts the energy consumption of data-intensive algorithms running on mobile devices.

Rashid et al. [53] analyzed the energy consumption of different implementations of sorting algorithms in different programming languages. The experiment results showed that different combinations of algorithms and programming languages change the level of energy efficiency. The Authors study provides the basic information of selecting algorithms and identifying main factors affecting energy consumption.

Code obfuscation prevents code piracy; however, code obfuscation has become an important concern about its impact for energy efficiency on mobile application. Therefore, Sahin et al. [55] presented an empirical study of the impact of 18 code obfuscations on energy consumption. The authors' experiment included 15 usage scenarios on 11 Android applications. The experiment results indicated that using code obfuscation likely to increase energy consumption.

Hasan et al. [25] profiled the energy consumption of popular operations performed on data collections (Java List, Map, and Set Abstractions). They showed that using the wrong data collection type according to the profile could lead to an increase in energy consumption that could reach 300%.

Hunt et al. [30] proposed a using lock-free data structure to improve performance, scalability, and energy efficiency. Three different types of lock-free and locking data structures have been implemented to run excessive workloads and compare the execution time and the energy efficiency of each data structure type. Using threads to access a shared data structure requires synchronization of the threads and assurance of data consistency and integrity. However, thread synchronization causes performance problems in multithreaded programs. As a result, the lock-free data showed better performance and higher energy efficiency.

Existing studies provided helpful information about the energy efficiency of platforms and programs and presented new techniques, such as using different types of data structures or algorithms and migrating heavy computation to outsource machines. However, maintaining the code of these platforms and programs is mandatory and can have a positive or negative impact on energy efficiency. Moreover, energy efficiency is beneficial to software developers while maintaining the code and implementing refactoring techniques. Therefore, I propose a study that analyzes the energy efficiency and performance of mobile devices to define the interrelationship between them and understand the cause behind the positive or negative impact, which will provide a guideline to software engineers and developers.

2.2. Defect Analysis

With the widespread presence of traditional software, Chillarege et al. [16] [15] introduced the ODC concept to classify defects and find their root causes. This allows for reaching a short cycle of defect analysis during the software development process.

With the internet and web applications have become worldwide providers of online services, Ma and Tian [40] presented an adaptation of ODC for web errors based on defects found in web logs, and they provided analyses of their results to improve the reliability of web applications.

Cloud computing is the new revolution of web services by providing Software as a Service (SaaS). Alannsary and Tian [2] proposed a defect analysis framework by adapting ODC to SaaS. The new framework considered new characteristics introduced in SaaS, such as multi-tenancy and isolation.

With the lack of defect classifications during black-box testing, Li et al. [35] presented a new defect classification framework called Orthogonal Defect Classification for Black-box Defect (ODC-BD). The proposed framework aims to help black-box defect analyzers and black-box testers, while enhancing the efficiency of the analysis and testing processes.

Wasserman [62] provided an overview of research issues in software engineering for mobile software development. The overview included development processes, tools, user interface design, application portability, quality, and security.

Different approaches inspired by the new types of faults and failures that arise in mobile applications during and after the development process have been discussed. Holl and Elberzhager [28] classified mobile application failures and defined the relationships between their failures and various aspects of their faults. Lelli et al. [34] proposed a model that identifies and classifies GUIs' faults. In their work, they presented an empirical analysis and assessment for their model. However, a classification framework that does not cover all types of mobile application defects hinders in-process feedback from being fast and accurate.

Existing studies provided helpful frameworks to classify traditional software defects in general. However, due to the popularity and high demand of mobile applications, classifying mobile application defects is crucial to software development. Therefore, I propose adapting the ODC concept to mobile environments to classify the new types of defects, improve in-process feedback, and present the results and analyses of applying my framework to bug reports of mobile applications.

2.3. Reliability

Due to the high demand of complex heterogeneous software, software reliability models have become more useful to assessing and predicting the correctness of the software. Lyu [39] presented software reliability models in practice to help researchers and practitioners quantitatively address the characteristics of the SDLC. In addition, these models guide developers and testers to understand and apply software reliability techniques.

Tian et al. [60] evaluated the reliability of web applications after determining their defects and usage. In addition, the possibility of enhancing web application reliability was inspected. The author used the characteristics of web applications as a base to classify web defects. The website workload was measured and characterized at different levels and perspectives, and combined with the failure information about the website to evaluate the operational reliability. The experiment results indicate the efficacy and benefits of the authors' approach.

Many SRGMs for estimating and predicting the reliability of software have been developed and introduced. However, some of these models show inaccurate results, such as delayed S-shape and the exponential type, which indicates that these models may not fit when spending effort that is not constant on testing to detect faults. Therefore, Huang et al. [29] reviewed the logistic testing effort function that can be used to describe the amount of testing effort spent on software testing. In addition, the author proposed how to integrate the logistic testing-effort function into software reliability models. The proposed models show more accurate results compared to the traditional SRGMs.

Software as a Service (SaaS) is a software distribution model that is provided through cloud computing. Alannsary and Tian [3] proposed a method for measuring and predicting the reliability of SaaS. The authors analyzed web server log files to extract failure data. The input domain reliability model was used to measure the operational reliability. SRGMs were used to measure the growth in SaaS reliability.

The Application Programming Interface (API) is an interface, which is used to allow clients and servers to interact. Bokhary and Tian [13] proposed a framework for measuring the reliability of APIs. The authors followed a three-stage approach to collect available failure data, and then the API reliability was estimated. In addition, the authors introduced a case study based on Google Map APIs and showed the effectiveness and success of the proposed framework.

New technologies have been added to mobile phones due to the high demand of end-users. Consequently, the predicted field failure rate has deviated from the actual rate. Therefore, developing new methods for predicting the failure rate before the release has become a challenge for researchers and practitioners. Perera [51] presented a reliability prediction method to overcome the inapplicable traditional reliability prediction methods and deliver more accurate results.

The number of lines of code of software applications in mobile devices has increased to millions. Development organizations must produce predictable fault-free software products. Almering et al. [4] presented an empirical study to assess the reliability of software and validate SRGMs during the integration and test phases. In addition, the capability of the prediction model was compared to predictions by experts. Moreover, obtaining solid reliability assessment and prediction before software release using SRGMs was shown to be possible.

Researchers have proposed studies to assess the software reliability of mobile operating systems and applications. Ivanov et al. [31] presented a comparison between the reliability of three operating systems in mobile environments by applying SRGMs. In addition, Meskini [42] evaluated the reliability of three mobile applications by applying SRGMs to failure data extracted from mobile devices. However, to successfully assess and predict software reliability, the characteristics of mobile environments must be considered. Therefore, to achieve more accurate results in this work, I propose applying software reliability growth models to defect data extracted from bug reports.

Chapter 3

Code Refactoring

As presented in Chapter 1, refactoring techniques are used to optimize software code. Evaluating the impact of these techniques in mobile environments for energy and performance will help developers enhance the quality of the software. In this chapter, Section 3.1 presents the methodology of the study. Section 3.2 explains the experiment preparation. Section 3.3 provides the experiment setup and execution. Section 3.4 presents four case studies. Section 3.5 provides a discussion and analysis of the experiment results. Finally, Section 3.6 presents threats to validity.

3.1. Methodology

The main objective of this study is to profile the positive and negative impact of refactoring techniques on mobile application code using GPS-UP metrics to determine whether the impact on performance and energy consumption caused by refactoring is beneficial. Previous studies measured performance or energy efficiency without finding the interrelationship between them. The goal of finding the interrelationship between performance and energy efficiency is to find the cause of increasing or decreasing energy consumption when refactoring techniques are implemented. Therefore, this study focuses on assessing the positive or negative impact of refactoring on mobile application code. The impact of refactoring techniques on mobile application code has rarely been investigated, and previous experiments ended up just measuring performance or energy consumption for desktop applications. my approach quantitatively evaluates and categorizes refactoring techniques when implemented to applications in mobile environments.

3.2. Experiment Preparation

Following is a description of the experiment preparation, including a list of the selected refactoring techniques, an explanation of GPS-UP metrics and the categories, and the lines of code changed when each refactoring technique is implemented.

3.2.1. Selection of Code Refactoring Techniques

From Fowler's 72 code refactoring techniques, 21 code refactoring techniques were selected. The selection of these techniques was based on two conditions. First, the selected refactoring technique has to change the internal structure of the software code in order to have an effect on the energy efficiency and performance of the mobile application software. Second, there are different groups of refactoring techniques; therefore, at least one refactoring technique was selected from each group.

Below is a list of the selected code refactoring techniques organized in the order they were introduced in Fowler's book:

1. Composing Methods

   (a) Extract Method
   (b) Inline Method
   (c) Inline Temp
   (d) Replace Temp with Query
   (e) Split Temporary Variable
   (f) Remove Assignments to Parameters
   (g) Replace Method with Method Object

2. Moving Features Between Objects
   (a) Move Method
   (b) Move Field
   (c) Extract Class
   (d) Inline Class

3. Organizing Data
   (a) Self Encapsulate Field
   (b) Replace Data Value with Object
   (c) Replace Array with Object
   (d) Replace Type Code with State Strategy

4. Simplifying Conditional Expressions
   (a) Decompose Conditional
   (b) Replace Conditional with Polymorphism

5. Making Method Calls Simpler
    (a) Add Parameter
    (b) Remove Parameter
6. Dealing with Generalization
    (a) Pull Up Field
    (b) Pull Up Method.

3.2.2. Greenup, Powerup, and Speedup Metrics

GPS-UP metrics are used to categorize code refactoring techniques after implementing them to the software code. Each refactoring technique implemented to the software code is categorized into one of the ten categories shown in Fig. 3.1. The original GPS-UP metrics had eight categories. In my work, I added two more categories in response to the experiment results.

The execution time of the mobile application (T) in second(s) and the energy consumption of the mobile application in joule(s) are measured in order to calculate Speedup and Greenup. Speedup indicates the ratio of the non-refactored code runtime to the refactored code runtime. Greenup indicates the ratio of the total energy consumption of the non-refactored code to the total energy consumption of the refactored code. The interrelationship between Speedup and Greenup presents Powerup which is the ratio of the average power consumption:

$$\begin{aligned} EBR &= TBR * PBR\,, \\ EAR &= TAR * PAR\,, \end{aligned} \tag{3.1}$$

where the Power Before Refactoring (PBR) is the average energy consumption before refactoring the software code, and the Power After Refactoring (PAR) is the average energy consumption after refactoring the software code.

$$\text{Speedup(s)} = \frac{TBR}{TAR}\,, \tag{3.2}$$

where the Time Before Refactoring (TBR) is the total runtime before refactoring the application software code, and the Time After Refactoring (TAR) is the total runtime after refactoring the application software code.

$$\text{Greenup(j)} = \frac{EBR}{EAR}\,, \tag{3.3}$$

where the Energy Before Refactoring (EBR) is the total energy consumption before refactoring the application software code, and the Energy After Refactoring (EAR) is the total energy consumption after refactoring the application software code.

$$\text{Powerup} = \frac{\text{Speedup}}{\text{Greenup}}\,, \tag{3.4}$$

if the value of PowerUp is greater than 1, this indicates that the implemented code refactoring technique changed the software code to be less efficient by consuming more energy. If the value of PowerUp is less than 1, this indicates

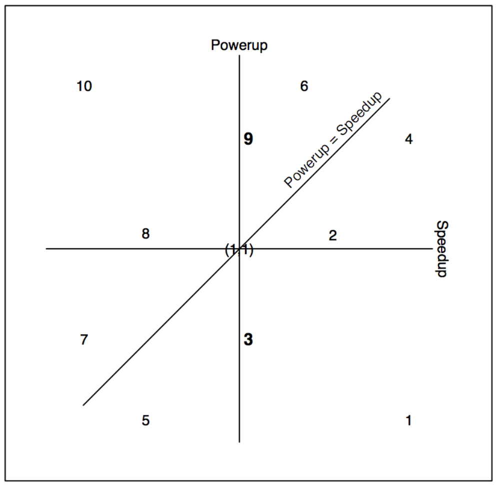


Figure 3.1: GPS-UP application energy efficiency graph.

that the implemented code refactoring technique changed the software code to be more efficient in energy consumption by consuming less energy.

