An Empirical Investigation into Code Smells Elimination Sequences for Energy Efficient Software

A DISSERTATION
SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR
THE AWARD OF THE DEGREE
OF

MASTER OF TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING

SUBMITTED BY
Garima Dhaka
Roll No. 14203015

UNDER THE SUPERVISION OF
Dr. Paramvir Singh
Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
DR. B. R. AMBEDKAR NATIONAL INSTITUTE OF TECHNOLOGY
JALANDHAR – 144011, PUNJAB (INDIA)
JULY, 2016
CANDIDATE’S DECLARATION

I hereby certify that the work, which is being presented in the dissertation, entitled “An Empirical Investigation into Code Smells Elimination Sequences for Energy Efficient Software” by “Garima Dhaka” in partial fulfillment of requirements for the award of degree of M.Tech. (Computer Science and Engineering) submitted to the Department of Computer Science and Engineering of Dr. B R Ambedkar National Institute of Technology, Jalandhar, is an authentic record of my own work carried out during a period from August, 2015 to July, 2016 under the supervision of Dr. Paramvir Singh, Assistant Professor. The matter presented in this dissertation has not been submitted by me in any other University/Institute for the award of any degree.

Garima Dhaka
Roll no. 14203015

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Dr. Paramvir Singh (Supervisor)
Assistant Professor
Department of Computer Science & Engineering
Dr. B. R. Ambedkar NIT, Jalandhar

The M.Tech (Dissertation) Viva-Voce examination of Garima Dhaka, Roll No. 14203012, has been held on ____________ and accepted.

External Examiner
Supervisor
Head of Department
ACKNOWLEDGEMENTS

Foremost, I would like to express my gratitude to my supervisor Dr. Paramvir Singh, Assistant Professor, for the useful comments, remarks and engagement through the learning process of this master thesis. I cannot thank him enough for his tremendous support and help. He motivated and encouraged me throughout this work. Without his encouragement and guidance this project would not have materialized. I consider myself extremely fortunate to have a chance to work under his supervision. In spite of his busy schedule, he was always approachable and took his time off to guide me and gave appropriate advice.

I also wish to thank whole heartily all the faculty members of the Department of Computer Science and Engineering for the invaluable knowledge they have imparted on me and for teaching the principles in most exciting and enjoyable way. I also extend my thanks to the technical and administrative staff of the department for maintaining an excellent working facility.

I would like to thank my family for their continuous support and blessings throughout the entire process, both by keeping me harmonious and helping me putting me pieces together. I also like to thank all my batch mates for the useful discussions, constant support and encouragement during whole period of the work.

Last but not the least, I would like to thank almighty GOD for giving me enough strength and lifting me uphill this phase of life.

Garima Dhaka
ABSTRACT

Green IT is a modern that refers to the development of energy efficient products (software and hardware). Software that runs on the hardware is responsible for the amount of energy consumed by it. Incorporating sustainability concerns into software development practices is one of the fastest emerging trends in software industry. Recent sustainability studies have shown that changing the internal code structure of the software affects its energy consumption behavior as well as architecture maintainability.

Code smells refer to the bad design structures which makes software difficult to evolve, maintain, read, reuse etc. Various refactoring techniques are defined to eliminate code smells. Refactoring alters the internal code structure of software systems while preserving the external behavior. Since internal code structure is modified in the process of eliminating code smells, it changes software energy consumption behavior as well as software metrics.

This work empirically investigates the impact of eliminating three of the notorious code smells - god class (G), feature envy (F) and long method (L), individually as well as in all possible sequences (GFL, GLF, FGL, FLG, LGF, LFG), on architecture maintainability and energy consumption behavior of software systems. The experiments have been performed using three open source java applications.

The study outcomes show that these permutations yield variant levels of energy consumption values for the resulting refactored software versions. It is also observed that a particular permutation (GFL) is learned to yield most energy efficient refactored software version, in comparison to those yielded by all other permutations for all three applications; thus revealing consistency. Results also revealed that there is no definite relationship between software architecture metrics and software energy consumption. These findings can be useful for the software developers in understanding and adopting those code smells elimination sequences which may result in more sustainable refactored software versions.
LIST OF PUBLICATIONS

# TABLE OF CONTENTS

- **CERTIFICATE**
- **ACKNOWLEDGEMENT**
- **ABSTRACT**
- **LIST OF PUBLICATIONS**
- **LIST OF FIGURES**
- **LIST OF TABLES**
- **LIST OF ABBREVIATIONS**

## CHAPTER 1

**INTRODUCTION**

1.1 Green IT 01

1.2 Energy Aware Software 02

1.3 Energy Measurement Tool 02

1.4 Code Smells and Refactoring Techniques 03

  1.4.1 Code Smells 03

  1.4.2 Refactoring techniques 05

1.5 Software Architecture Metrics 07

1.6 Motivation 07

  1.6.1 An Example Basis for Exploring Code Smell Elimination Sequences 08

1.7 Research Objectives 14

1.8 Thesis Outline 14
CHAPTER 2
LITERATURE REVIEW

2.1 Code Smell and Impact Analysis 15

2.2 Refactoring and Impact Analysis 17
   2.2.1 Refactoring 17
   2.2.2 Impacts of Refactorings on Energy Consumption of Software 18

2.3 Impact of Design Patterns on Energy Consumption of Software 23

2.4 Approaches for Developing Energy Aware Software 25

2.5 Chapter Summary 26

CHAPTER 3
EXPERIMENTAL DESIGN AND METHODOLOGY

3.1 Sample Applications 27

3.2 Selected Code Smells and Refactoring Techniques 28

3.3 Selected Metrics 29

3.4 Tools Used 30
   3.4.1 JDeodorant 30
   3.4.2 Atlassian Clover 34
   3.4.3 Code pro Analytix 35
   3.4.4 ckjm 36
   3.4.5 Jalen 37

3.5 Methodology 39
   3.5.1 Experimental Subject Creation and Metric Calculation 39
CHAPTER 4

EXPERIMENTAL RESULTS AND ANALYSIS

4.1 Analysis of Type 1 Refactored Versions
   4.1.1 Analysis of Feature Envy Removed Version
   4.1.2 Analysis of God Class Removed Version
   4.1.3 Analysis of Long Method Removed Version
   4.1.4 Analysis of Energy Trends

4.2 Analysis of Type 1 Refactored Versions
   4.2.1 Analysis of Versions of JHotdraw
   4.2.2 Analysis of Versions of Commons Beanutils
   4.2.3 Analysis of Versions of Commons IO

4.3 Result Summary

4.4 Chapter Summary

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusions

5.2 Future Work

REFERENCES
LIST OF FIGURES

Figure 1.1 Removal of feature envy after long method—ATM Example 10
Figure 1.2 Removal of long method after feature envy—ATM Example 10
Figure 1.3 Removal of god class after long method—ATM Example 12
Figure 1.4 Removal of long method after god class—ATM Example 12
Figure 1.5 Removal of god class after feature envy—ATM Example 13
Figure 1.6 Removal of feature envy after god class—ATM Example 13
Figure 3.1 Visual representation of god class 31
Figure 3.2 Visual representation of feature envy 32
Figure 3.3 Illustration of long method code smell 33
Figure 3.4 Eliminated long method code smell 33
Figure 3.5 Metric values using atlassian’s clover tool 35
Figure 3.6 Metric values using code pro analytix tool 36
Figure 3.7 Metrics calculation using ckjm tool 37
Figure 3.8 Working of Jalen tool 38
Figure 3.9 Energy consumption using Jalen 42
Figure 3.10 Methodology flow 41
Figure 4.1 Energy trends across sample applications 46
# LIST OF TABLES

| Table 1.1 | Energy measurement tools | 03 |
| Table 1.2 | Code smell detection tools | 05 |
| Table 1.3 | Mapping code smells with energy consumption | 08 |
| Table 2.1 | Description of code smells | 16 |
| Table 2.2 | Refactoring’s benefits and risks | 18 |
| Table 2.3 | Potential energy smells | 23 |
| Table 3.1 | Sample java applications | 27 |
| Table 3.2 | Refactoring techniques corresponding to code smells | 30 |
| Table 4.1 | Comparison of architectural metrics and code smell information of type 1 refactored versions | 45 |
| Table 4.2 | Architectural metrics and code smell information for refactored versions of JHotdraw | 48 |
| Table 4.3 | Architectural metrics and code smell information for refactored versions of Commons Beanutils | 49 |
| Table 4.4 | Architectural metrics and code smell information for refactored versions of Commons IO | 50 |
### LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Feature Envy</td>
</tr>
<tr>
<td>L</td>
<td>Long Method</td>
</tr>
<tr>
<td>G</td>
<td>God Class</td>
</tr>
<tr>
<td>OA</td>
<td>Original Application</td>
</tr>
<tr>
<td>RA</td>
<td>Refactored Version</td>
</tr>
<tr>
<td>CC</td>
<td>Cyclomatic Complexity</td>
</tr>
<tr>
<td>CBO</td>
<td>Coupling Between Objects</td>
</tr>
<tr>
<td>LOC</td>
<td>Lines Of Code</td>
</tr>
<tr>
<td>JH</td>
<td>JHotdraw</td>
</tr>
<tr>
<td>CB</td>
<td>Commons Beanutils</td>
</tr>
<tr>
<td>CI</td>
<td>Commons IO</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

Environment is being adversely affected by human practices, resources are being depleted and level of carbon dioxide in atmosphere is already above the safety limit. Various industries and factories dispose off their harmful wastes in environment improperly without analyzing its adverse consequences. Measures must be taken in all areas to motivate the sustainable use of resources to prevent further damage to environment by adopting environment friendly practices in our day to day life.

Information Technology (IT) sector has significant contribution in human greenhouse gas emissions worldwide [14]. Energy consumed by hardware and software has negative impact on environment. With increasing environmental related issues, optimizing software’s energy consumption has become pivotal. One of the greatest challenges of research and industry is to optimize the energy consumption behavior of software applications. Significant number of studies has been conducted for improving the energy efficiency of hardware. It is the software which impacts the energy efficiency of entire system indirectly through hardware. Therefore, the focus is being shifted from hardware to software in terms of energy efficiency, to stimulate the development of energy efficient software systems. Initial steps are being taken towards developing energy aware software systems. Hermans et al. [11] presents the importance of energy aware software and how their development will be enforced in industries. Marginal improvement in energy efficiency in small scale software systems will leverage improvement in energy efficiency of the entire system significantly.

1.1 Green Information Technology (IT)

A relatively new concept of “Green IT” [18] has emerged, which refers to the eco-friendly practices that must be adopted in information and communication technology sector. Activities comprise of development of energy efficient software, designing and manufacturing of energy efficient hardware and proper disposal of computers, associated subsystems and servers so as to prevent any adverse effects on environment. It encourages the sustainable [33] development of sustainable products (software and hardware). Industries are fast adopting Green IT (or Green Computing), the practice of environment friendly computing
However, green computing is still in its initial stage, and lot of research work is still required to be done.

1.2 Energy Aware Software

Software which is energy efficient can be termed as “Energy Aware Software”. The demands for energy aware software (or green software) are increasing; hence it becomes necessary to evaluate the software change in terms of energy consumption also. In near future, software will be rated by developers and customers according to their greenness, IDEs will be such that apart from highlighting errors such as compilation errors, they will support the development of energy aware software by warning about the energy wasteful conditions in source code [11]. Several studies have been conducted [1, 2, 4, 6–8, 12-14, 20, 23-29] which support the fact that changing the source code of software alters its energy consumption behavior. Refactoring techniques (see Section 1.4.2) defined by Martin Fowler [5] do not take into consideration the energy consumption of software system. Hence it becomes necessary to analyze the changes in source code of software in terms of energy efficiency. Lot of research work is needed to carry out in the field of green software engineering to aid in the development of green software (or energy aware software).

