# An Empirical Investigation into Code Smell Elimination Sequences for Energy Efficient Software

Garima Dhaka
National Institute of Technology
Jalandhar, India
E-mail: garima.dhaka1992@gmail.com

Paramvir Singh
National Institute of Technology
Jalandhar, India
E-mail: singhpv@nitj.ac.in

***Abstract*—Recent research has shown that maintainability improving activities, such as removing code smells using recommended refactoring activities, may degrade the energy consumption behavior of software systems. However, current research is still immature and requires considerable effort for transferring findings to practice. This work empirically investigates the impact of eliminating a set of three notorious code smells, individually as well as in all six possible sequences, on energy consumption behavior of software systems. It also analyzes whether any relationship exists between software architecture sustainability (in terms of energy efficiency) and maintainability within the context of individual and sequential code smell elimination. The study outcomes show that the selected code smell removal permutations yield variant levels of energy consumption values for the resulted refactored software versions. Also, a particular permutation is learned to yield most energy-efficient refactored software versions, when compared to all other code smell removal permutations.**

***Keywords—energy consumption; sustainability; code smell; refactoring; software architecture; maintainability; green IT.***

## I. INTRODUCTION

Due to the exponentially rising demand for IT based products and services, the technology markets worldwide are increasingly being flooded with complex software-intensive systems. Ironically, this trend has also brought the software development to the forefront of various environmental issues emerging from the high system energy consumption, for which hardware was once considered as solely responsible. Therefore, in order to stimulate the development of energy-efficient software systems, and encourage Green IT [1], researchers are shifting their focus from hardware to software system components.

Conventional software development processes focus on enhancing software quality attributes such as maintainability, mostly neglecting the sustainability through energy consumption attribute. Code smells have been defined as surface indications that usually correspond to deeper problems in a software system [2]. If not cured in time, these code smells could deteriorate a software's quality, primarily in terms of maintainability; thus hampering its growth and evolution. In order to eliminate code smells, refactoring has now long been the preferred mechanism. Refactoring is a behaviour-preserving code transformation activity that not only improves the software quality but also increases the developer satisfaction [3]. A negotiable trade-off between the quality and sustainability of software may help refactoring adoption in industry [4].

Except for a limited number of code smells and refactoring techniques that take energy behavior of software into consideration [5], most of the others [6] are defined to target only the software quality attributes such as readability, maintainability, etc. Over the past decade, there has been some active research toward evaluating software change against energy consumption [7]. However, there is limited research work [8, 9] that aims to investigate the impact of code smell elimination (using refactoring) on software energy efficiency.

The objective of this study is to empirically analyze the impact of eliminating code smells on the energy consumption behavior of object-oriented software systems. For this purpose, we experiment with eliminating all instances of three commonly diagnosed code smells (god class (G), feature envy (F) and long method (L)), individually as well as in six possible permutations (FLG, FGL, LGF, LFG, GFL, GLF), from three open source java applications. To provide further insights, the relationship between software architecture metrics and software energy consumption is also empirically explored.

## II. RELATED WORK AND MOTIVATION

Pinto *et al*. [11] conducted a survey-based research to identify arenas where refactoring can be highly beneficial to improve energy efficiency. Studies conducted by Park *et al.* [12] and Sahin *et al*. [13] indicated that different refactoring techniques exhibit different energy consumption behaviors. Reimann and Aβmann [14] defined an approach to explicitly derive relationship among quality (in terms of energy efficiency), code smells and refactorings. Hindle [7] analyzed a set of software metrics against power consumption across multiple software versions, and observed that power consumption was not related to metrics such as LOC. Rodriguez *et al.* [9] empirically observed that improving software maintainability and flexibility through code smell removal increases battery consumption of mobile devices. Gottschalk *et al.* [5] defined a number of energy code smells to track the energy inefficient code portions of software applications.

The motivations behind this work have been derived from two previous studies ([8, 12]). Recently, Castillo and Piattini [8] analyzed the impact of removing god class code smells (from two open source software applications) on software energy consumption behavior. Results obtained from their study indicate that removing god class code smells increases the energy consumption of software applications, primarily due to an increase in the inter-class

message traffic. We hence decided to additionally explore the effects of two other quite common code smells, feature envy and long method, on energy consumption behavior.

Park *et al*. [12] investigated the impact of applying single instances of various refactoring techniques [2] on software energy consumption behaviour. We mapped our selected set of three code smells to their preferred refactoring techniques (as defined in JDeodorant[1], a free Eclipse plug-in tool for code smell detection and removal), and their energy effects as given by Park *et al*. [12]. This mapping [16] resulted in decreased software energy consumption after the removal of feature envy code smell (by move method), and increased energy consumption after the removal of both god class (by extract class) and long method (by extract method) code smells. This observation further encouraged us to investigate the software energy consumption behavior against the removal of multiple code smell instances, generally present in real world software. Another motivating example is available at [16].