3.2.3. GPS-UP Metrics Categories

There are ten categories in the GPS-UP metrics which are split into two groups. The first group (categories 1 to 5) as a green category group indicates that the refactoring technique increased the energy efficiency (reduced energy consumption). The second group (categories 6 to 10) as a red category group indicates that the refactoring technique decreased the energy efficiency (increased energy consumption). I added two categories to the GPS-UP metrics because some of the refactoring techniques results did not fall in the original eight GPS-UP metrics categories. Table 3.1 shows the mapping of the GPS-UP metrics categories in the original work [1] and the GPS-UP

Table 3.1: GPS-UP Categories Mapping, Where Speedup Is (S) and Powerup Is (P)

Cat. in study	Cat. in ref. [1]	Condition
1	1	P <1 and S >1
2	2	P =1 and S >1
3	Newly added	S =1 and P <1
4	3	P >1, S >1, and S >P
5	4	P <1, S <1, and S >P
6	5	P >1, S >1, and S <P
7	6	P <1, S <1, and S <P
8	7	P =1 and S <1
9	Newly added	S =1 and P >1
10	8	P >1 and S <1

metrics categories in my work.

1. Category 1 is when Powerup < 1 and Speedup > 1. This category indicates that the refactoring technique improved the energy efficiency and performance. The cache memory consumes less energy and is faster than RAM memory. Therefore, the refactoring technique in this category can be used when CPU computation uses more cache memory than RAM memory.

2. Category 2 is when Powerup $= 1$, and Speedup > 1. This category indicates that the refactoring technique improved only the performance and maintained the same level of energy efficiency. The refactoring technique in this category can be used when the performance is more important than the energy efficiency in heavy and linear computations.

3. Category 3 (newly added to the categories) is when Speedup $= 1$ and Powerup < 1. This category indicates that the refactoring technique improved only the energy efficiency and maintained the same level of performance. The refactoring technique in this category can be used when energy efficiency is more important than performance in order to make mobile device batteries last for a longer time.

4. Category 4 is when Powerup > 1, Speedup > 1, and Speedup $>$ Powerup. This category indicates that the refactoring technique slightly improved the performance whereas the level of energy efficiency was decreased. The refactoring technique in this category can be used when performance is needed while the amount of the energy required is not high, such as parallel computations.

5. Category 5 is when Powerup < 1, Speedup < 1, and Speedup $>$ Powerup. This category indicates that the refactoring technique reduced the performance to consume less energy. The refactoring technique in this category can be used when adjusting the performance to achieve better energy efficiency.

6. Category 6 is when Powerup > 1, Speedup > 1, and Speedup $<$ Powerup. This category indicates that the refactoring technique improved the performance; however, the increase in the energy consumption exceeded the improvement in the performance. The refactoring technique in this category can be used when using parallel computations with parallel devices where more energy is consumed, but the performance level does not increase.

7. Category 7 is when Powerup $<$ 1, Speedup $<$ 1, and Speedup $<$ Powerup. This category indicates that the refactoring technique decreased the performance and increased the energy efficiency; however, the more computations the software runs, the more energy that will be consumed, which leads to decreased the energy efficiency.

8. Category 8 is when Powerup = 1, Speedup $<$ 1. This category indicates that the refactoring technique decreased the average of the performance, and the energy efficiency did not improve.

9. Category 9 (newly added to the categories) is when Speedup = 1 and Powerup $>$ 1. This category indicates that the refactoring technique maintained the same level of performance and reduced the energy efficiency, such as when a mobile device performs heavy computation and the battery charge is consumed faster.

10. Category 10 is when Powerup $>$ 1, Speedup $<$ 1. This category indicates that the refactoring technique increased the energy consumption and decreased the performance. The refactoring technique is needed only when the requirement to make the application software code maintainable is more important than the performance and energy consumption.

3.3. Experiment Setup and Execution

Two versions of the application software code (non-refactored and refactored) are compared for each code refactoring technique by using the GPS-

UP metrics in order to show the improvement or decline in performance and energy efficiency.

3.3.1. Experiment Execution Setup

The scenario of the experiment contains seven steps:

1. Develop Fowler's sample code in Android Studio to create a mobile application.
2. Run the application in Android mobile platform.
3. Measure energy consumption and performance.
4. Apply a code refactoring technique to the code.
5. Run the application again.
6. Measure energy consumption and performance.
7. Apply results to GPS-UP Metrics to evaluate and categorize the code refactoring technique.

3.3.2. Experiment Environment

Android Studio was used to develop a mobile application that runs the sample code to measure the code refactoring techniques. The application was installed on a Samsung Galaxy S5 smartphone and an LG Nexus 5X smartphone. The specifications of each mobile device are listed in Table 3.2 and Table 3.3, respectively.

Table 3.2: Samsung Galaxy S5 Specifications

Category	Specifications
Processors	Qualcomm Snapdragon 801 Processor, 2.5GHz Quad-core Krait 400 Adreno 330 GPU
Memory	RAM: 2GB
Storage	Internal: 32GB
Platform	Android 6.0 Marshmallow, TouchWiz UI

Table 3.3: LG Nexus 5X Specifications

Category	Specifications
Processors	Qualcomm Snapdragon 808 Processor, 1.8GHz Hexa-core 64-bit Adreno 418 GPU
Memory	RAM: 2GB
Storage	Internal: 32GB
Platform	Android 6.0 Marshmallow

3.3.3. Energy Measurement Tool

PowerTutor [64] is the application used to measure energy consumption in my study. This application was developed for Android mobile devices to measure and display the power consumed by the device components and any running application. In addition, PowerTutor enables software developers to monitor changes in software energy efficiency while modifying the software design or code. PowerTutor has a power consumption estimation within 5%.

3.4. Case Studies

There were four case studies presented to profile code refactoring techniques, Fowler's sample code, common algorithms code, AnotherMonitor application, and Simple Calculator application. The energy consumption and execution time of the mobile application were measured ten times before implementing each refactoring technique, and ten times after implementing each refactoring technique. After the average, median, and variance of each ten runs were calculated and analyzed, there were no extreme score, and the average value was found as the best value to be applied to the GPS-UP metrics. Then, Greenup, Speedup, and Powerup were calculated in order to categorize each refactoring technique.

3.4.1. Flower's Sample Code

I developed Fowler's sample code into a mobile application and implemented the chosen 21 refactoring techniques to the software code to measure the energy consumption and performance as in [50] [18] [54]. The results were applied to the GPS-UP metrics to categorize the refcatoring techniques. Table 3.4 and Table 3.8 illustrates the results of the Samsung Galaxy S5 and LG Nexus 5X respectively.

3.4.2. Common Algorithms Code

In this experiment, I implement a mobile application with a code that has a common data structure (linked list) and two algorithms (quick sort and

Table 3.4: GPS-UP Metrics Categories for 21 Code Refactoring Techniques for Fowler's sample code in the Samsung Galaxy S5, Where (s) Is Second(s) and (j) Is Joule(s)

Samsung Galaxy S5	AVG of 10 Runs		AVG of 10 Runs		GPS-UP Metrics				LOC	
Code refactoring techniques	TBR(s)	EBR(j)	TAR(s)	EAR(j)	Greenup	Speedup	Powerup	Category	Total	Changed
Inline Method	62.36	11.37	59.46	9.91	1.15	1.05	0.91	C1	264	4
Move Method	61.43	11.06	58.47	8.80	1.26	1.05	0.84	C1	288	16
Inline Class	63.38	12.78	58.50	9.56	1.34	1.08	0.81	C1	298	5
Remove Parameter	60.67	10.00	58.50	9.62	1.04	1.04	1.00	C2	360	6
Pull Up Field	60.59	10.71	60.45	10.50	1.02	1.00	0.98	C3	354	10
Pull Up Method	60.61	10.55	60.50	10.40	1.01	1.00	0.99	C3	354	5
Inline Temp	60.55	10.84	58.51	10.63	1.02	1.03	1.01	C4	264	2
Remove Assignments to Parameters	61.44	10.97	59.45	10.69	1.03	1.03	1.01	C4	272	3
Replace Type Code with State Strategy	62.60	10.46	60.43	10.39	1.01	1.04	1.03	C4	320	20
Replace Method with Method Object	60.65	10.69	59.44	11.86	0.90	1.02	1.13	C6	274	15
Move Field	61.72	10.73	61.30	11.70	0.92	1.01	1.10	C6	288	10
Replace Array with Object	62.41	11.04	59.98	11.74	0.94	1.04	1.11	C6	306	19
Extract Class	60.28	12.28	62.84	12.39	0.99	0.96	0.97	C7	298	20
Replace Data Value with Object	62.00	11.44	63.12	11.63	0.98	0.98	1.00	C8	296	10
Decompose Conditional	61.13	11.13	60.94	12.12	0.92	1.00	1.09	C9	335	6
Replace Temp with Query	62.37	11.30	62.59	11.70	0.97	1.00	1.03	C9	265	7
Split Temporary Variable	60.16	10.68	60.38	12.09	0.88	1.00	1.13	C9	270	4
Extract Method	61.14	11.10	62.13	11.84	0.94	0.98	1.05	C10	263	7
Self Encapsulate Field	61.42	12.54	63.10	14.18	0.88	0.97	1.10	C10	293	3
Replace Conditional with Polymorphism	63.21	10.17	64.13	12.91	0.79	0.99	1.25	C10	340	17
Add Parameter	59.15	9.85	61.02	11.10	0.89	0.97	1.09	C10	354	6

Table 3.5: GPS-UP Metrics Categories for 10 Code Refactoring Techniques for common algorithms code in the Samsung Galaxy S5, Where (s) Is Second(s) and (j) Is Joule(s)

Samsung Galaxy S5 - Common Algorithms	AVG of 10 Runs		AVG of 10 Runs		GPS-UP Metrics				LOC	
Code refactoring techniques	TBR(s)	EBR(j)	TAR(s)	EAR(j)	Greenup	Speedup	Powerup	Category	Total	Changed
Inline Method	88.49	28.86	86.19	27.16	1.06	1.03	0.97	C1	310	8
Move Method	89.01	29.35	86.52	28.17	1.04	1.03	0.99	C1	310	20
Remove Parameter	89.07	28.82	88.28	28.53	1.01	1.01	1.00	C2	310	8
Inline Temp	88.88	28.77	87.52	28.48	1.01	1.02	1.01	C4	310	5
Replace Array with Object	88.66	27.99	87.26	29.84	0.94	1.02	1.08	C6	310	25
Decompose Conditional	88.43	29.07	89.48	29.35	0.99	0.99	1.00	C8	310	10
Replace Temp with Query	88.57	28.30	88.79	29.61	0.96	1.00	1.04	C9	310	11
Split Temporary Variable	88.38	28.39	88.48	29.67	0.96	1.00	1.04	C9	310	5
Extract Method	87.54	27.74	88.82	29.09	0.95	0.99	1.03	C10	310	9
Add Parameter	87.32	28.36	88.78	29.20	0.97	0.98	1.01	C10	310	6