1.3 Energy Measurement Tools

Energy measurement techniques can broadly be classified into three categories [21]:

- **Hardware Measurement Techniques**: As the name suggests, this technique requires external hardware for its implementation. It provides accurate results but at coarse granularity. This measurement technique is difficult to scale up and evolve.

- **Power models**: This category provides power models based upon which tools can be developed. Energy estimation using these models is possible, but models based tools are either platform dependent or too coarse or generic to be used.

- **Software Measurement Techniques**: This category provides promising results in energy consumption measurement. They can further be divided as instrumentation and sampling techniques. Instrumentation technique provides accurate results but it has an overhead of introducing instrumentation statements in code structure. Sampling technique on the other hand provides approximate energy values and no overhead is introduced.
Noureddine et al. [21] performed a review on energy measurement tools and outlined different categories of energy measurement approaches presented in Table 1.1.

<table>
<thead>
<tr>
<th>Energy Measurement Tool</th>
<th>Monitored Resources</th>
<th>Energy Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerScope [52]</td>
<td>Current Program Counter</td>
<td>Process, Procedure</td>
</tr>
<tr>
<td>PTop [53]</td>
<td>Hardware resources (CPU, disk, network)</td>
<td>Process</td>
</tr>
<tr>
<td>PowerTop [54]</td>
<td>Hardware Resources</td>
<td>Application</td>
</tr>
<tr>
<td>Energy Checker [55]</td>
<td>Hardware Resources, Application Counters</td>
<td>Application</td>
</tr>
<tr>
<td>JouleMeter [56]</td>
<td>Hardware resources,</td>
<td>Process</td>
</tr>
<tr>
<td>PowerAPI [57, 58]</td>
<td>Hardware resources</td>
<td>Process</td>
</tr>
<tr>
<td>Jalen [22]</td>
<td>Software resources</td>
<td>Code blocks (methods)</td>
</tr>
</tbody>
</table>

In this study, Jalen is used [22] (see Section 3.4.5) as it follows a software based approach and provides energy measurements at a finer level of granularity.

1.4 Code Smells and Refactoring Techniques

Code smell or bad smell term is used to describe the bad code structure which is difficult to understand, read and maintain. There are twenty two types of code smells described by Fowler [5]. In order to get rid of these code smells, refactoring techniques are used. Around seventy two refactoring techniques are defined by Fowler [5]. Refactoring techniques change the internal code structure without modifying the external behavior of software system, thus improving understandability, readability and maintainability of software code. It is a controlled technique for enhancing the quality attributes of existing software system.

1.4.1 Code Smell

According to M. Fowler [5] “code smell is a surface indication that usually corresponds to a deeper problem in the software system”. It is known to deteriorate the software quality, thus making it difficult to evolve and maintain. Code smells do not hinder the normal functioning of system; their presence make software design weak in terms of increased risk of bugs and
failures. Hence, they must be eliminated for software system to be healthy i.e. maintainable, readable, understandable etc. Twenty two code smells defined by Fowler can be categorized as Bloaters, Object Orientation Abusers, Change Preventers, Dispensables, and Couplers.

Following is the description of above mentioned categories:

- **Bloaters**: This category refers to those types of code smells in which functions, classes, parameters etc. are elongated over time, and have become difficult to understand and work with. It includes God Class, Long Method, Primitive Obsession, Long Parameter List and Data Clumps.

- **Object Orientation Abusers**: This category includes those code smells which represents inappropriate exercise of object oriented principles. It includes Switch Statements, Temporary Field, Refused Bequest, and Alternative Classes with different interfaces.

- **Change Preventers**: As the name itself suggests, this category of code smells make it strenous to perform even small modification in source code. It includes Divergent Change, Shotgun Surgery and Parallel Inheritance Hierarchies.

- **Dispensables**: This category presents those code smells which reflects irrelevant parts of code such as duplicated code, dead code etc, without which code will look neat and easy to understand, and evolve. This category includes Comments, Duplicate Code, Dead Code, Data Class, Speculative Generality and Lazy Class.

- **Couplers**: This category of code smells represents that code structure that indicates high coupling or unnecessary delegation. It includes Feature Envy, Inappropiate Intimacy, Message Chain and Middle Man.

The detail of each of these code smells is presented in Chapter 2.

While developing software, many code smells may go unattended. In order to handle such scenarios, code smell detection and elimination tools must be used. Table 1.2 shows the characteristics of code smell detection tools, with first column representing the tool name, second column representing the type whether standalone or not, third column represents the programming languages that tool can support, fourth column represents whether refactoring can be performed by the tool or not, and the fifth column represents whether tool provides direct link to code or not i.e. whether it indicates which section of code structure has design problem or not.
Table 1.2 Code smell detection tools [2]

<table>
<thead>
<tr>
<th>Tool</th>
<th>Type</th>
<th>Supported languages</th>
<th>Automated refactoring</th>
<th>Direct link to code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkstyle</td>
<td>Eclipse Plugin</td>
<td>Java</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Décor</td>
<td>Standalone</td>
<td>Java</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Iplasma</td>
<td>Standalone</td>
<td>C++, Java</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Infusion</td>
<td>Standalone</td>
<td>C, C++, Java</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>JDeodorant</td>
<td>Eclipse Plugin</td>
<td>Java</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>PMD</td>
<td>Eclipse Plugin</td>
<td>Java</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Stench Blossom</td>
<td>Eclipse Plugin</td>
<td>Java</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

In this study, JDeodorant is used (see Section 3.4.1) as it supports code smell detection as well as elimination.

1.4.2 Refactoring Techniques

In order to eliminate code smells, refactoring is performed. It is the process of changing code structure of the software system without affecting or modifying its external behavior [4]. Refactoring techniques are believed to improve the non functional attributes such as maintainability, understandability, readability, reliability and reusability etc. of the software system.

Existing refactoring techniques can be categorized as:

- **Composing Methods**: The refactoring techniques which belong to this category systematize methods, eliminate duplicated code and set at ease the future enhancements. It includes *Extract Method, Inline Method, Extract Variable, Inline Temp, Replace Temp with Query, Split Temporary Variable, Remove Assignments to parameters, Replace Method with Method Object and Substitute Algorithm.*
• **Moving Feature Between Objects:** Refactoring techniques belonging to this category effectively move functionalities among classes, generate new classes from existing ones and conceal implementation specifics to prevent public access. It include *Move Method, Move Field, Extract Class, Inline Class, Hide Delegate, Remove Middle Man, Introduce Foreign Method and Introduce Local Extension.*

• **Organizing Data:** This type of refactoring techniques assist in handling data, disentangling of class associations making them reusable and portable. It includes *Self Encapsulate Field, Replace Data Value with Object, Change Value to Reference, Change Reference to Value, Replace Array with Object, Duplicate Observed Data, Change Unidirectional Association to Bidirectional, Change Bidirectional Association to Unidirectional, Replace Magic Number with Symbolic Constant, Encapsulate Field, Encapsulate Collection, Replace Type Code With Class, Replace Type Code With Subclasses, Replace Type Code with State/Strategy, Replace Subclass with Fields.*

• **Simplifying Conditional Expressions:** Logic of conditionals is likely to become more complex over time; this category of refactoring techniques are used to address this issue. It include *Decompose Conditional, Consolidate Conditional, Consolidate Duplicate Conditional Fragments, Remove Control Flag, Replace Nested Conditional with Guard Clauses, Replace Conditional with Polymorphism, Introduce Null Object and Introduce Assertion.*

• **Simplifying Method Calls:** Refactoring techniques belonging to this type make function calls elementary and easy to understand thus simplifying interfaces for communication among classes. It includes *Rename Method, Add Parameter, Remove Parameter, Separate Query From Modifier, Parameterize Method, Replace Parameter with Explicit Methods, Preserve Whole Object, Replace Parameter with Method Call, Introduce Parameter Object, Remove Setting Method, Hide Method, Replace Constructor with Factory Method, Replace Error Code with Exception and Replace Exception with Test.*

• **Dealing with Generalization:** These types of refactoring techniques are associated with migrating responsibilities along the hierarchies of classes, introducing new interfaces and classes and replacing inheritance hierarchies with delegation and vice versa. It includes *Pull Up Field, Pull Up Constructor, Pull Up Method, Push Down Method, Push Down Field, Extract Subclass, Extract Superclass, Extract Interface,*
1.5 Software Architecture Metrics

Software metrics can be defined as functions which are used to measure the characteristics of source code such as number of lines in a source code, number of classes, number of methods, coupling between objects etc. Metrics are very useful as they help tracking down the risks and those code sections that can be troublesome in future.

These metrics capture static characteristics of software systems such as number of methods, lines of code, coupling between objects, number of classes, cyclomatic complexity etc. They are used to monitor the complexity of the software and locate the possible risk areas in software code. Information obtained from these metrics aid in estimating the cost of software development and is use to analyze the quality of software.

1.6 Motivation

This section elucidates the encouragement behind this study. It discusses the work performed by Castillo et al. [24] and Park et al. [23], followed by the motivation and brief outline of this work. It further discusses an example of ATM system to explain the motivation and research objectives of this study.

Castillo et al. [24] analyzed the impact of removing god classes on two open source software applications in terms of energy consumption. Results obtained from their study indicated that removing god classes increase energy consumption of software applications as message traffic increases. Several other code smells exist in software systems, motivated from their initial investigations into the impact of god class code smell on energy consumption behavior of software system; this work is dedicated to explore the impact of other code smells as well on energy consumption behavior of the software system. In addition to god class, this work explores the effect of two other most common code smells, feature envy and long method, on energy consumption behavior of three open source java applications.

Park et al. [23] investigated the energy consumption of refactoring techniques. Their findings are mapped with the code smells and estimated the influence of removing code smells by checking the influence of corresponding refactoring technique on software energy.
consumption as shown in Table 1.3. This estimation resulted in decreased energy consumption due to the removal of feature envy code smell (removed by move method refactoring) and increased energy consumption due to the removal of god class code smell (removed by extract class refactoring) and long method code smell (removed by extract method refactoring). This theoretical estimation further motivated us to empirically analyze the changes in software energy consumption behavior resulting from the elimination of code smell.

Table 1.3 Mapping code smells with energy consumption

<table>
<thead>
<tr>
<th>Code Smell</th>
<th>Refactoring Technique</th>
<th>Energy Consumption (µJ)</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Refactor</td>
<td></td>
</tr>
<tr>
<td>Feature Envy</td>
<td>Move Method</td>
<td>70.066</td>
<td>↓</td>
</tr>
<tr>
<td>God Class</td>
<td>Extract Method</td>
<td>83.077</td>
<td>↑</td>
</tr>
<tr>
<td>Long Method</td>
<td>Extract Method</td>
<td>72.361</td>
<td>↑</td>
</tr>
</tbody>
</table>

Also considering the more practical scenario that code smells are removed in sequences, this paper investigates the impact of different permutations of removing the above mentioned code smells on energy consumption behavior of software applications.