## III. Empirical Study Design

### A. Subjects under Study

The key specifications for three selected open source java applications, namely JHotDraw[2], Commons BeanUtils[3] and Commons IO[4] are presented in Table I. We based this study on the test suite execution of selected applications, hence only the sample applications having their test suites available were selected. Test cases are expected to cover at least the core functionalities of a software, and hence are believed to form useful criteria for the selection of sample applications. We denote three selected applications (in the order specified above) with JH, CB and CI respectively throughout the rest of this paper.

### B. Methodology

The overall research methodology employed in this study is presented in Fig. 1.

*1) Experimental subject creation and metric collection:* Experimental versions are created through following sequence of actions. First, all the applications alongwith their supporting jar libraries are imported in Eclipse using Github. Next, we use Clover[5] tool's coverage feature to determine the test suite covered code sections of selected applications. The information collected by Clover assisted us in identifying those sections of applications that are covered by the test suites. This helped us skip refactoring the uncovered code sections as the effects of refactoring such sections would be unnoticeable. Lastly, JDeodorant[1] tool is used to create the required experimental subject versions by identifying and eliminating all instances of three selected code smells (using refactoring techniques as mentioned in Section II).

TABLE I. Sample Applications

| Application | # Classes | # Test Cases | Cov. (%) |
|---|---|---|---|
| JHotDraw 6.2.0 | 552 | 3654 | 54.7 |
| Commons BeanUtils 1.9.3 | 322 | 1660 | 78.3 |
| Commons IO 2.6 | 262 | 1157 | 91.9 |

*Type-1* experimental subjects are generated by removing all instances of a particular code smell from a given application, and are represented as RV1 (Refactored Version 1) in Fig. 1. In order to create *Type-2* experimental subjects, all three code smells are removed in all six possible sequences (FLG, FGL, LGF, LFG, GFL, GLF) from each sample application. Given an RV1, all instances of one of the remaining two code smells are eliminated by applying appropriate refactorings; generating version RV2. Then, all instances of the last remaining code smell present in RV2 are eliminated generating the final experimental subject version, RV3. It is noteworthy that we do not perform any investigations on RV2 versions in this study.

Each refactored version created by either eliminating an individual code smell or a sequential group of code smells is denoted as "AppAcronym_Smell(/s)Acronym". For instance, JH_F refers to the refactored version generated after removing feature envy code smell from the original JH (JH_O) application, and JH_LFG refers to the JH version generated after eliminating code smells in above mentioned sequence (LFG). In total, there are 27 experimental subjects (9 *Type-1* subjects + 18 *Type-2* subjects). For each experimental subject, we then calculate a set of architecture metrics, including LOC (Lines Of Code), NOC (Number Of Classes), NOM (Number Of Methods), CBO (Coupling Between Objects) and CC (Cyclomatic Complexity), to analyze the effects of eliminating code smells on software architecture. We use Atlassian's Clover[5] tool to calculate NOC, NOM and LOC metrics; CodePro AnalytiX[6] tool to calculate CC; and ckjm[7] tool to calculate CBO metrics.

*2) Energy consumption measurement:* We use a software energy measurement tool called Jalen [15] for determining the energy consumption values for 27 experimental and 3 original application versions. It uses a statistical sampling technique to provide high precision energy values (in .csv format) at the method-level granularity of software code. In order to gather the correct energy values, we configured Jalen according to our system specifications by modifying its configuration properties file [16]. We execute each java application through its respective JUnit test suite by creating a driver class [16] containing a "main function", which invokes the available set of test cases. Each such driver class serves as the entry point for its respective software in the application jar file, which is then passed to Jalen for energy measurements.

As Jalen provides the energy consumption values at method level, the energy values for all application methods are summed up to get the energy consumption of the whole

[1] https://marketplace.eclipse.org/content/jdeodorant
[2] www.jhotdraw.org
[3] https://github.com/apache/commons-beanutils
[4] https://github.com/apache/commons-io
[5] https://www.atlassian.com/software/clover
[6] https://marketplace.eclipse.org/content/codepro-analytix
[7] http://www.spinellis.gr/sw/ckjm/