Table 3.6: GPS-UP Metrics Categories for 6 Code Refactoring Techniques for AnotherMonitor Application in the Samsung Galaxy S5, Where (s) Is Second(s) and (j) Is Joule(s)

Samsung Galaxy S5 - AnotherMonitor	AVG of 10 Runs		AVG of 10 Runs		GPS-UP Metrics				LOC	
Code refactoring techniques	TBR(s)	EBR(j)	TAR(s)	EAR(j)	Greenup	Speedup	Powerup	Category	Total	Changed
Inline Method	87.54	17.42	84.47	15.53	1.12	1.04	0.92	C1	2394	70
Inline Temp	87.57	17.44	86.55	17.29	1.01	1.01	1.00	C2	2394	15
Replace Array with Object	87.56	17.49	85.52	18.49	0.95	1.02	1.08	C6	2394	60
Extract Method	87.59	17.44	87.78	18.55	0.94	1.00	1.06	C9	2394	75
Decompose Conditional	87.32	17.47	88.96	18.47	0.95	0.98	1.04	C10	2394	60
Split Temporary Variable	87.47	17.52	88.57	18.44	0.95	0.99	1.04	C10	2394	25

Table 3.7: GPS-UP Metrics Categories for 6 Code Refactoring Techniques for Simple Calculator Application in the Samsung Galaxy S5, Where (s) Is Second(s) and (j) Is Joule(s)

Samsung Galaxy S5 - Simple Calculator	AVG of 10 Runs		AVG of 10 Runs		GPS-UP Metrics				LOC	
Code refactoring techniques	TBR(s)	EBR(j)	TAR(s)	EAR(j)	Greenup	Speedup	Powerup	Category	Total	Changed
Inline Method	62.21	11.00	61.52	9.98	1.10	1.01	0.92	C1	700	21
Inline Temp	62.34	10.56	61.52	10.43	1.01	1.01	1.00	C2	700	10
Replace Array with Object	61.78	10.33	60.16	11.00	0.94	1.03	1.09	C6	700	15
Extract Method	62.82	10.39	62.99	11.28	0.92	1.00	1.08	C9	700	35
Decompose Conditional	62.76	11.81	63.25	12.11	0.98	0.99	1.02	C10	700	29
Split Temporary Variable	61.16	11.32	62.33	12.02	0.94	0.98	1.04	C10	700	17

Table 3.8: GPS-UP Metrics Categories for 21 Code Refactoring Techniques for Fowler's sample code in the LG Nexus 5X, Where (s) Is Second(s) and (j) Is Joule(s)

LG Nexus 5X	AVG of 10 Runs		AVG of 10 Runs		GPS-UP Metrics				LOC	
Code refactoring techniques	TBR(s)	EBR(j)	TAR(s)	EAR(j)	Greenup	Speedup	Powerup	Category	Total	Changed
Inline Method	67.68	13.20	66.54	12.70	1.04	1.02	0.98	C1	264	4
Move Method	67.81	12.39	63.94	10.56	1.17	1.06	0.90	C1	288	16
Inline Class	66.18	13.18	62.71	11.03	1.19	1.06	0.88	C1	298	5
Remove Parameter	62.67	13.22	61.41	12.82	1.03	1.02	0.99	C1	360	6
Pull Up Field	65.82	12.41	64.63	11.63	1.07	1.02	0.95	C1	354	10
Pull Up Method	65.50	13.27	65.37	12.39	1.07	1.00	0.94	C3	354	5
Inline Temp	66.85	13.17	63.99	12.96	1.02	1.04	1.03	C4	264	2
Remove Assignments to Parameters	66.78	13.13	68.57	12.12	1.08	0.97	0.90	C5	272	3
Replace Type Code with State Strategy	66.67	13.76	68.14	12.64	1.09	0.98	0.90	C5	320	20
Replace Method with Method Object	68.28	13.35	66.30	14.69	0.91	1.03	1.13	C6	274	15
Move Field	64.30	13.73	62.80	14.42	0.95	1.02	1.08	C6	288	10
Extract Class	64.79	13.78	63.49	14.24	0.97	1.02	1.05	C6	298	20
Replace Array with Object	66.77	13.29	64.35	13.88	0.96	1.04	1.08	C6	306	19
Decompose Conditional	66.17	12.96	66.24	14.40	0.90	1.00	1.11	C9	335	6
Extract Method	64.51	12.74	68.13	14.82	0.86	0.95	1.10	C10	263	7
Replace Temp with Query	66.36	14.33	68.11	15.28	0.94	0.97	1.04	C10	265	7
Split Temporary Variable	67.36	13.12	68.87	14.39	0.91	0.98	1.07	C10	270	4
Replace Data Value with Object	63.65	12.17	66.25	14.10	0.86	0.96	1.11	C10	296	10
Self Encapsulate Field	64.46	12.62	66.91	14.21	0.89	0.96	1.08	C10	293	3
Replace Conditional with Polymorphism	65.55	13.42	66.89	14.55	0.92	0.98	1.06	C10	340	17
Add Parameter	63.49	12.51	64.98	13.20	0.95	0.98	1.03	C10	354	6

Table 3.9: GPS-UP Metrics Categories for 10 Code Refactoring Techniques for common algorithms code in the LG Nexus 5X, Where (s) Is Second(s) and (j) Is Joule(s)

LG Nexus 5X - Common Algorithms	AVG of 10 Runs		AVG of 10 Runs		GPS-UP Metrics				LOC	
Code refactoring techniques	TBR(s)	EBR(j)	TAR(s)	EAR(j)	Greenup	Speedup	Powerup	Category	Total	Changed
Inline Method	89.17	29.02	85.80	26.13	1.11	1.04	0.94	C1	310	8
Move Method	88.34	29.31	86.99	27.45	1.07	1.02	0.95	C1	310	20
Remove Parameter	88.86	29.02	87.86	28.83	1.01	1.01	1.00	C2	310	8
Inline Temp	88.25	29.18	86.65	28.89	1.01	1.02	1.01	C4	310	5
Replace Array with Object	89.00	28.99	87.09	30.19	0.96	1.02	1.06	C6	310	25
Decompose Conditional	88.80	28.87	89.76	29.04	0.99	0.99	1.00	C8	310	10
Replace Temp with Query	88.79	29.05	88.95	31.03	0.94	1.00	1.07	C9	310	11
Split Temporary Variable	88.71	29.63	88.84	30.84	0.96	1.00	1.04	C9	310	5
Extract Method	88.87	28.85	91.05	30.79	0.94	0.98	1.04	C10	310	9
Add Parameter	88.92	28.42	90.29	30.88	0.92	0.98	1.07	C10	310	6

Table 3.10: GPS-UP Metrics Categories for 6 Code Refactoring Techniques for AnotherMonitor Application in the LG Nexus 5X, Where (s) Is Second(s) and (j) Is Joule(s)

LG Nexus 5X - AnotherMonitor	AVG of 10 Runs		AVG of 10 Runs		GPS-UP Metrics				LOC	
Code refactoring techniques	TBR(s)	EBR(j)	TAR(s)	EAR(j)	Greenup	Speedup	Powerup	Category	Total	Changed
Inline Method	89.14	18.45	85.51	17.68	1.04	1.04	1.00	C2	2394	70
Inline Temp	89.45	18.55	85.68	18.35	1.01	1.04	1.03	C4	2394	15
Replace Array with Object	87.43	19.09	89.29	19.31	0.99	0.98	0.99	C7	2394	60
Extract Method	88.87	18.64	88.91	18.74	0.99	1.00	1.01	C9	2394	75
Decompose Conditional	88.50	18.30	89.59	19.10	0.96	0.99	1.03	C10	2394	60
Split Temporary Variable	87.93	17.98	89.14	18.79	0.96	0.99	1.03	C10	2394	25

Table 3.11: GPS-UP Metrics Categories for 6 Code Refactoring Techniques for Simple Calculator Application in the LG Nexus 5X, Where (s) Is Second(s) and (j) Is Joule(s)

LG Nexus 5X - Simple Calculator	AVG of 10 Runs		AVG of 10 Runs		GPS-UP Metrics				LOC	
Code refactoring techniques	TBR(s)	EBR(j)	TAR(s)	EAR(j)	Greenup	Speedup	Powerup	Category	Total	Changed
Inline Method	63.67	10.37	62.28	10.10	1.03	1.02	1.00	C2	700	21
Inline Temp	63.04	10.17	61.71	10.03	1.01	1.02	1.01	C4	700	10
Replace Array with Object	62.99	10.91	64.13	10.99	0.99	0.98	0.99	C7	700	15
Extract Method	62.75	10.03	62.86	11.28	0.89	1.00	1.12	C9	700	35
Decompose Conditional	63.21	11.09	63.68	11.82	0.94	0.99	1.06	C10	700	29
Split Temporary Variable	63.22	11.26	63.84	13.11	0.86	0.99	1.15	C10	700	17

binary search) in Java code [14] [65] [53] [25]. The linked list contains 4,500 objects. Ten code refactoring techniques are applicable to the application code. After we implemented each refactoring technique to the code, results were applied to the GPS-UP metrics to categorize each refactoring technique. Table 3.5 and Table 3.9 illustrate the results of the Samsung Galaxy S5 and LG Nexus 5X, respectively.

3.4.3. Two Open-Source Applications

To generalize our experiment results, we chose two commonly used applications from F-Droid [36] software repository to measure the impact of 6 refactoring techniques on energy consumption and performance in a real open-source mobile application as in [43] [44] [24] [20] [5].

AnotherMonitor [22] is a mobile application that monitors and records CPU utilization and memory usage. It generates graphic results in 0.5, 1, 2 and 4 second intervals. To perform the experiment and measure the application energy consumption and performance, the interval time was disabled and swapped with a loop that has 20k iterations. Six code refactoring techniques out of the 21 were applicable to the mobile application code. After implementing each refactoring technique to the code, the results were applied to the GPS-UP metrics to categorize the used refactoring technique. Table 3.6 and Table 3.10 illustrate the results of the Samsung Galaxy S5 and LG Nexus 5X, respectively.

Simple Calculator [33] is a mobile application that performs simple mathematical functions. Decimal numbers were injected as an input to the application code, to eliminate the human factor and measure the application energy consumption and performance. Six code refactoring techniques out of the 21 were applicable to the mobile application code. After we implemented each refactoring technique to the code, the results were applied to the GPS-UP metrics to categorize each refactoring technique. Table 3.7 and Table 3.11 illustrate the results of the Samsung Galaxy S5 and LG Nexus 5X, respectively.

3.5. Discussion

Following is an explanation of the technical reasons behind the positive or negative improvement for each refactoring technique shown in the result of my case studies.

3.5.1. Green Categories

The refactoring techniques that fell in this green area improved performance or energy efficiency or both together. The Inline Method technique replaced the method call with its body which eliminates the fetch-decode-execute cycle to call the method. The Move Method technique reduced the cost of the queries between two classes by moving the method to the class that has more features with it. The Inline Class technique deleted the class that is not needed very often and moved its work to another class.