1.6.1 An Example Basis for Exploring Code Smell Elimination Sequences

This subsection builds a theoretical basis for the need to explore the code smell removal sequences with an aim to study the software energy consumption behaviour. We consider an example of an ATM system. Here only a small set of a typical ATM system classes has been considered. Three cases are discussed with the help of an ATM example to figure out the final code smell elimination permutation that could possibly lead to low energy consumption as compared to other permutations. There are two main classes in this example, class ATM and class Account. Class ATM performs several functions such as getBalance(), getReceipt(), getMiniStatement(), withdraw(), fundTransfer(). In this example, function fundTransfer() is considered as long method as well as method which is responsible for feature envy code
smell. It creates feature envy by accessing the more data of class Account than from class ATM to complete its functionalities. Class Account performs functions such as deposit(), withdraw() and createTransaction(). In this example class ATM is considered as god class for Cases 1 and 2 and class Account is considered as god class for Case 3. Case 1 compares and analyzes the removal of long method code smell and feature envy code smell, Case 2 compares and analyzes the removal of long method code smell and god class code smell and Case 3 compares and analyzes the removal of feature envy code smell and god class code smell. Afterwards results of all three cases are combined to get the final sequence of removing code smells which is expected to increase the energy efficiency of software applications.

**Case 1**: Analysis, mentioned next, stands over the assumption that removing feature envy code smell does not induce more feature envy code smells in any of the classes. Figure 1.1 and 1.2 represents an ATM system which initially has two classes ATM and Account. Class ATM contains a long method (fundTransfer()), which is also responsible for feature envy code smell as it accesses data from class Account. As shown in Figure 1.1 removing long method code smell by applying extract method refactoring creates one extracted method (fundTransferR()) in class ATM. Class ATM now has one more method fundTransferR() thus eliminating the long method code smell. Some of the functionalities of fundTransfer() are now being performed by fundTransferR(). Then by removing feature envy code smell by applying move method refactoring, there arises the possibility that fundTrasnferR() moves to class Account whereas fundTransfer() remains in class ATM (refer Figure 1.1). Now, in order to access fundTransferR(), fundTransfer() has to exchange extra messages hence introducing additional overhead and possibly increasing energy consumption. For the sake of removing this overhead of extra messaging, feature envy code smell must be eliminated first, by applying move method refactoring, thus moving fundTransfer() to class Account (see Figure 1.2) and then removing long method code smell, as a result both methods fundTransfer() and fundTransferR() remains in the same class i.e. class Account (see Figure1.2). As both functions are in the same class, therefore number of messages required for interaction between them would be less, thus making this order of removing code smells more energy efficient. It is now evident from Figures 1.1 and 1.2 that removing feature envy code smell by applying move method refactoring before long method code smell, which is eliminated by applying extract method refactoring, is expected to consume less energy, thus generating energy efficient software version comparatively.
Figure 1.1 Removal of feature envy after long method - ATM Example

Figure 1.2 Removal of long method after feature envy - ATM Example
Case 2: Class ATM is a god class which contains a long method fundTransfer() (see Figure 1.3). Removing long method code smell, by applying extract method refactoring, creates one extracted method fundTransferR() in class ATM. Class ATM now has one more method fundTransferR() thus eliminating the long method code smell. Next, removing god class code smell by applying extract class refactoring extracted class ATMProduct is created which contains some of the functionalities of class ATM. In the process of creating class ATMProduct, possibility is one of the methods (fundTransfer() or fundTransferR()) moves to class ATMProduct and one of them remains in class ATM, thus creating the same situation as is created in Case 1 (see Figure 1.3). Hence, to avoid this energy wasteful situation god class code smell must be removed first by applying extract class refactoring thus creating an extracted class ATMProduct and then removing long method code smell. As a result both methods (fundTransfer() and fundTransferR()) remain in same class i.e. either in class ATM or in class ATMProduct (see Figure 1.4). From the above discussion it is expected that refactored version obtained by removing long method code smell after eliminating god class and feature envy code smells would consume less energy.

Case 3: Consider a class ATM has method fundTransfer() which is responsible for feature envy code smell. It accesses data from class Account and class Account is a god class. Removing feature envy code smell, by applying move method refactoring, moves method fundTransfer to class Account (see Figure 1.5). Next, eliminating god class code smell by applying extract class refactoring on class Account creates extracted class Transaction (see Figure 1.5). In this process of eliminating god class code smell, method fundTransfer() may further move to newly created extract class Transaction (see Figure 1.5) but actually fundTransfer() still accesses the data from class Account, now method fundTransfer() will generate extra messages to access the data from class Account thus introducing overhead and possibly increasing energy consumption. While on the other hand if god class code smell is removed first and then feature envy code smell (see Figure 1.6), then in that case extracted class Transaction is created first and then method fundTransfer() is moved to class Account, thus preventing the generation of extra messages and function calls and making the code energy efficient. It is now evident from Case 3 that refactored version created by eliminating feature envy code smell after the elimination of god class code smell is expected to consume less energy.
Figure 1.3 Removal of god class after long method-ATM Example

Figure 1.4 Removal of long method after god class-ATM Example
Figure 1.5 Removal of god class after feature envy-ATM Example

Figure 6. Removal of feature envy after god class- ATM Example
Above discussion (Case 1, Case 2 and Case 3) gives complete insight of why removing code smell in GFL order must yield minimum energy consumption.

1.7 Research Objectives

In this study, following are the objectives to be addressed:

Objective 1: To analyze the energy consumption trends of software applications after the elimination of three of the code smells, god class, long method and feature envy, individually.

Objective 2: To analyze the energy consumption trends in software applications after the elimination of three of the code smells (god class, long method, feature envy), in different permutations.

Objective 3: To analyze the variation in architectural metrics as a consequence of eliminating code smells (god class, long method, feature envy) individually and in different permutations, to identify relationship for the same with energy consumption values.

1.8 Thesis Outline

The rest of this thesis is organized as follows:

- Chapter 2 presents the previous work related to code smells, refactoring techniques, energy efficient software applications, effect of design patterns on energy efficiency, effect of refactoring on energy efficiency, approaches defined for energy consumption measurement of software and survey on energy related tools.
- Chapter 3 describes methodology and implementation including sample applications under study, selected code smells, tools used in this study and methodology.
- Chapter 4 presents the experimental results and their analysis. It elaborates the analysis of experimental outcomes.
- Chapter 5 concludes the study and highlights some possible future directions.
CHAPTER 2

LITERATURE REVIEW

Extensive research effort in both academia and industry has been directed toward analysing the effects of code smells and refactoring techniques on various software quality attributes such as maintainability, readability, change proneness etc. This chapter describes the studies representing the impacts of code smell and refactoring on quality attributes and energy efficiency of software applications. It also introduces the studies depicting the impact of design patterns on energy efficiency of software and different approaches to measure software energy consumption.

2.1 Code Smells and Impact Analysis

Code smells refer to poor code structure that decreases the quality of software applications such as maintainability, readability etc. Fowler [5] has introduced twenty two types of code smells whose description is given in Table 2.1. Khomh et al. [49] analyzed the impact of code smells on change proneness of classes in which they are present. They performed this analysis on various releases of eclipse and azureus. Results revealed that classes containing code smells are more prone to change than the ones without code smells.

Olbrich et al. [50] analyzed the impact of two code smells, god class and shotgun surgery, on the historical data of two open source applications. They observed that components that are infected by code smells exhibit abnormal change behavior.

Studies [69-73] examined the impact of code smells on maintainability. They [69-73] performed an analysis to determine the appropriateness of code smells in recognizing code sections that have low maintainability. It is realized that maintainability is dependent on different perspectives.

Van Emden and Moonen [46] recommended an approach to detect and visualize code smells automatically. Based on the approach recommended by them, they developed jCOSMO which is a prototype for detecting and visualizing code smells for source codes written in java. Further they validated this prototype using a case study.
Marinescu [47] proposed “factory strategy”, a quality model for code smell detection. Unlike, traditional methods which detect the problems in design of source code based on software metrics, it detects the flaws in design of source code based on detection strategies which include design principles, heuristic and rules. It provides developer a convenient way to detect bad code design.

<table>
<thead>
<tr>
<th>Code smell</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long Method</td>
<td>Method having too many lines of code, making it hard to read and understand.</td>
</tr>
<tr>
<td>God Class</td>
<td>A class is too large and has lots of responsibilities.</td>
</tr>
<tr>
<td>Primitive Obsession</td>
<td>Use of primitive data types instead of ValueObjects ex. Using String to represent messages.</td>
</tr>
<tr>
<td>Long Parameter List</td>
<td>Methods containing more number of parameters.</td>
</tr>
<tr>
<td>Data Clumps</td>
<td>Primitive values that are always used together.</td>
</tr>
<tr>
<td>Switch Statement</td>
<td>Complex and similar switch statements dispersed all through the program creating duplication of code.</td>
</tr>
<tr>
<td>Temporary Field</td>
<td>Variable is used only under certain circumstances.</td>
</tr>
<tr>
<td>Refused Bequest</td>
<td>Sub class uses only a few methods inherited from its super class and rest of the methods go unused.</td>
</tr>
<tr>
<td>Alternative Classes With Different Interfaces</td>
<td>Two classes have methods having same functionalities but different signatures.</td>
</tr>
<tr>
<td>Divergent Change</td>
<td>Several changes are required to be done in a class to incorporate modifications.</td>
</tr>
<tr>
<td>Shotgun Surgery</td>
<td>To incorporate single change, multiple classes need modifications.</td>
</tr>
<tr>
<td>Parallel Inheritance Hierarchies</td>
<td>Sub class made for one class enforces the need to make sub class for another class.</td>
</tr>
<tr>
<td>Comments</td>
<td>Descriptive statements for code.</td>
</tr>
<tr>
<td>Duplicate Code</td>
<td>Identical code architecture in different methods.</td>
</tr>
<tr>
<td>Lazy Class</td>
<td>A class that performs too less.</td>
</tr>
<tr>
<td>Data Class</td>
<td>Class comprises of fields and raw methods to access them. It is data container accessed by other classes and cannot operate independently.</td>
</tr>
<tr>
<td>Dead Code</td>
<td>Section of source code that is no longer executed.</td>
</tr>
<tr>
<td>Speculative Generality</td>
<td>Code section that is never implemented but created for future use</td>
</tr>
<tr>
<td>Feature Envy</td>
<td>Method belonging to one class makes extensive use of another class.</td>
</tr>
<tr>
<td>Inappropriate Intimacy</td>
<td>Class accesses methods and fields of another class.</td>
</tr>
<tr>
<td>Message Chains</td>
<td>Series of function calls. Depicts dependency of one object on another.</td>
</tr>
<tr>
<td>Middle Man</td>
<td>Class delegating most of its functionalities to another class.</td>
</tr>
</tbody>
</table>
Castillo et al. [24] investigated the impact of removing god class code smell on two open source software applications and tried to devise a relationship between architectural metrics and power consumption of software system. Results from their study revealed that removing god class increases power consumption, LOC, number of classes, number of methods, couplings and number of messages exchanged whereas decreases the cyclomatic complexity.

2.2 Refactoring and Impact Analysis

2.2.1 Refactoring

Refactoring is believed to improve the productivity of developer as well as quality of software. Tourwé and Mens [48] conducted a survey in area of software refactorings which includes refactoring techniques that are encouraged, tools that support these activities, concerns while developing refactoring tool, impact of performing refactoring on software applications and type of code structure that must be refactored.

Kannangara et al. [51] investigated the impact of applying refactoring techniques on various internal quality attributes such as maintainability index, cyclomatic complexity, depth of inheritance, etc., and external quality attributes such as resource utilization, time behaviour, etc. They observed improvement in maintainability index whereas no effect was observed on other internal quality attributes. According to them external attributes were unaffected i.e. no improvement was observed after applying refactoring.