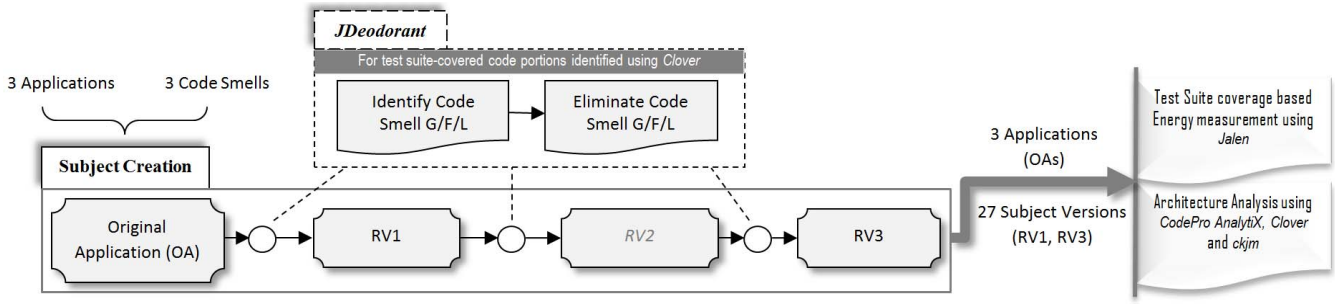


Figure 1. Illustration of research methodology

application. In order to counter the possible fluctuations in energy values, an average of 25 energy readings for each subject version is taken in a controlled environment to increase the accuracy of results. The experiments are performed on a computer system with Intel (R) Core (TM) i5-4210U CPU of 2.40GHz and 4GBs of RAM.

## IV. Results and Analysis

Table II summarizes the code smell information (test suite covered code only) related to three sample java applications along with their respective architectural metric and energy consumption values. For JH, 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 CB and CI, 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.

From the metric values of *Type-1* subjects shown in Table II, it is observed that there is an increase in LOC and NOM values across all _F versions. However, as move method refactoring technique does not increase the count of classes, NOC values remain unaltered across these versions. Inconsistency is, however, observed in CBO and CC values. For _G versions, LOC and NOM increase due to the increase in number of classes as a result of applying extract class refactoring. LOC, NOM and NOC values are showing increasing trends across all _G versions. For JH_G and CB_G, number of extracted classes is more than number of god classes, because often more than one extracted class is required to completely remove a god class. CBO values for all versions have increased. Extract method refactoring increases NOM and LOC values across all _L versions, however there is no impact on NOC values, and no definite impact on CC values. All three applications show similar energy consumption trends. As expected, _F versions consume less energy whereas _G and _L versions consume more energy, as compared to their respective original (_O) versions. Of all nine *Type-1* refactored versions, _G versions are found to consume the highest energy.

Table III shows the architectural metrics and energy consumption values for all 18 *Type-2* experimental subjects. Since all three refactoring techniques (move method, extract class and extract method) are applied to create *Type-2* refactored versions, an increase in LOC, NOM and NOC values in all versions is observed. All architectural metrics (except CC) show atleast some increase in the values for all *Type-2* refactored versions. Also, different refactored versions yield different energy consumption values, primarily due to the changes in the internal software designs across these versions as evident from their varied architecture metric values. _GFL versions yield minimum energy consumption whereas _FGL versions yield maximum energy consumption across all *Type-2* experimental subject versions.

_G versions of all applications consume more energy, and have higher LOC values than the corresponding _O versions; indicating a strong relationship between energy consumption and LOC metric within the context of god class code smell. _GFL versions consume lowest energy, and have lowest LOC and NOM values; indicating a strong relationship of LOC and NOM metrics with energy consumption. Also, _FLG versions consume highest energy, and have highest LOC and NOM values; indicating a strong relationship of LOC and NOM metrics with energy consumption. On the same lines, many more such relationships can be retrieved from the supporting information [16] derived from Tables II and III.

TABLE II. Architectural Metrics, Code Smell Information and Energy Values of Type-1 Refactored Versions

| Metric | JH_O | JH_F | JH_G | JH_L | CB_O | CB_F | CB_G | CB_L | CI_O | CI_F | CI_G | CI_L |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LOC | 73231 | 73267 | 74821 | 74989 | 71767 | 72065 | 72667 | 71786 | 55473 | 55517 | 56081 | 55704 |
| NOC | 552 | 552 | 624 | 552 | 322 | 322 | 345 | 322 | 262 | 262 | 280 | 262 |
| NOM | 5008 | 5025 | 5228 | 5278 | 3118 | 3128 | 3205 | 3141 | 2648 | 2667 | 2725 | 2680 |
| CBO | 513 | 513 | 573 | 513 | 299 | 294 | 332 | 302 | 256 | 257 | 274 | 258 |
| CC | 1.33 | 1.33 | 1.33 | 1.32 | 1.99 | 1.98 | 1.96 | 1.99 | 1.77 | 1.77 | 1.75 | 1.77 |
| # F | 17 | 0 | 17 | 17 | 5 | 0 | 5 | 5 | 15 | 0 | 15 | 15 |
| # G | 43 | 43 | 0 | 43 | 15 | 15 | 0 | 15 | 18 | 18 | 0 | 18 |
| # L | 117 | 117 | 117 | 0 | 23 | 23 | 23 | 0 | 22 | 22 | 22 | 0 |
| Energy(mJ) | 64230 | 56422 | 82340 | 68930 | 163580 | 162400 | 164470 | 164010 | 82250 | 80510 | 82890 | 82680 |