The Remove Parameter technique deleted the parameter that is no longer needed in the method. As a result, the recent two refactoring techniques eliminated the extra load in the memory. The Pull Up Field technique moved the field that the two subclasses had into the upper class which reduced the cost of duplicate parameters.

The Pull Up Method technique which improved only the energy efficiency and maintained the same level of performance. The refactoring technique moved the same method that the two subclasses had into the upper class which reduced the energy consumption to allocate one method in the memory instead of two duplicate methods.

The Inline Temp technique improved the performance more than the energy efficiency. Deleting temporary variables reduced the time for fetching the unnecessary temporary variable from the main and cache memories. As SpeedUp is greater than PowerUp, the Inline Temp technique is still considered a positive improvement to the energy efficiency.

The following refactoring techniques did not improve performance and maintained the same level of energy efficiency which is considered a positive improvement. The Remove Assignment to Parameter technique assigned a value to a local variable instead of a parameter which improved maintainability and readability. The Replace Type Code With State Strategy technique changed the behavior of the class by adding subclasses for the type of object which made the code updateable. In addition, because there was no change in performance and energy efficiency, these refactoring techniques are still considered green categories.

Table 3.12: Comparison of GPS-UP Metrics Categories for Code Refactoring Techniques in the Three Case Studies, Where (-) Is Not Applicable

	Categories							
	Fowler's Sample		Common Algorithms		AnotherMonitor		Simple Calculator	
Code refactoring techniques	Samsung Galaxy 5S	LG Nexus 5X	Samsung Galaxy 5S	LG Nexus 5X	Samsung Galaxy 5S	LG Nexus 5X	Samsung Galaxy 5S	LG Nexus 5X
Inline Method	C1	C1	C1	C1	C1	C2	C1	C2
Move Method	C1	C1	C1	C1	-	-	-	-
Inline Class	C1	C1	-	-	-	-	-	-
Remove Parameter	C2	C1	C2	C2	-	-	-	-
Pull Up Field	C3	C1	-	-	-	-	-	-
Pull Up Method	C3	C3	-	-	-	-	-	-
Inline Temp	C4	C4	C4	C4	C2	C4	C2	C4
Remove Assignments to Parameters	C4	C5	-	-	-	-	-	-
Replace Type Code with State Strategy	C4	C5	-	-	-	-	-	-
Replace Method with Method Object	C6	C6	-	-	-	-	-	-
Move Field	C6	C6	-	-	-	-	-	-
Extract Class	C7	C6	-	-	-	-	-	-
Replace Array with Object	C6	C6	C6	C6	C6	C7	C6	C7
Decompose Conditional	C9	C9	C8	C8	C10	C10	C10	C10
Extract Method	C10	C10	C10	C10	C9	C9	C9	C9
Replace Temp with Query	C9	C10	C9	C9	-	-	-	-
Split Temporary Variable	C9	C10	C9	C9	C10	C10	C10	C10
Replace Data Value with Object	C8	C10	-	-	-	-	-	-
Self Encapsulate Field	C10	C10	-	-	-	-	-	-
Replace Conditional with Polymorphism	C10	C10	-	-	-	-	-	-
Add Parameter	C10	C10	C10	C10	-	-	-	-

3.5.2. Red Categories

The refactoring techniques that consumed more energy although several refactoring techniques improved performance. When the method is long and the Extract Method technique cannot be implemented to the code, The Replace Method with Method Object technique is the solution to turn the method into an object with its attributes in order to split the long method into short methods and that makes the code better organized and faster. However, the code consumed more energy because of the extra computing and allocation of the extra attributes and methods.

The Move Field technique moved the field to the class that uses the field more than its original class; therefore, the technique improved performance only by reducing unnecessary queries from two classes to the field. The Extract Class technique split the class that did the work of two classes which improved performance; however, this technique did not improve energy efficiency because of the increase in the number of classes in the memory.

The Replace Array With Object technique changed the array that had different types of elements into an object and the different types of elements into the object's attributes which made the code more understandable and faster over the cost of consuming more energy for accessing the object's attributes instead of the array's elements.

The Decompose Conditional technique replaced each part of a complicated condition if-then-else into a method which is more readable and understandable; however, calling the created methods costs more energy. The

Extract Method technique extracted part of the long method to be in a new separate method. As a result, the refactoring technique downgraded the performance and energy efficiency for calling the new method. However, this technique is still beneficial because the new method can be called by other methods.

The Replace Temp With Query technique replaced the temporary variable and its references with a query from a method which made the CPU execute the method every time there is reference to the temporary variable. The positive improvement is that the created query and its method can be used by other methods. The Split Temporary Variable technique replaced a temporary variable that is assigned to two values with two temporary variables which made the code more understandable. The price was sacrificing performance and energy by loading two variables instead of one to the memory.

The Self Encapsulate Field technique added a Get Method to access the field instead of directly accessing the field in order to achieve encapsulation. However, the Set and Get methods are extra, which lead to consuming more time and energy to execute. The Replace Data Value with Object technique replaced the data attribute that needed more data or behavior with an object which means creating a new class for the object; therefore, creating a new class made the CPU and RAM slower and consumed more energy.

The Replace Conditional with Polymorphism technique replaced the condition that chooses different tracks and different objects with a subclass and method for each object; similarly, adding classes leads to the same result for performance and energy efficiency. The last refactoring technique, the Add Parameter technique added a parameter to the method that needed more information which means more energy is needed to access this parameter from the RAM. These refactoring techniques did not improve performance or energy efficiency despite being useful for reaching maintainability.

3.5.3. Analysis and Comparison

The four case studies are compared next to each other in Table 3.12 based on GPS-UP metrics categories for refactoring techniques. In Fowler's Sample, the categories of the 21 refactoring techniques are slightly different in the two mobile environments. However, the 21 refactoring techniques fell within the same green or red area of the GPS-UP metrics. In common algorithms, the 10 refactoring techniques fell exactly in the same GPS-UP metrics categories in both mobile environments (Samsung Galaxy 5S and LG Nexus 5X) because the code is very simple and only has one Java class. In Another Mobile and Simple Calculator, the categories of the 6 refactoring techniques fell in the same GPS-UP metrics categories in each mobile environment. However, within the same application, different mobile environments led to slightly different GPS-UP metrics categories within the same green or red area. However, in all case studies, each refactoring technique had the same impact on energy consumption and performance in each mobile application code and environment.

3.6. Threats to Validity

The main threat to the validity of my results is the possibility of getting different results when testing refactoring techniques in different mobile operating systems and devices; moreover, it will prevent us from generalizing my experiment results. Thus, testing these techniques in different environments will minimize this threat.

Another threat to validity is that I tested only Fowler's refactoring techniques in Fowler's sample code; therefore, I need to test these refactoring techniques in real-world mobile applications in order to get additional experiment results and better insights.

Chapter 4

Defect Analysis

As explored in Chapter 2, adapting the ODC model to classify mobile application defects will improve in-process feedback and finding the root cause. In this chapter, Section 4.1 explains the proposed method to adapt ODC to mobile environments. Section 4.2 presents a case study. Section 4.3 provides an analysis of results and discussion.

4.1. New Methodology

When mobile software development started, mobile applications were small with only a few thousand lines of code compared to traditional software. Since then, mobile software development has grown exponentially and has become more complex. Software engineering and software quality were applied to mobile software development to ensure its accuracy. This introduced new requirements that need to be considered during the development process, such as power consumption, interacting with other mobile applications and environments, screen and sensor handling, and limitations of mobile devices.

The characteristics of mobile application environments were examined and studied to identify the differences between defects of traditional software and those of mobile applications. In the original ODC, defect types were characterized as *function, interface, assignment, checking, timing/serializa-*

Table 4.1: Adapted ODC for Mobile Environments

Type	Description	Process Associations
Function	Capability, product interface, interface with hardware or global data structures (requires a formal design change)	Design
Interface	Errors from interactions with other components, modules or device drivers via macros, call statements, control blocks, or parameter lists.	LLD
Checking	Program logic that failed to validate data and values before they are used	LLD or Code
Assignment	A few lines of code errors, such as initialization of control blocks or data structure	Code
Timing/Serialization	Real-time resources	LLD
Build/Package/Merge	Library systems, management of changes, or version control	Library Tools
Documentation	Publication and maintenance	Publications
Algorithm	Errors include efficiency or correctness problems of tasks or data structures	LLD
Energy	**Exhausting battery energy through excessive usage of mobile device components such as CPU, memory, sensors, etc.**	**LLD**
Network	**Improper handling of network connections**	**Code or LLD**
Incompatibility	**Mobile application is not customized for the mobile device, operating system, interactions with other applications, or supporting mobile features**	**LLD**
GUI	**Errors generated from end-user interface or screen changes**	**Design, LLD, or Code**
Interruption	**Improper handling of receiving calls or messages, activating screen saver state, or switching between applications**	**Components or Code**
Notification	**Errors from application alerts, including sound, vibration, visual, and text**	**LLD or Code**

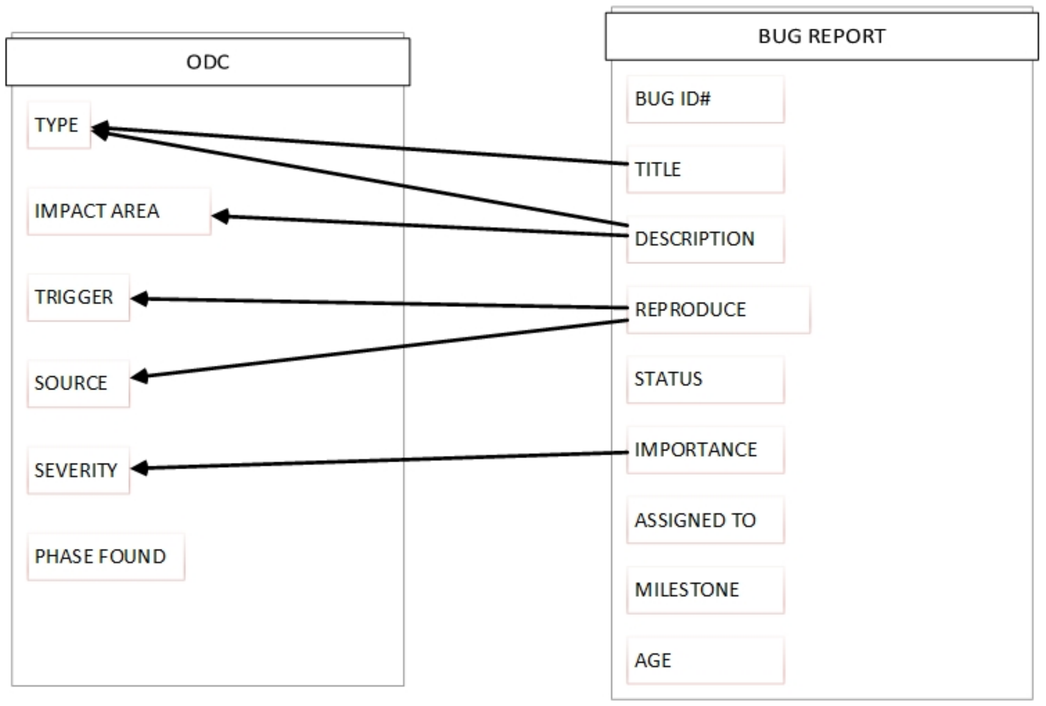


Figure 4.1: Mapping of Mobile Application Defect Report components to ODC components

tion, build/package/merge, and *algorithm*. However, there are new defects that cannot be classified using the original ODC. Therefore, the proposed framework caters for mobile application defects. As shown in bold in Table 4.1, I added the following defect types to the original ODC. A brief description for each defect type is provided below:

1. ***Energy***: Mobile devices consume energy from batteries to run, whereas PCs use power. Due to excessive runs of application code, defects start arising by consuming more energy.