Kim et al. [16] quantitatively evaluated the advantages of performing refactorings on software systems from the view point of developers’ perspective. They followed three different and complementary methods to analyze the perspective of developers about software refactoring. In first method they conducted a survey on software engineers employed at Microsoft, which comprises of twenty two questions related to their job role and code refactoring experiences. Those twenty two questions include questions about their experiences in software industry, various refactoring techniques used, refactoring definitions, refactoring tools available and used in industry etc. Results from this survey revealed that developers find refactorings highly beneficial. These benefits include improved understandability, maintainability, readability, ease to add more features of code etc. However refactorings involve considerable cost and time. In second method which is semi structured interview, they conducted one to one interviews with professional software engineers and those who have been involved in
refactoring Windows 7. They figured out that decision of performing refactoring is centralized and is carried out after properly analyzing the dependency structures. In order to refactor software system, developers use customized refactoring tools. Authors statistically analyzed the impact of refactoring on various software quality attributes. For this purpose, they analyzed the changes in metric values among different versions of Windows 7, and demonstrated that refactored versions indicate the decrease in inter module dependency and complexity values and significant increase in the size of application.

Du Bois and Mens [74] analyzed the impact of refactoring on software metrics such as NOM, NOC, CBO, RFC, LCOM. Du Bois et al. [75] analyzed the impact of refactoring techniques on coupling and cohesion of software applications.

Leppanen et al. [17] investigated the relevance of refactoring according to developers. They conducted semi structured interviews in nine different companies with twelve software developers in different sessions. Results from their study indicate that refactoring is of great importance and according to developers it makes software easy to understand and maintain and new features can be added easily, however the primary attention of industries remained towards delivering more features rather than on restructuring the previously written code. Table 2.2 indicates the benefits and risks of performing refactorings according to interviewees.

<table>
<thead>
<tr>
<th>Benefits</th>
<th>Risks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy development in future</td>
<td>Breaking already structured code</td>
</tr>
<tr>
<td>Improved understandability</td>
<td>May cause externally visible modifications</td>
</tr>
<tr>
<td>Easy to reuse</td>
<td>May degrade the code quality</td>
</tr>
<tr>
<td>Improved performance</td>
<td>Wastage of time and effort.</td>
</tr>
</tbody>
</table>

2.2.2 Impact of Refactorings on Energy Efficiency

Refactoring techniques change the internal behavior of software, and hence impacting the energy consumption behavior of software systems. Pinto et al. [26] conducted a survey reflecting the current research in the field which relates refactoring to software energy
consumption. They discovered increasing attention towards energy efficient software development with energy efficiency of mobile applications possessing more number of opportunities. They identified four categories, mobile applications, parallel/concurrent programming, approximate programming and DVFS techniques, where refactoring can be highly beneficial.

In mobile devices, the only source of power supply is batteries which are limited in capacity due to their small sizes. This drives the need for mobile applications to be energy efficient. According Li et al. [34], using dark colors for display consumes less energy as compared to lighter ones, so refactoring techniques are written to automatically change dark colors to their light colored counterparts. Kwon et al. [35] suggested that CPU offloading can be done from mobile to cloud for CPU intensive applications in order to reduce battery usage. However, it incurs additional energy consumption due to GSM and Wi-Fi. There is a trade off in using this technique; this can only be beneficial if energy savings by offloading outweighs the energy cost of networking.

Li et al. [36] discovered that classes handling HTTP requests consume more energy; corrective energy efficient refactorings must be discovered in order to make the HTTP requests energy friendly. Code obfuscation techniques are used to prevent software piracy which in turn increases energy consumption of software. Sahin et al. [37] reported 2.1% increase in energy consumption due to code obfuscation. According to Banerjee et al. [38], third party advertisements are one of the main sources of energy consumption in mobile devices, which keep the GPS working even after exiting the application. They applied refactorings on application to run third party advertisements in asynchronous thread instead of main thread, which can decrease energy consumption of software application. Continuously running applications exchange data between server and mobile devices periodically. Nikzad et al. [39] has proposed a refactoring technique to save the energy and to maintain the integrity of software application.

To take advantage of multicore technology, applications must be in executed parallel/concurrently. Fork Join java framework [59] provides solution for completely independent recursive programs. In recursion, data gets copied in every recursive call. Pinto et al. [40] observed that sharing data instead of copying saves energy up to 15.38 %. Kambadur and Kambadur et al. [41] showed that parallel counterparts of sequential versions are more energy
efficient and refactoring techniques can be developed to convert sequential programs into parallel ones so as to develop energy efficient applications. Scanniello et al. [42] explored that Graphical Processing Units can perform more work in parallel as it can execute more threads simultaneously, thus increasing energy efficiency. They proposed a scheme to convert CPU intensive systems to GPU (Graphical Processing Unit) based architecture and observed 60% savings in energy.

Misailovic et al. [43] observed that applying approximate programming techniques increase energy efficiency of applications. They introduced Chisel, an approximate programming framework and observed the energy savings from 8.7% to 19.9%.

Dynamic Voltage and Frequency Scaling (DVFS) is a feature of CPU in which voltages and frequencies of CPU can be changed dynamically. Cohen et al. [44] recommended a new type of system in which DVFS calls are automatically instrumented and compiler decides whether to scale up or scale down the frequencies to reduce energy consumption. They recorded up to 50% energy savings. Bartenstein et al. [45] reported energy savings in stream line programming by proposing an approach in which output of one filter is produced “just in time” for another filter’s consumption. They recorded the energy savings up to 28%.

Existing refactoring techniques were designed to focus on extending the life time of software applications by improving software quality attributes such as maintainability, readability, understandability etc. As these refactoring techniques do not take into account the energy consumption of software system to be refactored, it becomes necessary to analyze their impacts on energy consumption behavior of software system. Park et al. [23], Sahin et al. [30], analyzed why and how does various refactoring techniques impact energy efficiency of software systems, by measuring the energy consumption of software before and after performing refactorings.

Park et al. [23] analyzed the impact of applying different refactoring techniques on energy consumption behavior of software system. For their study, they took into account sixty three refactoring techniques defined by Fowler [5]. Sample source codes were available as open source codes and they developed refactored versions of open source software codes manually. To estimate power consumption they used XEEMU, a software tool for power estimation. Results from their study exposed the fact that refactored versions created by applying
refactoring techniques exhibit different energy consumption behavior. They categorized these techniques into three groups, one which has positive impact on energy efficiency, second which has no impact on energy efficiency and third which has negative impact on energy efficiency. Twenty six refactoring techniques fall in group one, seven techniques fall in group two and others fall in group three.

Sahin et al. [30] investigated the impacts of applying six refactoring techniques on nine open source java applications of varied sizes and domains. After applying refactorings, they generated 197 refactored versions whose energy consumption is measured using Leap power profiler. 9 applications and 197 refactored versions were executed on two different platforms i.e. JVM 6 and JVM 7. Results from their study indicated that implementation of refactoring techniques affect energy consumption behavior of software applications and their impact was found to be inconsistent across software applications as well as across different platforms (JVM 6 and JVM 7).

Gottschalk et al. [6–8] and Jelschen et al. [14] addressed the issue of optimizing energy efficiency of mobile applications by analyzing code for energy inefficient code fragments and replacing it by energy efficient code using refactoring and reengineering services such as dynamic analysis and refactoring for monitoring, analyzing and optimizing the energy profiles of mobile devices. Gottschalk et al. [7] used two mobile devices to evaluate the impact of refactorings on energy consumption behavior of two applications GPSPrint and TreeGenerator. For analyzing the impact of refactoring techniques, Binding Resources Too Early and Third-Party Advertisements, GPSPrint is used; and TreeGenerator is used to analyze Backlight and Data Transfer, Statement Change and Third-Party Advertisement refactoring technique. Refactored versions of above applications are generated and energy consumption is measured by executing them on two different mobile devices using three different energy measurement techniques for validation. Results revealed that some refactorings increased the energy consumption and some decreased the energy consumption. Gottschalk et al. [8] worked on various refactoring techniques and investigated their impact on energy consumption values of mobile devices and they proved that it is possible to detect code smells using GreQL.

Hindle [12] analyzed how power consumption changes across different versions of two applications Mozilla Firefox and BitTorrent. He observed considerable changes in power
consumption across versions and tried to relate this power consumption behavior with metrics like Number Of Children (NOC) and Depth Of Inheritance (DIT). He observed that this relationship greatly depends upon test case structure to be executed. Hindle [13] analyzed how power consumption varied across different versions of Firefox web-browser and related some software metrics to software power consumption. He analyzed more than 500 versions of Firefox web browser and investigated the changes in various metrics and power consumption values. It was observed that power consumption was not related to software metrics such as LOC and apart from CPU, disk and memory were also responsible for power consumption.

Pinto [25] aimed to gain in depth knowledge about the correlation between energy usage and parallel programming constructs and to propose novel tools and techniques to refactor concurrent programs in order to increase energy efficiency. For investigation they focused on two widely used languages Java and Scala.

Reimann and Aβmann [27] defined an approach to explicitly derive relationship between quality which includes software energy consumption, code smells and refactorings, as a result of which specific qualities can be focused in early phases to detect energy deficiencies. They came up with a novel term “quality smell”, and also introduced meta models for a repository of quality smells and its calculations, which in turn is expected to aid developers in establishing relationship between quality, code smells and refactorings. They also evaluated the energy efficiency of java codes.

Silva et al. [4] and Vetro et al. [32] studied the software energy consumption trends in embedded software. Silva et al. [4] analyzed how performance and energy of embedded software changes as a result of applying inline method refactoring. They investigated the changes in performance and energy efficiency using three java applications and used dynamic profiling to get the count of method calls and then accordingly applied inline method refactoring. For energy and performance measurement purposes, they used DESEJOS, an estimation tool. Increase in performance and decrease in energy consumption was expected after applying inline method refactoring technique. However, two applications satisfied this assumption while the remaining one exhibits contradictory results. Vetro et al. [32] analyzed energy impacts of different code patterns on embedded software. The code patterns they selected for their analysis are the patterns defined in two static analysis tools: CppCheck and
FindBugs. They implemented the programs in C++, so they implemented relevant patterns identified from FindBugs in C++. To measure the power consumption, they built a circuit (power meter) and connected it to WASP (the hardware architecture or embedded system used in experiment). They defined a new term “energy smells” which refer to the section of those code structures which decrease the energy efficiency of software programs.

Table 2.3 Potential energy smells

<table>
<thead>
<tr>
<th>Potential Energy Smell</th>
<th>Pattern Description</th>
<th>Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter by Value</td>
<td>Using call by value for parameter passing.</td>
<td>CppCheck</td>
</tr>
<tr>
<td>Self Assignment</td>
<td>Assigning variable to itself. (e.g., x=x)</td>
<td>CppCheck</td>
</tr>
<tr>
<td>Mutual Exclusion OR</td>
<td>Performing OR operation with two mutually exclusive events as operands (thus always true).</td>
<td>CppCheck</td>
</tr>
<tr>
<td>Switch Redundant Assignment</td>
<td>Assigning a value to a variable in a case block without a following break instruction, then re-assigning another value to it in the subsequent case block.</td>
<td>CppCheck</td>
</tr>
<tr>
<td>Dead Local Store Return</td>
<td>Statement assigning value to a variable which is never used.</td>
<td>FindBugs</td>
</tr>
<tr>
<td>Dead Local Store Return</td>
<td>A return statement assigning a value to a local variable, which is never used.</td>
<td>FindBugs</td>
</tr>
<tr>
<td>Repeated Conditionals</td>
<td>Evaluating a condition twice.</td>
<td>FindBugs</td>
</tr>
<tr>
<td>Non Short Circuit</td>
<td>Conditions evaluating non-short-circuit logic boolean operators (e.g., &amp; or k) rather than short-circuit logic ones (&amp;&amp; or kk).</td>
<td>FindBugs</td>
</tr>
<tr>
<td>Useless Control Flow</td>
<td>e.g., an if statement with an empty body.</td>
<td>FindBugs</td>
</tr>
</tbody>
</table>

Table 2.3 presents the potential energy smells and their description. They also analyzed whether these code patterns affect software performance or not. Results from their study revealed that energy code smells are not performance smells i.e. they did not introduce a decrease in performance.