2. ***Network***: In mobile environments, networks change rapidly. Due to improper handling of these changes, mobile applications produce defects and cause crashing or freezing.

3. ***Incompatibility***: Due to the variety of mobile devices and their operating systems, mobile environments do not support some mobile applications or their features; therefore, defects are produced from lacking to consider mobile environment limitations.

4. ***GUI***: There is a strong relationship between user interface and mobile screen properties such as size, touch, landscape, color, brightness, etc. Since the user interface is essential when working with mobile applications, in my proposed framework, I extracted the user-interface subcategory from the *function* defect type and created a new defect type called *GUI*.

5. ***Interruption***: Mobile devices get interrupted by calls, messages, alerts, etc. Improper handling of these interruption may cause defects that make mobile applications stop responding.

6. ***Notification***: Many different types of mobile notifications may produce defects by giving false notifications (time, place, or content).

To properly classify defects using bug reports, components of these reports (title, description, reproduce, and importance) are mapped to the components of the ODC model (type, impact area, trigger, source, and severity) as illustrated in Figure 4.1. This enables performing one-way, two-way, and multi-way analyses.

4.2. Case Study

Canonical has developed a website and a web application called Launchpad [37] to allow open-source software to be developed and maintained by end-user developers. Launchpad has 42,947 projects and 1,779,680 bug reports. I chose two popular mobile applications from Launchpad. First, the Tomdroid [9] is a note-taking application that has a unique wiki-style display in the Android platform. In addition, Tomdroid has a format that enables syncing notes with the Tomboy application [23]. Second, the Telegram [26] is a cloud-based instant messaging and voice-over IP service. The reporting bug site categorizes bugs by severity (critical, high, medium, low, wishlist, and undecided). Tomdroid had 220 bug reports [10], and Telegram had 246 bug reports [27]. In my study, I only considered bugs that had severity levels from *critical* to *low*. Wishlist bugs were discarded since they were not considered defects. Undecided bugs were ignored due to they were not categorized. Therefore, 131 defects for Tomdroid and 68 defects for Telegram were applicable for my case study. Furthermore, each bug was manually studied and classified based on one of the defect types available in the proposed ODC framework. This classification process was preformed by three individuals with computer science backgrounds specifically in software engineering.

4.3. Analyses of the Results and a Discussion

Classifying defects of mobile applications provides one-way, two-way, and multi-way analyses. I analyzed the descriptions and importance of the bug

Table 4.2: One-Way Analysis Based On Defect Severity

Severity	Tomdroid # of Defects	Telegram # of Defects
Critical	5	5
High	52	21
Medium	36	33
Low	38	9

Table 4.3: One-Way Analysis Based On Defect Type

Type	Tomdroid # of Defects	Telegram # of Defects
Function	6	14
Interface	3	1
Checking	-	-
Assignment	47	17
Timing/Serialization	11	-
Build/Package/Release	1	1
Documentation	-	-
Algorithm	3	2
Energy	-	2
Network	-	5
Incompatibility	29	5
GUI	26	10
Interruption	2	1
Notification	3	10

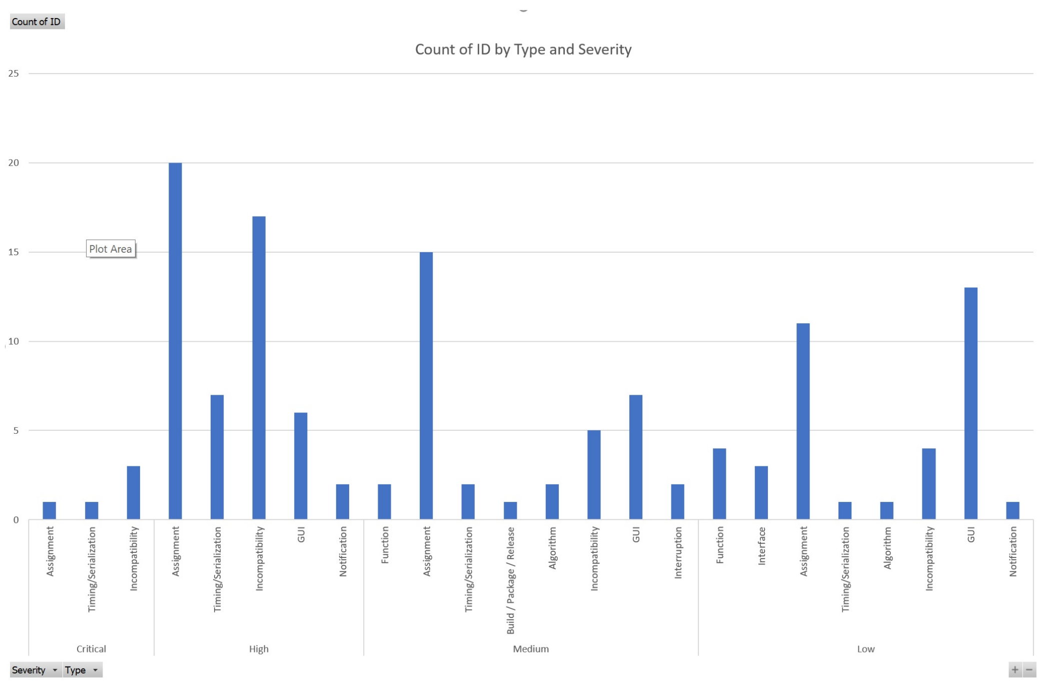


Figure 4.2: Tomdroid: Two-Way Analysis Based On Defect Severity and Type

reports for the selected mobile applications to identify the *severity* level and *type* for each defect. In addition, I performed one-way analysis by examining one of the proposed ODC attributes, such as *type* and/or *severity*, in order to focus on areas where defects were highly dense. Moreover, I performed two-way analysis by examining the intersection between *type* and *severity* to explore areas that were not covered in the one-way analysis.

Table 4.4: Tomdroid: Two-Way Analysis Based On Defect Severity and Type

Severity	Type	# of Defects
Critical	Assignment	1
	Timing/Serialization	1
	Incompatibility	3
High	Assignment	20
	Timing/Serialization	7
	Incompatibility	17
	GUI	6
	Notification	2
Medium	Function	2
	Assignment	15
	Timing/Serialization	2
	Build/Package/Release	1
	Algorithm	2
	Incompatibility	5
	GUI	7
	Interruption	2
Low	Function	4
	Interface	3
	Assignment	11
	Timing/Serialization	1
	Algorithm	1
	Incompatibility	4
	GUI	13
	Notification	1

Table 4.5: Telegram: Two-Way Analysis Based On Defect Severity and Type

Severity	Type	# of Defects
Critical	Network	2
	Incompatibility	2
	Notification	1
High	Function	1
	Assignment	4
	Algorithm	1
	Energy	1
	Network	2
	Incompatibility	3
	GUI	5
	Interruption	1
	Notification	3
Medium	Function	11
	Interface	1
	Assignment	9
	Algorithm	1
	Energy	1
	Network	1
	GUI	3
	Notification	6
Low	Function	2
	Assignment	4
	Build/Package/Release	1
	GUI	2

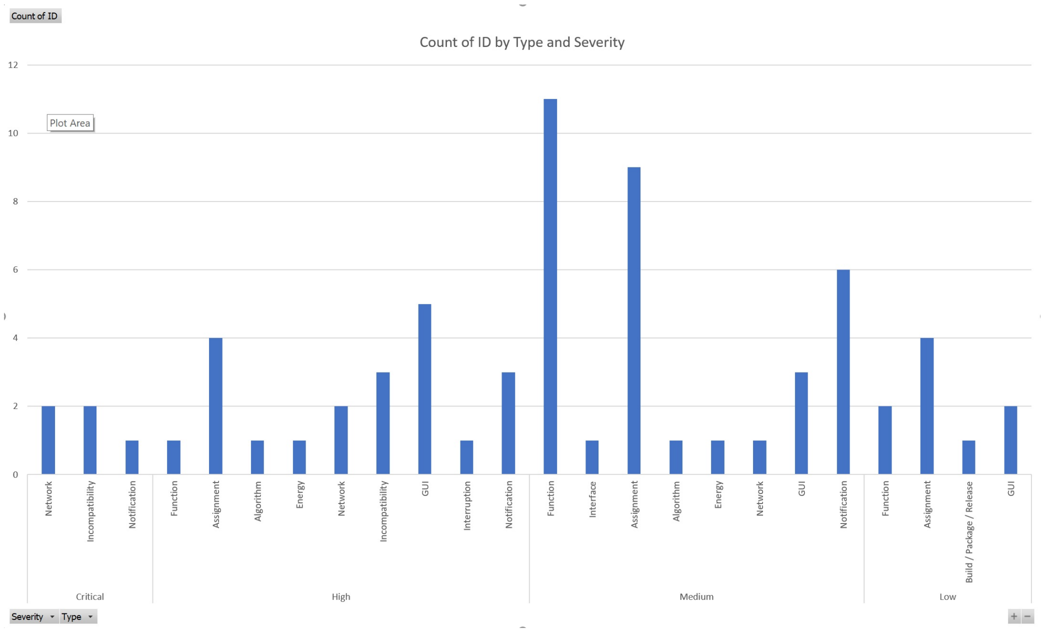


Figure 4.3: Telegram: Two-Way Analysis Based On Defect Severity and Type

4.3.1. One-Way Analysis

One-way analysis can be done by analyzing one attribute of the ODC framework at a time. The *severity* and *type* defect attributes are used to find those areas with high defect rates that need attention to reduce the number of defects and eliminate their root causes.

Table 4.2 illustrates the classification of defects by severity level for both mobile applications. Most defects in the Tomdroid application were produced with *high severity*, indicating that developers should focus on these defects and find their root causes. However, no red flags were raised for the Telegram application since most defects were classified as *medium severity*.

Table 4.3 illustrates the classification of defects by the *type* attribute. For both applications, most defects were produced for the *assignment type*, raising a red flag about the level of the programmers' skills and experience. For Tomdroid, the second most found defects were produced for the *incompatibility* and *GUI* defect *types*, indicating that developers need to be aware of the application's interactions with different systems, and improve the handling of the user interface and device screen. For Telegram, The *function* defect *type* showed that the software development stages-analysis and design-were producing the second highest number of defects in the application. In addition, the *GUI* and *notification* defect *types* also had high numbers of defects, indicating that the handling of the application notifications need to be improved, and the user interface and screen states need to be properly designed and adjusted to the nature of the mobile environment.

4.3.2. Two-Way Analysis

Two-way analysis can be done by analyzing the intersection between two of the ODC attributes, allowing the information that is related to the defects to be displayed, identify their root causes, and find the best solutions. In this case study, I analyzed the intersection between the *type* and *severity* defect attributes.

Table 4.4 and Figure 4.2 illustrate that the high rate of defects was concentrated in the intersection between the attribute severity: *high* and the types: *assignment* and *incompatibility.* This confirms that programmers are injecting the code with faults as shown in the one-way analysis, which may indicate their lack of programming experience or skills. This might also suggest improving the code testing stage before the mobile application release.

Table 4.5 and Figure 4.3 illustrate that the high rate of defects was concentrated in the intersection between the attribute severity: *medium* and the types: *function*, *assignment*, and *notification.* Since the severity was *medium*, this does not raise much concern. However, the *types function* and *notification* indicate that the analysis and design stages need to be performed with more consideration of the mobile environment's characteristics and nature.

4.3.3. Discussion

This paper demonstrated adapting the ODC concept to mobile environments to confirm its feasibility. Defects generated from mobile environments and related to new types, such as *energy* and *notification*, can be classified by the original ODC. However, it will provide misleading information related to the defect's origin and may lead to not exposing the root cause, which can prevent finding the best solution. Exposing mobile application defects by implementing the proposed ODC framework will benefit developers in acquiring short and accurate in-process feedback. In addition, classifying the two mobile application defects proved that the proposed ODC adaptation is significantly effective for defect classification and resolution, specifically when implementing one-way and two-way analyses.

Chapter 5

Reliability

5.1. Methodology

The evolution of mobile software development started with the development of applications for mobile devices with a few thousand lines of code. Today, mobile applications have become more complex due to the high demand from end-users. Software engineering and software quality are involved to ensure the accuracy of the new functionalities and features of mobile applications. This introduces new requirements which need to be considered to improve the stability and reliability of mobile applications.

The nature of mobile environments is different from that of PC and server environments. In addition, mobile application developers rarely share application defect data generated during the testing phase. Therefore, due to the lack of failure data for mobile applications, I propose using bug reports to analyze defect data and measure, assess, and predict application reliability. The proposed method consists of the following:

- Phase 1: Extract mobile application defects from the bug report repository.
- Phase 2: Analyze the bug reports found to discard those that are not

related to software reliability, such as defects that originate from the mobile operating system or hardware.

- Phase 3: Weigh each bug report based on its classification as shown in Table 5.1.

Table 5.1: Suggested Bug Report Weight

Importance	***Weight***
Critical High Medium Low	1
Wishlist	0
Undecided	Defect validity ratio

- Phase 4: Relate the date of each bug report to the total number of days since the release day of the mobile application.

- Phase 5: Assess and evaluate the reliability of the selected mobile application and predict its future reliability using SRGMs.

- Phase 6: Use the purification rate and the standard error of the estimate for assurance.

5.2. Case Study

Measuring and predicting reliability in mobile applications are carried out through the testing stage, collecting defect data from the bug report repository, and applying the most commonly used SRGMs. In this chapter, I selected an open-source mobile application as a case study because its failure data is available for developers and end-users to present the adequacy of the selected SRGMs in mobile applications.

Table 5.2: Weighted Bug Reports: Version 1

Importance	***Number of bug reports***	***Weight***
Critical	2	1
High	13	1
Medium	22	1
Low	9	1
Wishlist	25	0
Undecided	43	0.65

5.2.1. Defect Data Set

The bug reports are extracted from the bug report repository of Launchpad [38]. Launchpad classifies bug reports based on importance and status. For importance, the bug reports are classified as *critical, high, medium, low, wishlist, and undecided.* For status, the bug reports are classified as *new, incomplete, invalid, confirmed, in progress, fix committed, fix released, under*

Table 5.3: Weighted Bug Reports: Version 2

Importance	***Number of bug reports***	***Weight***
Critical	3	1
High	8	1
Medium	11	1
Low	0	1
Wishlist	3	0
Undecided	111	0.88

consideration for removal, triaged, and won't fix. For this study, I also chose the open-source mobile application, Telegram [26] [27]. Telegram in Launchpad has two versions. Version 1 has 114 bug reports from 5 September 2014 to 7 April 2016 and version 2 has 136 bug reports from 8 April 2016 to 4 December 2019.

I weigh each bug report based on its validity. Valid bug reports whose importance levels range from *critical* to *low* are weighted 1. Wishlist bug reports are weighted 0 because they are not considered valid defects. Undecided bug reports could be valid 1 or wishlist 0. Therefore, I calculate the weight of an undecided bug report based on the ratio of the valid bug reports to the total of valid and wishlist bug reports as illustrated in Table 5.2 and Table 5.3.

The following equation is used to determine the undecided bug report weight:

$$U_{weight} = \frac{C + H + M + L}{C + H + M + L + W}\,, \tag{5.1}$$

where C is critical, H is high, M is medium, L is low, W is wishlist, and U is undecided.

Applying Equation 5.1, I get the following ratio of the undecided bug report for version 1 of the Telegram application:

$$\frac{2 + 13 + 22 + 9}{71} = 0.65. \tag{5.2}$$

The ratio of the undecided bug report for version 2 of The Telegram application is calculated as follows:

$$\frac{3 + 8 + 11}{25} = 0.88. \tag{5.3}$$

5.2.2. Modeling SRGMs

The main goal of introducing many software reliability growth models is to assess and analyze reliability growth through software testing and related defect arrival and removal. Non-homogeneous Poisson process (NHPP) software reliability models were developed to overcome the inconsistency of failure occurrence intervals. The NHPP models assume defects that are discovered during the testing phase are removed without introducing new defects, and the mobile application used in the field environment is the same as that used during the testing phase. In this case study, I use the following three commonly used SRGMs and Song's newly proposed model et al. [57]:

- The Goel-Okumoto model by Goel and Okumoto [49] is one of the most frequently used of the NHPP models and is defined as:

$$m(t) = N(1 - e^{-bt})\,, \tag{5.4}$$

  where N is the estimated total defects, and b is a constant.

- The S-shaped model by Yamada et al. [63]) is also an NHPP model, which predicts the cumulative defects in each given time (t) with constants $b > 0$ and $N > 0$ and is defined as:

$$m(t) = N(1 - (1 + bt)e^{-bt})\,, \tag{5.5}$$

  where b and N can be estimated from observation data.

- The Musa-Okumoto model by Musa et al. [46] is a logarithmic execution time model. This model is a different type of NHPP model and is defined as:

$$m(r) = \frac{1}{\phi}\log(\lambda_0\phi_r + 1)\,, \tag{5.6}$$

  where r is the measurement of the CPU-time execution, λ_0 is the intensity of the initial failure, and ϕ is a model parameter.

- A newly proposed model, which was presented by Song et al. [57] to measure software reliability while considering the uncertainty of operating systems and learn-curve in the fault detection rate function, is as follows:

$$m(t) = N\left(1 - \frac{\beta}{\beta + ln(\frac{a+e^{bt}}{1+a})}\right)^{\alpha}, \qquad (5.7)$$

where $\alpha >= 0$ and $\beta >= 0$ are constant.

5.3. Discussion and Analysis

After the selected SRGMs are applied to the weighted defect data sets, the reliability assessment and prediction of the case study are analyzed and discussed.

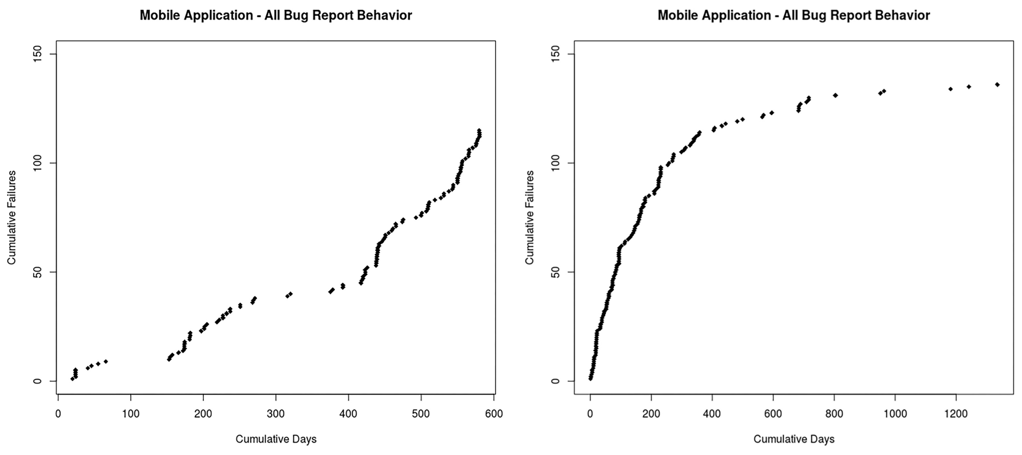
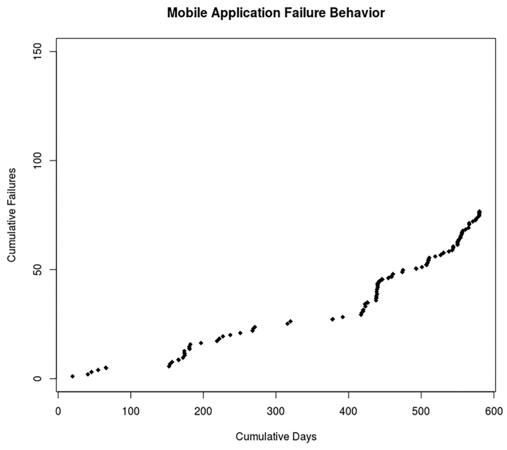


Figure 5.1: Bug Reports Distribution Over Time for Version 1 (Left) and Version 2 (Right)



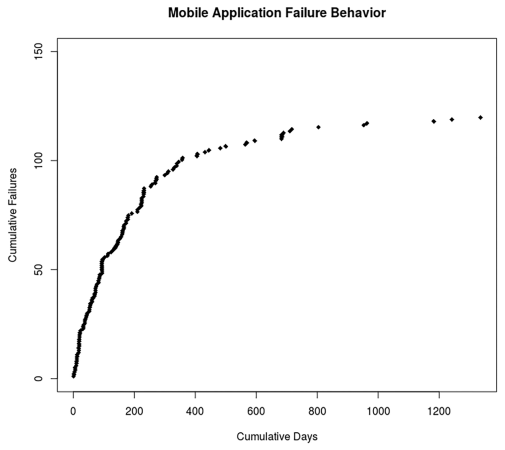


Figure 5.2: Valid Bug Reports Distribution for Version 1 (Left) and Version 2 (Right)

5.3.1. Reliability Assessment

After I weight each bug report for a valid defect, I plot the cumulative weighted defects over calendar time for Telegram version 1 and version 2. In the plot, the y-axis represents the cumulative number of defects, and x-axis represents the cumulative arrival day for each defect. Figure 5.1 shows the distribution of all bug reports, and Figure 5.2 shows the valid and weighted bug reports over time for both versions of the Telegram application. Version 1 contains 114 bug reports with a total of 76.6 cumulative defects. Version 2 contains 136 bug reports with a total of 119.68 cumulative defects.

To fit any SRGM model, the following should be satisfied. First, be aware of the used SRGM model assumptions. Second, the mobile application is tested by the same testing methodology in the same environment. Third, if the mobile application has new capabilities, the failure history should reflect these changes. Finally, the defect data set shows that the mobile application is reaching growth in reliability.

After plotting Telegram version 1 and version 2, version 1 plot shows that this version had ended before it reached a stable phase. Therefore, it was excluded from implementing SRGMs. Furthermore, SRGMs will only be fitted to version 2 for assessment and prediction. Figure 5.3 shows the fitted Goel-Okumoto, S-shaped, Musa-Okumoto, and Song's model on the number of defects over time. Table 5.4 shows the SRGM equation for the total number of defects over time.