2.3 Impact of Design Patterns on Energy Efficiency of Software

Bunse and Stiemer [2] and Sahin et al. [28,29] empirically examined the impact of software design patterns on the energy consumption of software applications. They observed that some
patterns improved energy efficiency of software systems, some patterns have no significant impact on energy consumption behavior of software systems and some patterns tend to decrease energy efficiency of software systems. Bunse and Stiemer [2] performed the study on mobile devices that use java and develop java based applications for each pattern (one with pattern and one without pattern). They analyzed the impact of six different patterns (template method, facade, observer, prototype, decorator and abstract factory) on software energy consumption behavior using Power Tutor App. Their results revealed that some patterns show no difference in energy needs while others show the large differences in energy values and execution times.

Sahin et al. [28, 29] worked towards mapping energy consumption of software application to high level design properties so as to assist developers, software designers to build energy efficient software. For energy measurement purposes, they developed their own tool, which they believed to be more accurate than existing ones and thus provided more accurate energy values. Their experiment was carried upon Spartan-6 FPGA, an embedded system. Results showed that their tool was able to detect differences in energy consumption behavior of software systems. They measured the energy usage of software with and without having design patterns (creational, structural and behavioral), and observed that design patterns belonging to same category do not exhibit similar energy consumption trends. Some increased the energy consumption, some had no significant impact on energy consumption, while others decreased it. They observed a general trend between energy consumption and high level artifacts with few exceptions.

Noureddine and Rajan [20] depicted compiler level energy optimization for software design patterns that are believed to exhibit negative impact on software energy consumption. They performed energy efficient transformations for observer design pattern and decorator design pattern at compiler level, and validated their results using open source applications from github of varied sizes. It is observed that their transformations saved energy efficiency to some extent, and energy savings were observed to be dependable on the count of design patterns present in software system.

Banerjee and Roychoudhury [1] worked on increasing the energy efficiency of smart phones by introducing energy aware patterns for smart phone applications. The design patterns proposed by them can be applied to real life situations to reduce energy consumption. Two
proposed design patterns are “frugal information gathering” and “approximated information processing”.

2.4 Approaches for Developing Energy Aware Software

Carção [3], Hao et al. [10], Josefiok [15], Shorin and Zimmermann [31] focussed on different approaches to deal with the issue of optimized software energy consumption. Carção [3] proposed a novel approach to identify energy leaks in source code. This approach includes spectrum-based fault localization technique to identify energy leaks in program. This technique uses run time trace of program to identify errors. They defined a term “red smell” to indicate the energy leak situation in source code and intend to propose green refactoring catalogue to deal with red smells. They used Intel Power Gadget [61] for energy measurements. For validation purposes they used two different programming languages: textual such as C, C++,Java and visual such as Lab VIEW environment [60].

Hao et al. [10] presented an approach to estimate energy of mobile application combining energy modeling per instruction and program analysis. They called this new technique to measure energy consumption as eLens and validated it on six different applications. It measures the energy consumption at finer levels of granularity such as class, method and statement. Energy values calculated using this technique are accurate within 10%.

Josefiok [15] proposed a measurement infrastructure that must facilitate the measurement of energy consumption values of mobile devices. This measurement infrastructure is known as Energy Abstraction Layer. It needs to access discharging current and voltage for energy estimations. For voltage estimations, Android API methods were used; while for measuring discharging current, three different techniques were used. First one is Delta-B which compares capacity of battery with changes in charge level of battery, second one uses file system to read the value of discharging current, and third one straddles upon the energy profiles given by vendor. They have implemented the prototype of given approach which fulfills the central functionalities of Energy Abstraction Layer.

Shorin and Zimmermann [31] proposed a methodology that used petrinets to evaluate energy values of embedded systems. It involves modeling applications and operations to Unified Modeling Language (UML) which extended MARTE profile and then transform the two into
stochastic pertinents for analysis of energy consumption behavior. This methodology is validated by actual lab set ups.

Naumann et al. [19] proposed a model “greensoft model” to support green software engineering and sustainability. This is a two-level model with first level suggesting green engineering practices which straddles upon other development processes like agile, sequential and iterative methods. First level also includes green guidelines and green metrics at each development stage to enforce the development of green software product. Level two of the model recommends a technique of using software product as a tool to help in green development by keeping track of resources being used in energy friendly manner.

Gupta et al. [9] described various approaches to analyze energy defects on Windows phone platform to equip engineers with the techniques to analyze power consumption. They analyzed Windows 7 to figure out modules consuming more energy and hence detecting energy wasteful code designs which can further be used by developers to introduce new energy friendly code designs, usage scenarios etc.

2.5 Chapter Summary

- Refactoring techniques are considered highly beneficial in industries and developers prefer performing refactorings to aid in maintainability and extended life time of software.
- It has been observed from above studies that modifying source code of software have huge impact on its energy consumption behavior, thus for software to be energy efficient refactorings must be performed keeping in mind energy as one of its quality attributes.
- It has been shown that different design patterns exhibit different energy consumption behavior and efficiency of design patterns can also be optimized.
- Various energy aware refactorings and design patterns have been proposed which can be incorporated in industries for the development of energy aware products.
- A study has recommended a model to make the product as well as development process energy aware.
CHAPTER 3

EXPERIMENTAL DESIGN AND METHODOLOGY

This chapter describes the characteristics of sample applications under study, selected code smells and software metrics to be calculated, software tools used for various purposes such as identification and elimination of code smell, energy measurement etc., and methodology followed to conduct the experiments.

3.1 Sample Applications

For analysis, three open source java applications are considered and the effect of individual as well as different sequences of code smell elimination are examined on software energy consumption and architectural metrics. Table 3.1 presents the specifications of selected applications including name of application, version number, number of classes, number of JUnit test cases and test case coverage.

<table>
<thead>
<tr>
<th>Application</th>
<th>No. of Classes</th>
<th>No. of Test Cases</th>
<th>Coverage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JHotDraw 6.2.0</td>
<td>552</td>
<td>3654</td>
<td>54.7</td>
</tr>
<tr>
<td>Commons Beanutils</td>
<td>322</td>
<td>1660</td>
<td>78.3</td>
</tr>
<tr>
<td>Commons IO 2.6</td>
<td>262</td>
<td>1157</td>
<td>91.9</td>
</tr>
</tbody>
</table>

Following is the description of applications:

- **JHotdraw** [62]: JHotDraw is a two-dimensional Java GUI framework for technical and structural graphics. It provides basic structure for GUI editor with distinct views, tool palette, and ability to load, save and print etc. It can be customized by combining components and using inheritance.

- **Commons Beanutils** [63]: It provides low level utility classes that assist in getting and setting property values on Java classes that follow the naming design patterns outlined in the JavaBeans Specification, as well as mechanisms for dynamically defining and accessing bean properties.
• **Commons IO** [64]: It is a utility library to aid in developing IO functionality. It has six pre-eminent functionalities which include: Input which implements Reader and Input stream, Output which implements Writer and Output stream, File monitor which monitors events of file system, Utility classes which has static methods for implementing common functionalities and Comparators for implementing java.util.comparator.

These applications are selected as they are of different domains and sizes (number of classes) which can help in generalizing the results. Also as we require test cases to run these applications, so the sample applications with their test cases available were selected. Test cases are believed to cover at least the core functionalities of a software system, and hence are believed to be useful criteria for the selection of sample applications for this work.

### 3.2 Selected Code Smells and Refactoring Techniques

Code smells [5] are certain structures in the code that indicate violation of fundamental design principles and hence negatively impact design quality. The impact of three most commonly present code smells is investigated on energy consumption in our study. Following is the brief description of these code smells:

- **Feature Envy**: This smell occurs when method belonging to one class makes extensive use of another class.
- **God Class**: Presence of this smell indicates the violation of single responsibility design principle. It occurs when class is too large and has lots of responsibilities.
- **Long Method**: This smell occurs when a method has too many lines of code, making it hard to read and understand.

While developing software, code smells may go unattended by the developers; hence in such cases tool support for code smell detection and removal is primarily useful. In this study, JDeodorant [65], an eclipse plugin, is used to identify and eliminate code smells. It applies “move method” refactoring to remove feature envy, “extract class” refactoring to remove “God class” and “extract method” refactoring to remove “long method”.

The description of refactoring techniques applied by JDeodorant (see Section 3.4.1) is as follows:
- **Move Method**: This refactoring technique creates a new method in the class to which the original method is more frequently accessing. Then move the code from original method to this newly created method. Lastly, modify the code of the original method into a reference to the new method in the other class (which is being frequently accessed).

- **Extract Class**: This refactoring technique extracts some of the functionalities of original class to newly created class.

- **Extract Method**: This refactoring technique extracts some functionality of original method to newly created method and replace the old code with a call to newly created method.

### 3.3 Selected Metrics

Following is the description of architecture metrics that are calculated in this work. All these are static metrics:

- **Lines Of Code (LOC)**: This software architecture metric evaluates the size of source code by computing number of lines written in source code.

- **Number of Methods**: This software metric computes the total number of methods contained in the source code of software.

- **Number of Classes**: This metric computes the total count of classes present in the source code of software.

- **Coupling Between Objects (CBO)**: This metric presents the count of classes coupled to a particular class. This coupling arises due to exceptions, arguments, method calls, return type, field access and inheritance.

- **Cyclomatic Complexity (CC)**: This metric computes the complexity of source code and is used in white box testing.

Lines Of Code (LOC), number of methods, number of classes are computed using Atlassian’s clover tool (see Section 3.4.2), and cyclomatic complexity is calculated using
3.4 Tools Used

This section describes various software tools used in this study. It describes the tool used to detect and eliminate code smells, to calculate various metrics and to measure energy consumption of software applications.

3.4.1 JDeodorant

It is an eclipse plugin used to figure out the design problems known as code smells in software. It exercises several novel techniques to find out and eliminate code smells by applying appropriate refactoring techniques. Table 3.2 specifies the code smells identified by JDeodorant [65] and corresponding refactorings to resolve them.

<table>
<thead>
<tr>
<th>Code Smell</th>
<th>Refactoring Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Envy</td>
<td>Move Method</td>
</tr>
<tr>
<td>Type Checking</td>
<td>Replace Conditional with Polymorphism and Replace Type code with State/Strategy</td>
</tr>
<tr>
<td>Long Method</td>
<td>Extract Method</td>
</tr>
<tr>
<td>God Class</td>
<td>Extract Class</td>
</tr>
<tr>
<td>Duplicated Code</td>
<td>Extract Clone</td>
</tr>
</tbody>
</table>

In this study, three code smells (god class, long method and feature envy) are considered, following discussion includes the working of JDeodoarant to identify and eliminate code these three smells. It is an automated tool to identify code smells and to apply appropriate refactoring techniques to eliminate code smell. JDeodorant equips the user with “code smell visualization” window in order to view pictorially the modifications in source code (i.e. code structure of the application after applying refactoring). Figure 3.1 demonstrates the visualization window for a single instance of god class code smell.
In Figure 3.1, org.jhotdraw.applet.DrawApplet is the source class which is also the god class, extracted class is the one which will be created after applying extract class refactoring. It represents the functionalities or responsibilities that will be assigned to extracted class beforehand.