As SRGMs are used to measure the growth in software reliability, there is a need to understand and evaluate the change in reliability. This is achieved through calculating the purification level ρ: The closer ρ is to 1, the more reliability growth in the application. When all failures are removed, ρ will become 1. This implies that the greater the ρ value, the more reliability growth I will have [59]:

$$\rho = \frac{\lambda_0 - \lambda_\tau}{\lambda_0} = 1 - \frac{\lambda_\tau}{\lambda_0}\,, \tag{5.8}$$

where the initial or peak failure rate for the models is represented by λ_0. The final failure rate is represented by λ_τ.

For further investigation and assurance of the results, I calculate the standard error of estimate (SEOE) to measure the accuracy of the SRGM predictions. The SEOE is defined as follows:

$$\sigma_{est} = \sqrt{\frac{\sum (Y - Y')^2}{N}}, \tag{5.9}$$

where σ_{est} is the standard error of the estimate, Y is an actual defect, Y' is a predicted defect, and N is the total number of defects. The numerator is the sum of the squared differences between the actual defects and the predicted defects.

Table 5.5 lists the ρ values of the selected SRGMs for version 2 of the mobile application, representing the potential reliability improvement estimated by the SRGMs.

Continuous testing and fault removal will lead to a decrease in the failure rate, or what is referred to as improvement in potential reliability, to be between 93.8% and 99.4% in version 2, which is significant.

Table 5.5 also shows the SEOE for the SRGMs in version 2 is between 0.288 and 0.735. This indicates that the most fitting model is Goel-Okumoto, and the prediction is close to the real data based on the calculated small values as shown in the results.

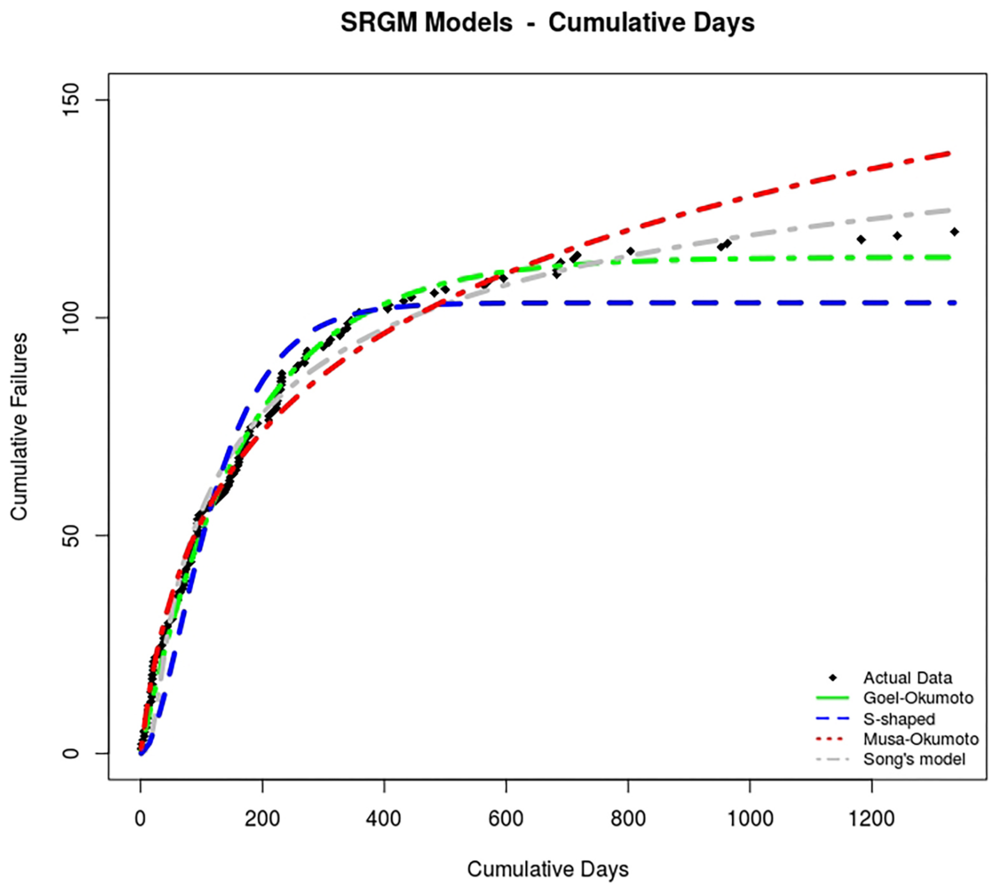


Figure 5.3: SRGMs Fitted on Valid Failures for Version 2

Table 5.4: SRGM Equations Fitted on Valid Failures For Version 2

SRGM	Equation
Goel-Okumoto	$\mu(t) = 113.9(1 - e^{-0.005897t})$
S-shaped	$\mu(t) = 103.43603(1 - (1 + 0.01591t)e^{-0.01591t})$
Musa-Okumoto	$\mu(\tau) = 1/0.02788 \ln((0.02788 * 1.23059t) + 1)$
Song's model	$\mu(t) = 197.211\big(1 - \frac{9}{9+ln(\frac{197.211+e^{0.2705t}}{1+197.211})}\big)^{0.2705}$

Table 5.5: Purification and SEOE Values of Selected SRGMs

SRGM	Version 2	
	ρ	$SEOE$
Goel-Okumoto	0.994	0.228
S-shaped	0.938	0.735
Musa-Okumoto	0.98	0.423
Song's model	0.975	0.469

5.3.2. Reliability Prediction

To test the prediction of the selected SRGMs, I use the undecided bug report weight for version 2 as a percentage to select the number of failures. Therefore, I use the first 88% of the defects to predict the last 12%. Figure 5.4 shows the prediction of the selected SRGMs where the vertical line separates the selected failures from the predicted failures. In addition, Table 5.6 shows the fitted model equations for the selected SRGMs. The results show that the models' predictions are not far from the actual failures. Therefore, developers can use known SRGMs to evaluate and predict the reliability of mobile applications.

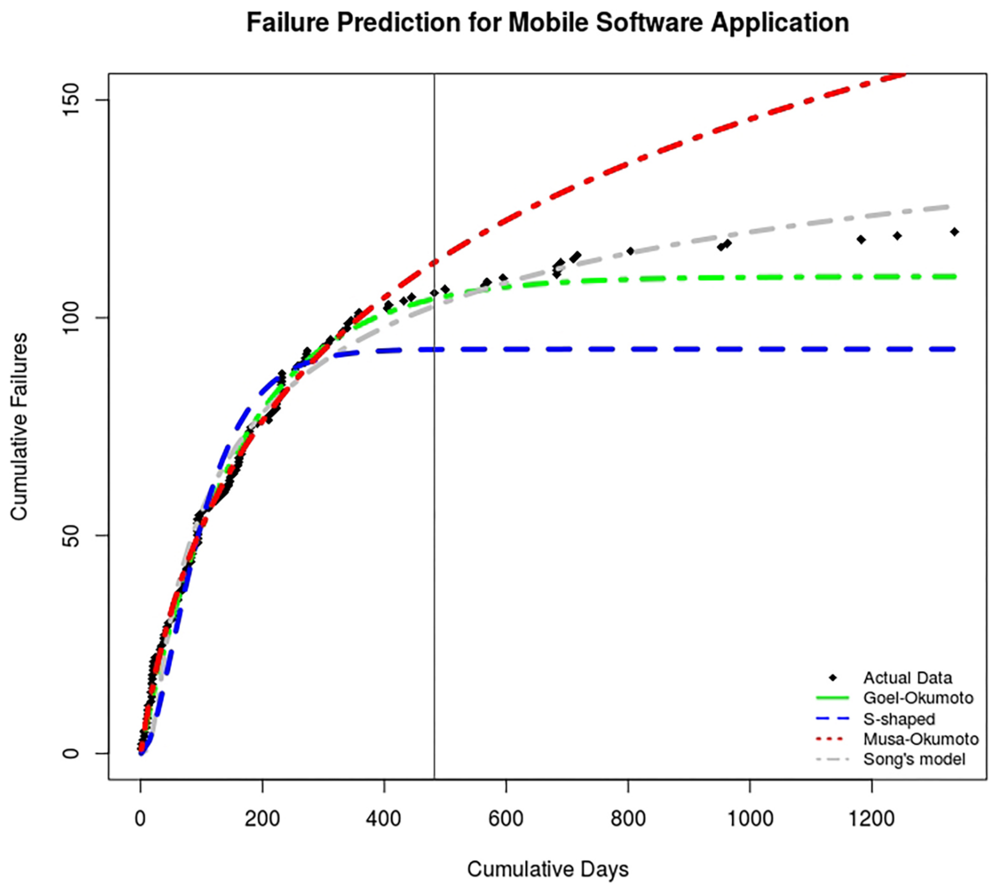


Figure 5.4: SRGMs Fitted on Partial Valid Failures for Version 2

Table 5.6: SRGM Equations Fitted on Partial Valid Failures for Version 2

SRGM	Equation
Goel-Okumoto	$\mu(t) = 109.46932(1 - e^{-0.00636t})$
S-shaped	$\mu(t) = 92.79993(1 - (1 + 0.01911t)e^{-0.01911t})$
Musa-Okumoto	$\mu(\tau) = 1/0.02134 \ln((0.02134 * 0.92742t) + 1)$
Song's model	$\mu(t) = 200.4941\big(1 - \frac{9}{9+ln(\frac{200.4941+e^{0.2669t}}{1+200.4941})}\big)^{0.2669}$

Chapter 6
SUMMARY, CONCLUSION, AND PERSPECTIVE

6.1. Accomplishments and Future Work

In this dissertation, I provided three contributions, code refactoring analysis, defect analysis, and reliability analysis, in order to improve the quality of mobile software.

In the first contribution, I provided an evaluation of code refactoring techniques for energy consumption in mobile software systems by using GPS-UP metrics. I also introduced two new categories to the GPS-UP metrics to better categorize the impact of refactoring techniques on mobile applications. I presented a case study using GPS-UP metrics to evaluate refactoring techniques in Fowler's sample code. In addition, I extend my work through evaluating refactoring techniques in mobile application code that contains common algorithms(Quick Sort and Binary Search) and data structures(Linked List) and two real open-source mobile applications(Simple Calculator and AnotherMonitor). Moreover, I provide a comparison between the results of all case studies.

Follow-up work in this area includes evaluating additional code refactoring techniques in open-source mobile applications running in different mobile software systems to generalize our refactoring technique profile. In addition, different metrics will be used to evaluate the impact of refactoring techniques and provide a comparison between the metrics results.

This work was published by:

- The 16th International Conference on Software Engineering Research and Practice(SERP, 2018) [6].
- International Journal of Computer Applications (IJCA, 2020) [8].

In the second contribution, I proposed adapting the ODC model to mobile environments to classify defects during software development and after release. New characteristics and factors of mobile environments and applications were studied and considered, which led to adding new defect types to the original ODC framework. In addition, a case study was presented where defects from bug reports were extracted and classified based on the proposed framework. Moreover, I provided one-way and two-way analyses and discussed their results. Our work will make in-process feedback more accurate during mobile software development and improve software reliability.

Follow-up work in this area includes classifying defects of mobile applications from different mobile environments in order to generalize my findings.

This work was published at:

- The 28th International Conference on Software Engineering and Data Engineering(SEDE, 2019) [7].

In the third contribution, I proposed measuring the reliability of mobile applications based on defects extracted from bug reports. The proposed process is composed of six steps. First, extract and characterize the bug reports for an open-source mobile application. Second, analyze the bug reports to discard ones that are not related to software reliability. Third, weight the bug reports based on their classification. Fourth, relate the date of each bug report to the total number of days since the release day of the mobile application. Fifth, assess and evaluate the reliability of the selected mobile application and predict its future reliability using SRGMs. Finally, use the purification rate and the SEOE for assurance and provide the fitted SRGM equations.

The results demonstrated that the reliability of mobile applications can be evaluated and predicted using SRGMs through defect data extracted from bug reports. This enables developers to evaluate and predict the reliability of mobile applications.

Follow-up work in this area includes assessing and predicting more open-source mobile application in different mobile environments.

This work was accepted by:

- Journal of Software Engineering and Applications (JSEA, 2020).

BIBLIOGRAPHY

[1] Abdulsalam, S., Zong, Z., Gu, Q., and Qiu, M. Using the greenup, powerup, and speedup metrics to evaluate software energy efficiency. In *2015 Sixth International Green and Sustainable Computing Conference (IGSC)* (Dec 2015), pp. 1–8.

[2] Alannsary, M., and Tian, J. Cloud-odc: Defect classification and analysis for the cloud. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)* (2015), The Steering Committee of The World Congress in Computer Science, p. 71.

[3] Alannsary, M. O., and Tian, J. Measurement and prediction of saas reliability in the cloud. In *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)* (2016), IEEE, pp. 123–130.

[4] Almering, V., van Genuchten, M., Cloudt, G., and Sonnemans, P. J. Using software reliability growth models in practice. *IEEE software 24*, 6 (2007), 82–88.

[5] Banerjee, A., Chong, L. K., Chattopadhyay, S., and Roychoudhury, A. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 588–598.

[6] Barack, O., and Huang, L. Effectiveness of code refactoring techniques for energy consumption in a mobile environment. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)* (2018), The Steering Committee of

The World Congress in Computer Science, Computer . . . , pp. 165–171.

[7] Barack, O., and Huang, L. Adaptation of orthogonal defect classification for mobile applications. In *Proceedings of the 28th International Conference on Software Engineering and Data Engineering (SEDE)* (2019), vol. 64, pp. 119–128.

[8] Barack, O., and Huang, L. A detailed comparison of the effects of code refactoring techniques in different mobile applications. *International Journal of Computer Applications* (2020).

[9] Bilodeau, O. Tomdroid application. https://launchpad.net/tomdroid.

[10] Bilodeau, O. Tomdroid bug report. https://github.com/tomboy-notes/tomdroid/issues.

[11] Blum, B. I. *Software Engineering: A Holistic View.* Oxford University Press, Inc., 1992.

[12] Boehm, B. Software engineering. *IEEE Transactions on Computers C-25*, 12 (Dec 1976), 1226–1241.

[13] Bokhary, A. Measuring cloud service reliability by weighted defects over the number of clients as a proxy for usage. In *Proceedings of the 32nd International Conference on Computers and Their Applications (CATA)* (2017), pp. 63–70.

[14] Bunse, C., Höpfner, H., Mansour, E., and Roychoudhury, S. Exploring the energy consumption of data sorting algorithms in embedded and mobile environments. In *2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware* (May 2009), pp. 600–607.

[15] Chillarege, R. Odc-a 10x for root cause analysis. 2006.

[16] Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J., Moebus, D. S., Ray, B. K., and Wong, M.-Y. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on software Engineering 18*, 11 (1992), 943–956.

[17] Comito, C., and Talia, D. Evaluating and predicting energy consumption of data mining algorithms on mobile devices. In *Data Science and Advanced Analytics (DSAA), 2015. 36678 2015. IEEE International Conference on* (2015), IEEE, pp. 1–8.

[18] da Silva, W. G., Brisolara, L., Corrêa, U. B., and Carro, L. Evaluation of the impact of code refactoring on embedded software efficiency. In *Proceedings of the 1st Workshop de Sistemas Embarcados* (2010), pp. 145–150.

[19] Fekete, K., Pelle, , and Csorba, K. Energy efficient code optimization in mobile environment. In *2014 IEEE 36th International Telecommunications Energy Conference (INTELEC)* (Sept 2014), pp. 1–6.

[20] Flinn, J., and Satyanarayanan, M. Powerscope: A tool for profiling the energy usage of mobile applications. In *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA'99. Second IEEE Workshop on* (1999), IEEE, pp. 2–10.

[21] Fowler, M. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[22] GNU. Anothermonitor application. https://f-droid.org/en/packages/org.anothermonitor.

[23] Graveley, A. Tomdroid application. https://github.com/tomboy-notes/tomboy.

[24] Hao, S., Li, D., Halfond, W. G., and Govindan, R. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE Press, pp. 92–101.

[25] Hasan, S., King, Z., Hafiz, M., Sayagh, M., Adams, B., and Hindle, A. Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering* (2016), ACM, pp. 225–236.

[26] Herrmann, T. Telegram application. https://launchpad.net/telegram-app.

[27] Herrmann, T. S. Telegram bug report. https://bugs.launchpad.net/telegram-app.

[28] Holl, K., and Elberzhager, F. A mobile-specific failure classification and its usage to focus quality assurance. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications* (2014), IEEE, pp. 385–388.

[29] Huang, C.-Y., Kuo, S.-Y., and Lyu, M. R. An assessment of testing-effort dependent software reliability growth models. *IEEE transactions on Reliability 56*, 2 (2007), 198–211.

[30] Hunt, N., Sandhu, P. S., and Ceze, L. Characterizing the performance and energy efficiency of lock-free data structures. In *Interaction between Compilers and Computer Architectures (INTERACT), 2011 15th Workshop on* (2011), IEEE, pp. 63–70.

[31] Ivanov, V., Reznik, A., and Succi, G. Comparing the reliability of software systems: A case study on mobile operating systems. *Journal of Information Sciences 423* (2018), 398–411.

[32] Jones, C. *Applied software measurement: global analysis of productivity and quality.* McGraw-Hill Education Group, 2008.

[33] Kaputa, T. Simple Calculator Application. https://f-droid.org/en/packages/com.simplemobiletools.calculator.

[34] Lelli, V., Blouin, A., and Baudry, B. Classifying and qualifying gui defects. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)* (2015), IEEE, pp. 1–10.

[35] Li, N., Li, Z., and Sun, X. Classification of software defect detected by black-box testing: An empirical study. In *2010 Second World Congress on Software Engineering* (2010), vol. 2, IEEE, pp. 234–240.

[36] Limited, F.-D. F-droid. https://www.f-droid.org.

[37] Ltd, C. Launchpad. https://launchpad.net.

[38] Ltd, C. Launchpad bug tracking. https://bugs.launchpad.net.

[39] Lyu, M. R., et al. *Handbook of software reliability engineering*, vol. 222. IEEE computer society press CA, 1996.

[40] Ma, L., and Tian, J. Web error classification and analysis for reliability improvement. *Journal of Systems and Software 80*, 6 (2007), 795–804.

[41] Medellin, J., Barack, O., Zhang, Q., and Huang, L. Enabling migration decisions through policy services in soa mobile cloud computing systems. In *Proceedings of the International Conference on Grid Computing and Applications (GCA)* (2015), The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), p. 23.

[42] Meskini, S., Nassif, A. B., and Capretz, L. F. Reliability models applied to mobile applications. In *2013 IEEE seventh international conference on software security and reliability companion* (2013), IEEE, pp. 155–162.

[43] Metri, G., Shi, W., and Brockmeyer, M. Energy-efficiency comparison of mobile platforms and applications: A quantitative approach. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications* (New York, NY, USA, 2015), HotMobile '15, ACM, pp. 39–44.

[44] Mittal, R., Kansal, A., and Chandra, R. Empowering developers to estimate app energy consumption. In *Proceedings of the 18th annual international conference on Mobile computing and networking* (2012), ACM, pp. 317–328.

[45] Morales, R., Saborido, R., Khomh, F., Chicano, F., and Antoniol, G. Earmo: an energy-aware refactoring approach for mobile apps. *IEEE Transactions on Software Engineering*, 1 (2017), 1–1.

[46] Musa, J. D., Iannino, A., and Okumoto, K. Software reliability. *Advances in computers 30* (1990), 85–170.

[47] Muss, J., Iannino, A., and Okumoto, K. Software reliability: Measurement, prediction, application, 1987.

[48] Naur, P. Software engineering-report on a conference sponsored by the nato science committee garimisch, germany. *http://homepages. cs. ncl. ac. uk/brian. randell/NATO/nato1968. PDF* (1968).

[49] Okumoto, K., and Goel, A. L. Optimum release time for software systems based on reliability and cost criteria. *Journal of Systems and Software 1* (1984), 315–318.

[50] Park, J. J., Hong, J.-E., and Lee, S.-H. Investigation for software power consumption of code refactoring techniques. In *SEKE* (2014), pp. 717–722.

[51] Perera, U. D. Reliability index-a method to predict failure rate and monitor maturity of mobile phones. In *RAMS'06. Annual Reliability and Maintainability Symposium, 2006.* (2006), IEEE, pp. 234–238.

[52] Ramirez, R. I., Rubio, E. H., Viveros, A. M., and Hernández, I. M. T. Differences of energetic consumption between java and jni android apps. In *2014 International Symposium on Integrated Circuits (ISIC)* (Dec 2014), pp. 348–351.

[53] Rashid, M., Ardito, L., and Torchiano, M. Energy consumption analysis of algorithms implementations. In *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on* (2015), IEEE, pp. 1–4.

[54] Sahin, C., Pollock, L., and Clause, J. How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (2014), ACM, p. 36.

[55] Sahin, C., Tornquist, P., Mckenna, R., Pearson, Z., and Clause, J. How does code obfuscation impact energy usage? In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on* (2014), IEEE, pp. 131–140.

[56] Shanmugam, L., and Florence, L. An overview of software reliability models. *International Journal of Advanced Research in Computer Science and Software Engineering 2*, 10 (2012).

[57] Song, K. Y., Chang, I. H., and Pham, H. Nhpp software reliability model with inflection factor of the fault detection rate considering the uncertainty of software operating environments and predictive analysis. *Symmetry 11*, 4 (2019), 521.

[58] The International Organization for Standardization. ISO/IEC 25010 Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models. https://www.iso.org/standard/35733.html.

[59] Tian, J. Integrating time domain and input domain analyses of software reliability using tree-based models. *IEEE Transactions on Software Engineering 21*, 12 (1995), 945.

[60] Tian, J., Rudraraju, S., and Li, Z. Evaluating web software reliability based on workload and failure data extracted from server logs. *IEEE Transactions on Software Engineering 30*, 11 (2004), 754–769.

[61] Vithani, T., and Kumar, A. Modeling the mobile application development lifecycle. In *Proceedings of the International MultiConference of Engineers and Computer Scientists* (2014), vol. 1.

[62] Wasserman, T. Software engineering issues for mobile application development. *FoSER 2010* (2010).

[63] Yamada, S., Ohba, M., and Osaki, S. S-shaped reliability growth modeling for software error detection. *IEEE Transactions on Reliability 5* (1983), 475–484.

[64] Yang, Z. Powertutor-a power monitor for android-based mobile platforms. *EECS, University of Michigan, retrieved September 2* (2012), 19.

[65] Zecena, I., Zong, Z., Ge, R., Jin, T., Chen, Z., and Qiu, M. Energy consumption analysis of parallel sorting algorithms running on multicore systems. In *Green Computing Conference (IGCC), 2012 International* (2012), IEEE, pp. 1–6.