Figure 3.2 shows the visualization window for feature envy code smell. As shown in Figure 3.2, write() is a method which is defined in org.jhotdraw_figuresAttributes but write() method is accessing more data from class org.jhotdraw.util.StorableOutput than the original class in which it is defined i.e. org.jhotdraw.figures.Attribute; thus causing feature envy code smell. In order to remove feature envy code smell, write() method must be moved to class org.jhotdraw.util.StorableOutput by applying move method refactoring.
Long method code smell cannot be represented using code smell visualization window because this code smell does not involve multiple classes. Hence to demonstrate how the JDeodorant deals with long method code smells screen shots of source code are taken. Figure 3.3 represents the source code with setSelected() method as long method and Figure 3.4 represents the source code with eliminated long method code smell. Extract method refactoring is used to eliminate long method code smell. In extract method refactoring technique, a new extracted method (setSelectedR()) is formed which performs some of the functionalities of method setSelected() and a function call to method setSelectedR() is added in method setSelected() (refer Figure 3.4). As can be observed from Figures 3.3 and 3.4, number of methods increase with the elimination of long method code smell. This refactoring technique makes the function easy to manage, maintain, read and understand.
Figure 3.3 Illustration of long method code smell

Figure 3.4 Eliminated long method code smell
3.4.2 Atlassian Clover

Atlassian clover coverage tool [66] is used in this study to identify the percentage coverage of application by automated test cases and to calculate the architecture metrics such as Lines of Code (LOC), number of classes and number of methods.

a) Approaches To Measure Coverage

Several approaches exist to measure the coverage of computer programs. They can be classified broadly into three categories:

- **Instrumentation of source code**: In this category, source code is first instrumented and then compiled to create instrumented compiled code.

- **Bytecode Instrumentation**: This approach generates instrumented class files by adding instrumentation bytecode statements in them.

- **Dynamic collection of information**: This approach collects the coverage information during the code execution from dynamic environment.

Atlassian Clover Coverage tool first instruments and compiles the source code and then executes it to collect the coverage information.

b) Types Of Coverage

Atlassian Clover Coverage tool performs three elementary types of coverage analysis:

- **Statement**: This type of coverage analysis returns the percentage of statements executed.

- **Branch**: This type of coverage analysis keep tracks of which conditional branch of the control flow structure is executed.

- **Method**: This type of coverage analysis returns the percentage of methods executed.

All the three measurements are utilized by clover to compute Total Coverage Percentage (TPC) of the classes, packages and whole program. TPC is calculated as follows:

\[
TPC = \frac{(BT + BF + SC + ME)}{(2*B + S + M) * 100}\% , \text{ where}
\]
Experimental Design and Methodology

BT - branches that evaluated to "true" at least once
BF - branches that evaluated to "false" at least once
SC - statements covered
ME - methods entered

B - total count of branches
S - total count of statements
M - total count of methods

3.4.3 Code Pro Analytix

Code Pro Analytix [67] is a freeware available for free downloading by google. This is a java based software testing and quality control tool having following features:

- Analysis of static code.
- Defect detection, reporting and repairing.
- Auditing for Struts, JSP, XML and Java.
- Generate JUnit test cases automatically.
- Computes static code metrics such as cyclomatic complexity, abstractedness, number of semicolons, number of constructors, efferent coupling, weighted methods etc.

Figure 3.5 Metric values using atlassian’s clover tool
It can easily be integrated with WebSphere, Eclipse, Rational and MyEclipse.

This tool is used to compute the cyclomatic complexity of all the refactored versions generated in this work. Figure 3.6 shows the metric values calculated using code pro analytix. In the Figure 3.6 values for abstractedness, average block depth represent the metric values of whole program whereas average complexity metric value is shown package wise.

![Figure 3.6 Metric values using code pro analytix tool](image)

### 3.4.4 ckjm

This tool calculates all the metrics defined by Chidamber and Kemerer [68] and few others. It computes the class level metrics values from the generated bytecode. Metrics calculated by ckjm tool are: Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number Of Children (NOC), Coupling Between Objects (CBO), Response For a Class (RFC), Lack Of Cohesion in Methods (LCOM), Afferent Coupling (Ca), Number of Public Methods for a class (NPM). Following command is used to calculate the metric values:

```java
java -jar /path/to/ckjm.jar path/to/class/file/*.class
```
This command returns the metric values for all class files residing at the specified location. To calculate the metric values for a specific class, change the ‘*’ to ‘class_name’.

This tool is used to calculate the CBO value of all refactored versions generated in our study. In Figure 3.7, fourth value represents the CBO metric values. For example CBO value is 3 for ConverUtilsBeanProduct1. Lastly, all the individual CBO values of each class are summed up to compute the CBO values of the whole software application.

3.4.5 Jalen

Several tools are available for energy measurement purposes at different levels of granularity [21] including Power Scope [52], pTop [53], Power Top [54], Energy Checker [55], Joule Meter [56], PowerAPI [57].

In this study, we use Jalen [22], a software energy measurement tool for measuring the energy consumption of software applications. It provides the energy values with high precision, at the granularity of software code (such as methods). It uses statistical sampling for measuring the energy consumption of software applications. After every 10 ms (small monitoring cycle, see Figure 3.8), Jalen checks which process id is currently being executed and stores it in a map. For each 500 ms (big monitoring cycle, see Figure 3.8), it estimates the energy consumed by the program, and distributes that energy to the method it identified statistically. We execute
each java application through its respective JUnit test cases by creating a dummy/driver class containing a “main function”, which invokes the available set of test cases. Each such driver class serves as the entry point for the respective software application in the application jar file, which is then passed to Jalen for energy measurements. Jalen is executed using following command: java  –javaagent: “Jalen.jar” –jar “our_program.jar”, where “Jalen.jar” represents the jar file of Jalen agent and “our_program.jar” represents the jar file of java application whose energy is to be monitored.

![Jalen Sampling Agent](image)

**Figure 3.8 Working of Jalen tool [22]**

Output of jalen is generated in csv format where first column represents the method name, second column represents the CPU energy consumed in Joules and third column represents the disk energy in Joules. Figure 3.8 demonstrates the working of Jalen.

Figure 3.9 demonstrates the implementation of Jalen. It represents the energy measurement of software application Commons-IOO, whose jar file (Commons-io.jar) is passed as an argument in the command line. Jalen-2.jar represents the jar file of jalen agent, here version 2 of Jalen is used.

So to summarize the working of Jalen:
1) Configure configuration file according to system’s specifications.
2) Run software application with Jalen as a Java agent.
3) When the application exits, Jalen will stop and creates a csv file containing energy values in joule for each method.

![Figure 3.9 Energy consumption using jalen](image)

**3.5 Methodology**

This section describes the step by step procedure of our study which includes the description of experimental subject creation, calculation of architecture metrics and energy consumption measurement. Figure 3.10 explains the procedure followed in our study.

**3.5.1 Experimental Subject Creation and Metric Calculation**

First step of our procedure comprises of experimental subject creation of all three applications under study. As this study focuses on finding the impact of eliminating individual code smells (god class, feature envy, long method) as well as different code smell removal sequences on energy efficiency, experimental subjects/versions are the final versions of software applications obtained after removing code smells, god class (G), feature envy (F) and long method (L) individually as well as in all six possible permutations (GFL, GLF, FGL, FLG, LGF, LFG) from original application. There are two types of experimental versions, 1) generated by eliminating all instances of an individual code smell and, 2) generated by eliminating instances of all three code smells in a particular sequence (e.g. FGL). After
generating experimental subjects, architecture metrics including LOC (lines of codes), number of classes, number of methods, CBO (coupling between objects) and cyclomatic complexity are calculated for all versions. Atlassian’s clover tool [66] is used to calculate number of classes, number of methods and LOC; CodePro Analytix [67] to calculate cyclomatic complexity; and ckjm [68] to calculate CBO.

Experimental versions are created by executing the following sequence of actions.

1. All the applications are imported in eclipse and supporting jars for each application are added.
2. Atlassian’s Clover coverage tool is used to determine the sections of selected applications covered by their respective test suites. The information collected by Atlassian is very useful as it helped us to focus only on those sections of applications that would be executed by the test suites, thus preventing us from applying refactoring in unexecuted sections as impact of refactoring in such sections would be unnoticeable.
3. Lastly, JDeodorant is used to identify and eliminate feature envy, god class and long method code smells. Hence, in this study creation, of experimental versions is performed using JDeodorant.

Figure 3.10 shows the overview of methodology followed in our study.

**Type 1 experimental subjects**: In order to create experimental subjects of this type, all the instances of each type of code smells are indentified and removed using refactorings as suggested by JDeodorant. As shown in Figure 3.10, refactored version 1 (RA1) represents this type of experimental subjects. In this study refactored versions created by eliminating feature envy code smell is termed “ApplicationName_SmellAcronym”. For instance, JHotdraw_F, JHotdraw_G and JHotdraw_L refer to the refactored versions generated after removing feature envy, god class and long method code smells respectively from the original JHotDraw application. As three applications and three code smells are considered, total count of experimental subjects of this type is nine (three experimental subjects per application).

**Type 2 experimental subjects**: Experimental subjects of type 2 are created by removing code smells from type 1 subjects i.e. RA1. In order to create these types of experimental subjects, all three code smells are removed in all six possible sequences. Hence for each such sequence,
Experimental Design and Methodology

Figure 3.10 Methodology flow
two additional refactored versions (intermediate version, RA2 and final subject version, RA3) will be generated as shown in Figure 3.10. Given a Type 1 Refactored Version (RA1), all instances of one of the two code smells left are eliminated by applying appropriate refactoring; thus creating intermediate Refactored Version 2 (RA2). Then, all instances of the remaining code smells present in RA2 are eliminated generating the desired experimental subject version, RA3 (refer Figure 3.10). A refactored version created by eliminating three selected code smells in a particular sequence (say, LFG) is denoted as “ApplicationName_LFG”. For example, JHotdraw_LFG refers to the version created by eliminating code smells in above mentioned sequence (L-F-G). As three code smells are considered, total number of experimental subjects will be equal to the number of possible permutations of three code smells. Hence, there are six experimental versions for each application, which sum up to eighteen experimental versions in total for all three selected applications. In total there are twenty seven experimental subjects (nine subjects of type 1 + eighteen subjects of type 2).

After all the experimental subjects are created, their architecture metric values are computed for analyzing the changes in architecture as a result of eliminating code smells.

3.5.2 Energy Consumption Measurement

To determine the energy consumption of twenty seven experimental versions and three original application versions, Jalen is used. Jalen is configured according to the system specifications, used to conduct experiment, by modifying the values of frequencies and voltages to those supported by our system’s CPU in configuration properties file to gather the correct energy consumption values. Each java application is executed through its respective JUnit test cases by creating a dummy/driver class containing a “main function”, which invokes the available set of test cases. Each such driver class serves as the entry point for the respective software application in the application jar file, which is then passed to Jalen for energy measurements. Jalen is executed using following command: java –javaagent: “Jalen.jar” –jar “our_program.jar”, where “Jalen.jar” represents the jar file of Jalen agent and “our_program.jar” represents the jar file of java application whose energy is to be monitored.
The energy consumption values for each original version as well as each experimental subject version is measured using Jalen’s command line facility (see Section 3.4.1). As Jalen provides energy consumption in csv format at method level, energy consumption of all methods is summed up to get the energy consumption of the whole application. Since there are small energy value fluctuations, average of 25 energy readings for all subject versions are taken in a controlled environment to increase the accuracy of results. These experiments are performed on computer system with Intel (R) Core (TM) i5-4210U CPU of 2.40GHz and 4GBs of RAM.

### 3.6 Chapter Summary

This chapter discussed the characteristics and specifications of sample applications, followed by the description of three of the code smells (feature envy, god class and long method) and their corresponding refactoring techniques. It further described the software metrics (LOC, number of methods, number of classes, cyclomatic complexity and CBO) which are computed in this work followed by the detailed description of various tools used to calculate them. Working of JDeodorant, a code smell detection and elimination tool and Jalen, an energy measurement tool is explained. At last, this chapter elaborated the methodology flow of work including subject creation, metric calculation and energy consumption measurement.
CHAPTER 4

EXPERIMENTAL RESULTS AND ANALYSIS

This chapter presents the detailed discussion of the results obtained by following the methodology described and analyzes the outcomes of this study to answer the research objectives. Detailed description of results including the analysis of type 1 refactored versions and type 2 refactored versions is presented.

Table 4.1 summarizes the code smell information related to three sample java applications along with the architectural metrics information. For JHotdraw, we identified 7.8% (43 of 552 classes) instances of god class smell, 2.3% (117 of 5008) instances of long method code smell and 17 instances of feature envy code smell. Similarly, for Commons Beanutils and Commons IO, we identified 4.6% (15 of 322 classes) and 6.8% (18 of 262 classes) instances of god class smell; 0.73% (23 of 3118 methods) and 0.83% (22 of 2648 methods) instances of long method code smell; and, 5 and 15 instances of feature envy code smells, respectively. The percentage instances of god classes are highest among those of all three code smells for all three applications.

4.1 Analysis of Type 1 Refactored Version

As explained earlier (see Section 3.5.1), type 1 refactored versions are those that are generated by eliminating all instances of only one type of code smell code smell. In total there are nine versions of this type. Table 4.1 represents the architecture metrics for the original as well as refactored versions (created after removing all instances of one type of code smell) for all applications.

4.1.1 Analysis of Feature Envy Removed Version

Feature envy code smell is eliminated by applying move method refactoring technique (see Section 3.2). It increases Lines Of Code (LOC) and number of methods, as is evident from its description in Section 3.2. Similar results are observed in our study (refer Table 4.1). There is an increase in Lines Of Code (LOC) and number of methods values across all feature envy removed versions. However, move method refactoring technique does not increase the count of classes, hence number of classes remain unaltered across these versions. Inconsistency is
however observed in CBO and CC values. CBO has variant value trends for Commons IO_F, Commons Beanutils_F and JHotdraw_F respectively, whereas cyclomatic complexity remained same for JHotdraw_F and Commons IO_F, and decreased for Commons Beanutils_F.

Table 4.1 Comparison of architectural metrics and code smell information of type 1 refactored versions

<table>
<thead>
<tr>
<th>Applications</th>
<th>LOC</th>
<th>#Classes</th>
<th>#Methods</th>
<th>CBO</th>
<th>CC</th>
<th>#God Class Instances</th>
<th># Long Method Instances</th>
<th>#Feature Envy Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>JH</td>
<td>73231</td>
<td>552</td>
<td>5008</td>
<td>513</td>
<td>1.33</td>
<td>43</td>
<td>117</td>
<td>17</td>
</tr>
<tr>
<td>JH_F</td>
<td>73267</td>
<td>552</td>
<td>5025</td>
<td>513</td>
<td>1.33</td>
<td>43</td>
<td>117</td>
<td>0</td>
</tr>
<tr>
<td>JH_G</td>
<td>74821</td>
<td>624</td>
<td>5228</td>
<td>573</td>
<td>1.33</td>
<td>0</td>
<td>117</td>
<td>17</td>
</tr>
<tr>
<td>JH_L</td>
<td>74989</td>
<td>552</td>
<td>5278</td>
<td>513</td>
<td>1.32</td>
<td>43</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>CB</td>
<td>71767</td>
<td>322</td>
<td>3118</td>
<td>299</td>
<td>1.99</td>
<td>15</td>
<td>23</td>
<td>5</td>
</tr>
<tr>
<td>CB_F</td>
<td>72065</td>
<td>322</td>
<td>3128</td>
<td>294</td>
<td>1.98</td>
<td>15</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>CB_G</td>
<td>72667</td>
<td>345</td>
<td>3205</td>
<td>332</td>
<td>1.96</td>
<td>0</td>
<td>23</td>
<td>5</td>
</tr>
<tr>
<td>CB_L</td>
<td>71786</td>
<td>322</td>
<td>3141</td>
<td>302</td>
<td>1.99</td>
<td>15</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>CI</td>
<td>55473</td>
<td>262</td>
<td>2648</td>
<td>256</td>
<td>1.77</td>
<td>18</td>
<td>22</td>
<td>15</td>
</tr>
<tr>
<td>CI_F</td>
<td>55517</td>
<td>262</td>
<td>2667</td>
<td>257</td>
<td>1.77</td>
<td>18</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>CI_G</td>
<td>56081</td>
<td>280</td>
<td>2725</td>
<td>274</td>
<td>1.75</td>
<td>0</td>
<td>22</td>
<td>15</td>
</tr>
<tr>
<td>CI_L</td>
<td>55704</td>
<td>262</td>
<td>2680</td>
<td>258</td>
<td>1.77</td>
<td>18</td>
<td>0</td>
<td>15</td>
</tr>
</tbody>
</table>

4.1.2 Analysis of God Class Removed Version

God class code smell is eliminated by applying extract class refactoring technique (see Section 3.2). Lines Of Code (LOC) and number of methods increase due to the increase in number of classes as a result of applying this technique. It is apparent from Table 4.1 that Lines Of Code (LOC), number of methods and number of classes are showing increasing
trends across all god class removed versions. For JHotdraw_G and Commons Beanutils_G, number of extracted classes is more than number of god classes, because sometimes more than one extract class may be required to completely remove one god class. CBO values for all versions has increased. Cyclomatic complexity value remains unaltered for JHotdraw_G, a decline in its values is observed for the god class removed versions of other two applications (Commons Beanutils_G and Commons IO_G).

4.1.3 Analysis of Long Method Removed Version

Long method code smell is eliminated by applying extract class refactoring (see Section 3.2). It increases Lines Of Code (LOC) and number of methods across all the long method removed versions as is evident from its description in Section 3.2, however there is no impact on number of classes. For JHotdraw_L, CBO value is unaltered; whereas for the long method removed versions of other two applications (Commons Beanutils_L and Commons IO_L), CBO value has increased. Cyclomatic complexity values decreased for JHotdraw_L, while remaining unchanged for Commons Beanutils_L and Commons IO_L.

4.1.4 Analysis of Energy Trends

As represented in graph shown in Figure 4.1, all three applications show similar energy consumption trends. As predicted, feature envy removed version consumes less energy than original version whereas other two versions i.e. G and L versions consume more energy than their respective original versions. Of all three refactored versions, god class removed version is found to consume the highest energy.

**Figure 4.1 Energy trends across sample applications**
4.2 Analysis of Type 2 Refactored Version

As explained earlier (see Section 3.5.1), type 2 refactored versions are those that are generated by eliminating code smells in different possible permutations. In total, there are eighteen versions of this type. Table 4.2-4.4 represents the architecture metrics and energy consumption values for experimental versions of type 2 of Jhotdraw, Commons Beanutils and Commons IO. Since all three refactoring techniques (move method, extract class and extract method) are applied to create this type of refactored versions, increase in LOC, number of methods and number of classes in all versions is obvious. Further investigating the amount with which architecture metric values have altered, following observations are made.

4.2.1 Analysis of Versions of JHotdraw

All versions of JHotdraw show variable increase in LOC, number of classes, number of methods and CBO value whereas cyclomatic complexity remains constant. LOC value is increased by 3524 in JHotdraw_FGL, by 3771 in JHotdraw_FLG, by 2965 in JHotdraw_GFL, by 3153 in JHotdraw_GLF, by 3469 in JHotdraw_LGF and by 3319 in JHotdraw_LFG. JHotdraw_FLG shows highest increase in number of classes by 100. In JHotdraw_FGL, JHotdraw_GFL and JHotdraw_GLF count of classes is increased by 72. JHotdraw_LGF and JHotdraw_LFG results in increase of 81 and 79 classes respectively. Count of methods is increased by 512, 532, 426, 455, 505 and 480 in JHotdraw_FGL, JHotdraw_FLG, JHotdraw_GFL, JHotdraw_GLF, JHotdraw_LGF and JHotdraw_LFG respectively. CBO value of JHotdraw_GFL shows minimum increase of 49 as compared to all others versions that show the increase of 77, 82, 51, 61 and 56 in JHotdraw_FGL, JHotdraw_FLG, JHotdraw_GFL, JHotdraw_GLF, JHotdraw_LGF and JHotdraw_LFG respectively. As is represented in Table 4.2, JHotdraw_FGL shows the maximum increase of 51,160 mJ in energy consumption value, whereas, as expected, JHotdraw_GFL shows the increase of 20,670 mJ in energy consumption value which is least of all other versions. Increase of 48077 mJ, 21330 mJ, 21550 mJ and 21320 mJ is observed in JHotdraw_FGL, JHotdraw_GLF, JHotdraw_LGF and JHotdraw_LFG respectively.

4.2.2 Analysis of Versions of Commons Beanutils

As shown in Table 4.3, all versions of Commons Beanutils show variable increase in architecture metrics value except for the value of cyclomatic complexity which shows
Table 4.2 Architectural metrics and code smell information for refactored versions of JHotdraw

<table>
<thead>
<tr>
<th>Applications</th>
<th>LOC</th>
<th>#Classes</th>
<th>#Methods</th>
<th>CBO</th>
<th>Cyclomatic Complexity</th>
<th>Energy Usage (mJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JH</td>
<td>73231</td>
<td>552</td>
<td>5008</td>
<td>513</td>
<td>1.33</td>
<td>64230</td>
</tr>
<tr>
<td>JH_FGL</td>
<td>76755</td>
<td>624</td>
<td>5520</td>
<td>590</td>
<td>1.33</td>
<td>112307</td>
</tr>
<tr>
<td>JH_FLG</td>
<td>77002</td>
<td>652</td>
<td>5540</td>
<td>595</td>
<td>1.33</td>
<td>115390</td>
</tr>
<tr>
<td>JH_GFL</td>
<td>76196</td>
<td>624</td>
<td>5434</td>
<td>562</td>
<td>1.33</td>
<td>84900</td>
</tr>
<tr>
<td>JH_GLF</td>
<td>76384</td>
<td>624</td>
<td>5463</td>
<td>564</td>
<td>1.33</td>
<td>85560</td>
</tr>
<tr>
<td>JH_LGF</td>
<td>76700</td>
<td>633</td>
<td>5513</td>
<td>574</td>
<td>1.33</td>
<td>85780</td>
</tr>
<tr>
<td>JH_LFG</td>
<td>76550</td>
<td>631</td>
<td>5488</td>
<td>569</td>
<td>1.33</td>
<td>85550</td>
</tr>
</tbody>
</table>

decrease across all versions. Increase of 1626, 1656, 1046, 1387, 1401 and 1087 in LOC is reported in Commons Beanutils_FGL, Commons Beanutils_FLG, Commons Beanutils_GFL, Commons Beanutils/GLF, Commons Beanutils/GLF, Commons Beanutils/LGF and Commons Beanutils/LFG respectively. Table 4.3 demonstrates the increase of 30 classes in Commons Beanutils_FGL and Commons Beanutils_FLG, increase of 23 in Commons Beanutils_GFL, Commons Beanutils/GLF and Commons Beanutils/LFG, increase of 26 Commons Beanutils_LGF. Count of methods is increased by 152, 157, 111, 145, 146, and 118 in Commons Beanutils_FGL, Commons Beanutils_FLG, Commons Beanutils_GFL, Commons Beanutils/GLF, Commons Beanutils/LGF, Commons Beanutils/LFG and Commons Beanutils_LFG respectively. Commons Beanutils_FGL and Commons Beanutils_FLG shows the maximum increase of 29 in CBO value whereas Commons Beanutils_GFL and Commons Beanutils/GLF shows the minimum increase of 20. Increase of 25 and 22 is reported in CBO value of Commons Beanutils_LGF and Commons Beanutils_LFG. Value of cyclomatic complexity is decreased by 0.04 in Commons Beanutils_FGL, Commons Beanutils_FLG, Commons Beanutils_GFL and Commons Beanutils_LGF and by 0.03 in remaining versions. Commons Beanutils_FLG indicates the increase of 8270 mJ in energy consumption value, which is the maximum increase, whereas, as expected Commons Beanutils_GFL indicates the increase of 1440mJ, which is least of all versions. Increase of 6650 mJ, 3430 mJ, 6120 mJ and 5900 mJ is
observed in Commons Beanutils_FGL, Commons Beanutils_GFL, Commons Beanutils_LGF and Commons Beanutils_LFG respectively.

### Table 4.3 Architectural metrics and code smell information for refactored versions of Commons Beanutils

<table>
<thead>
<tr>
<th>Applications</th>
<th>LOC</th>
<th>#Classes</th>
<th>#Methods</th>
<th>CBO</th>
<th>Cyclomatic Complexity</th>
<th>Energy Usage (mJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CB</td>
<td>71767</td>
<td>322</td>
<td>3118</td>
<td>299</td>
<td>1.99</td>
<td>163580</td>
</tr>
<tr>
<td>CB_FGL</td>
<td>73393</td>
<td>352</td>
<td>3270</td>
<td>328</td>
<td>1.95</td>
<td>170230</td>
</tr>
<tr>
<td>CB_FLG</td>
<td>73423</td>
<td>352</td>
<td>3275</td>
<td>328</td>
<td>1.95</td>
<td>171850</td>
</tr>
<tr>
<td>CB_GFL</td>
<td>72813</td>
<td>345</td>
<td>3229</td>
<td>319</td>
<td>1.96</td>
<td>165020</td>
</tr>
<tr>
<td>CB_GLF</td>
<td>73154</td>
<td>345</td>
<td>3263</td>
<td>319</td>
<td>1.95</td>
<td>167010</td>
</tr>
<tr>
<td>CB_LGF</td>
<td>73168</td>
<td>348</td>
<td>3264</td>
<td>324</td>
<td>1.95</td>
<td>169700</td>
</tr>
<tr>
<td>CB_LFG</td>
<td>72854</td>
<td>345</td>
<td>3236</td>
<td>321</td>
<td>1.96</td>
<td>169480</td>
</tr>
</tbody>
</table>

### 4.2.3 Analysis of Versions of Commons IO

As demonstrated in Table 4.4, all architecture metrics except cyclomatic complexity has shown variable increase in its values. LOC values are increased by 747, 826, 662, 756, 725 and 746 in Commons IO_FGL, Commons IO_FLG, Commons IO_GFL, Commons IO_GLF, Commons IO_LGF and Commons IO_LFG respectively. Commons IO_FGL and Commons IO_FLG results in increase of 20 classes, Commons IO_GFL, Commons IO_GLF and Commons IO_LGF results in increase of 18 classes and Commons IO_LFG results in increase of 16 classes. Count of methods is increased by 103, 107, 81, 92, 98 and 88 in Commons IO_FGL, Commons IO_FLG, Commons IO_GFL, Commons IO_GLF, Commons IO_LGF and Commons IO_LFG respectively. Commons IO_FGL and Commons IO_FLG indicates the increase of 16 in CBO value, Commons IO_GFL, Commons IO_GLF and Commons IO_LGF indicates the increase of 14 in CBO value and Commons IO_LFG indicates the increase of 12 in CBO value. Commons IO_FLG indicates decrease of 0.04 in cyclomatic complexity while other versions indicate the decrease of 0.05. As demonstrated in Table 4.3 Commons IO_GFL indicates the increase of 1030 mJ, which is least of all other
versions while other versions, Commons IO_FGL, Commons IO_FLG, Commons IO_GLF, Commons IO_LGF and Commons IO_LFG, indicate the increase of 1468 mJ, 1540 mJ, 1140 mJ, 1400 mJ, and 1154 mJ respectively.

Table 4.4 Architectural metrics and code smell information for refactored versions of Commons IO

<table>
<thead>
<tr>
<th>Applications</th>
<th>LOC</th>
<th>#Classes</th>
<th>#Methods</th>
<th>CBO</th>
<th>Cyclomatic Complexity</th>
<th>Energy usage (mJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CI</td>
<td>55473</td>
<td>262</td>
<td>2648</td>
<td>256</td>
<td>1.77</td>
<td>82250</td>
</tr>
<tr>
<td>CI_FGL</td>
<td>56220</td>
<td>282</td>
<td>2751</td>
<td>272</td>
<td>1.75</td>
<td>83718</td>
</tr>
<tr>
<td>CI_FLG</td>
<td>56299</td>
<td>282</td>
<td>2755</td>
<td>272</td>
<td>1.74</td>
<td>83790</td>
</tr>
<tr>
<td>CI_GFL</td>
<td>56135</td>
<td>280</td>
<td>2729</td>
<td>270</td>
<td>1.75</td>
<td>83280</td>
</tr>
<tr>
<td>CI_GLF</td>
<td>56229</td>
<td>280</td>
<td>2740</td>
<td>270</td>
<td>1.75</td>
<td>83390</td>
</tr>
<tr>
<td>CI_LGF</td>
<td>56198</td>
<td>280</td>
<td>2746</td>
<td>270</td>
<td>1.75</td>
<td>83650</td>
</tr>
<tr>
<td>CI_LFG</td>
<td>56219</td>
<td>278</td>
<td>2736</td>
<td>268</td>
<td>1.75</td>
<td>83404</td>
</tr>
</tbody>
</table>

4.3 Result Summary

The graph shown in Figure 4.1 represents the energy consumption behavior of different experimental versions. X-axis represents the refactored versions and y-axis represents the energy values corresponding to the refactored versions in mili joules. As shown in Figure 4.1, it becomes conspicuous that eliminating feature envy results in reduced energy consumption, whereas eliminating god classes and long methods results in increased energy consumption. God class removed version consumes more energy as compared to long methods removed version. All three open source java applications follow the same energy consumption trend. As number of classes increase after removal of god class smell, so number of function calls increase in order to interact with subordinate classes, thus introducing overhead of accessing subordinate classes for proper functioning and increasing energy consumption. In order to remove long method code smell, a method is extracted from long method which performs some of its functionalities and to access this extracted method additional function calls and message exchanging is required, thus increasing energy consumption. As more messages are
required to access subordinate classes than messages required to access function in same class, energy consumption of god class removed version is more than that of long method removed version. To remove feature envy code smell, method is moved to that class to which it is accessing the most thus decreasing message traffic, and improving energy efficiency.

As shown in Table 4.2-4.4 different refactored versions yield different architecture metrics indicating different internal structure, it becomes evident that energy consumed by different versions created by using different code smell permutation yield different energy consumption. As expected, experimental version GFL yields minimum energy consumption in all applications, thus revealing consistency across all applications.

God class removed versions consume more energy and CBO value of this version is also greater in all three applications while cyclomatic complexity value is lesser of all. While considering long methods removed versions, energy consumption as well as CBO value has shown an increase with an exception of CBO value of JHotdraw_L, which remains unaltered and cyclomatic complexity value has remained same. Feature envy removed version indicates decrease in energy consumption but value of CBO for JHotdraw remains unaltered, decreases for Commons Beanutils and increases for Commons IO and cyclomatic complexity has remained same for JHotdraw_F and Commons IO_F and decreased for CommonsBeanutils_F.

GFL version is known to yield minimum energy consumption and has low CBO value as compared to other refactored versions with an exception in the CBO value of Commons IO. Cyclomatic complexity value has also decreased, but there are other versions also (such as FLG, FGL) whose cyclomatic complexity is lower than that of GFL versions and energy consumption is greater than that of GFL.

It becomes conspicuous from above discussions that no definite relationship exists between architectural metrics and energy consumption values of refactored versions.

**4.4 Chapter Summary**

This chapter presented all the data collected (metric values and energy consumption values) of both types of refactored versions (created by eliminating instances of each type of code smells and created by eliminating instances of all code smells in different permutations) by following
the procedure demonstrated in Section 3.5. Analysis of refactored versions of type 1 is discussed in the following sequence, in first place analysis of architectural metrics of feature envy removed versions is carried out, in second place analysis of architectural metrics of god class removed versions is done, in third place analysis of architectural metrics of long removed versions is performed and lastly the energy consumption trends in all three versions are analyzed. For type 2 refactored versions, analysis is performed application wise, firstly the architecture metrics and energy consumption trends are discussed for all versions of JHotdraw, then for all versions of Commons Beanutils and lastly for all versions of Commons Beanutils. At the end of this chapter results are summarized.
CHAPTER 5

CONCLUSION AND FUTURE WORK

This chapter concludes this work by discussing its importance in the field of green software engineering, followed by a summary of observations and some possible future directions.

5.1 Conclusion

Due to the current human practices in Information Technology (IT), sector environment is getting affected adversely. There is an immediate need to take corrective measures by adopting green engineering practices which includes the development of sustainable software products. Nowadays focus is rapidly shifting toward realizing energy optimized software products. Several studies \([1, 2, 4, 6–8, 12-14 20, 23-29]\) proved that changes in software alter its energy consumption behavior. In this study, refactoring \([5]\) is performed to eliminate three code smells \([5]\) as a result of which changes are induced in internal structure of software application. Due to the changes in internal structure of code energy consumption behavior of software system changes.

It is known from previous studies that code smell elimination has impact on energy consumption behavior of software system. In this analysis, it is verified by analyzing the impact of removing three most common code smells (god class, feature envy and long method) on three open source java applications. This work further analyzed the impact of eliminating code smells in different possible permutations of these code smells on energy consumption behavior of selected software applications. Additionally, changes are analyzed in architectural metrics of all refactored versions to identify the relationship for the same with energy consumption values.

Results of our study depicted the same energy trends across all applications where refactored versions are created by eliminating all instances of only one smell. A uniform decline in energy consumption values of feature envy removed version is observed, whereas other two versions (god class removed and long method removed) have shown an increase in energy consumption values.
It has been noticed that different versions created by eliminating code smells in different possible permutations yield different energy consumption values. Also the architecture metric values for different versions were different thus indicating different internal structure and hence supporting the observations of different energy consumption values. Our results further demonstrated that refactored versions generated after removing code smells in GFL sequence yield minimum energy consumption across all applications as compared to other refactored versions. Architectural metrics have shown inconsistent behavior across applications, resulting in no definite relationship with energy consumption values of refactored versions.

5.2 Future Work

This work encourages the future work to gain better understanding of the impact of individual code smell elimination and permutations of code smell elimination on energy consumption behavior of software.

- More number of applications of varied domains can be analyzed by considering the effect of other code smells as well on energy consumption of software systems.
- Sequencing of three code smells have been performed in this study. Next goal is to devise permutations of code smell elimination for other code smells as well, so as to gain more knowledge which can aid in the development of energy aware software.
- Several energy measurement tools can be employed to examine and validate the energy consumption values to strengthen our work.
- Impact of various other refactoring techniques, that can be used to eliminate the code smells (god class, long method and feature envy), on architectural metrics and energy consumption of software applications can be explored. Further the impact of different permutations of removing these code smells (using other refactoring techniques) on architectural metrics and energy consumption of software can be investigated.
REFERENCES


References


Commons Beanutils: https://github.com/apache/commons-beanutils.

Commons IO: https://github.com/apache/commons-IO

JDeodorant: http://www.jdeodorant.com

Atlassian Clover: https://www.atlassian.com/software/clover

Code pro Analytix: https://marketplace.eclipse.org/content/codepro-analytix