TABLE III. ARCHITECTURAL METRICS AND ENERGY VALUES OF TYPE-2 REFACTORED VERSIONS

| Version | LOC | NOC | NOM | CBO | CC | Energy(mJ) |
|---|---|---|---|---|---|---|
| JH_O | 73231 | 552 | 5008 | 513 | 1.33 | 64230 |
| JH_FGL | 76755 | 624 | 5520 | 590 | 1.33 | 112307 |
| JH_FLG | 77002 | 652 | 5540 | 595 | 1.33 | 115390 |
| JH_GFL | 76196 | 624 | 5434 | 562 | 1.33 | 84900 |
| JH_GLF | 76384 | 624 | 5463 | 564 | 1.33 | 85560 |
| JH_LGF | 76700 | 633 | 5513 | 574 | 1.33 | 85780 |
| JH_LFG | 76550 | 631 | 5488 | 569 | 1.33 | 85550 |
| CB_O | 71767 | 322 | 3118 | 299 | 1.99 | 163580 |
| CB_FGL | 73393 | 352 | 3270 | 328 | 1.95 | 170230 |
| CB_FLG | 73423 | 352 | 3275 | 328 | 1.95 | 171850 |
| CB_GFL | 72813 | 345 | 3229 | 319 | 1.96 | 165020 |
| CB_GLF | 73154 | 345 | 3263 | 319 | 1.95 | 167010 |
| CB_LGF | 73168 | 348 | 3264 | 324 | 1.95 | 169700 |
| CB_LFG | 72854 | 345 | 3236 | 321 | 1.96 | 169480 |
| CI_O | 55473 | 262 | 2648 | 256 | 1.77 | 82250 |
| CI_FGL | 56220 | 282 | 2751 | 272 | 1.75 | 83718 |
| CI_FLG | 56299 | 282 | 2755 | 272 | 1.74 | 83790 |
| CI_GFL | 56135 | 280 | 2729 | 270 | 1.75 | 83280 |
| CI_GLF | 56229 | 280 | 2740 | 270 | 1.75 | 83390 |
| CI_LGF | 56198 | 280 | 2746 | 270 | 1.75 | 83650 |
| CI_LFG | 56219 | 278 | 2736 | 268 | 1.75 | 83404 |

## V. THREATS TO VALIDITY

In this study, we used an automated tool (JDeodorant) for code smell detection and refactoring purposes. The validity of refactored versions thus obtained depends on the correctness of this tool. To minimize this threat, manual checks were performed on each *Type-1* refactored version. In order to mitigate the effect of fluctuations in energy consumption readings (taken using Jalen tool), and increase the accuracy of results, we took multiple readings (25 readings per refactored version) in controlled computer system environment. The experiments do not consider the code smell instances present outside the test suite-covered code portions of selected applications. The results of this study may hence be affected by the potential variations in the software energy values caused due to changes in the respective test suite coverage.

We do not focus on the relative number of instances of the three selected code smells present in each of the selected applications. It was noticed that in most applications explored during pre-study phases, including three selected applications, long method code smell had the highest number of instances followed by god class and feature envy smells. Another threat is the limited number of applications and code smells considered. However, selected applications are of different sizes, and contain a considerable number of instances of the targeted code smells.

## VI. CONCLUSIONS AND FUTURE WORK

This work empirically analyzes the impact of removing three of the most common code smells (god class, feature envy and long method), individually as well as sequentially, on the energy consumption behavior of three open source java applications. Additionally, in order to study the relationship between maintainability and sustainability (in terms of energy), the variations in architectural metric values of all the experimental subject (refactored) versions are analyzed. Experimental results show uniform energy trends across all applications when refactored versions are created by eliminating all instances of any one individual code smell. It is also observed that the refactored versions generated by applying GFL sequence yield minimum energy consumption as compared to those generated by applying other sequences. A number of relationships between software architecture metrics and energy consumption values are also observed.

This work can be extended in following ways: 1) impact of removing code smells (and code smell sequences, especially those practiced in actual development) other than the ones considered in this work is needed to be studied; 2) more number of applications of varied domains can be experimented upon for validating the our findings; 3) an ideal tradeoff between software maintainability and sustainability remains to be empirically derived; 4) impact of various other refactoring techniques, which can be used to remove the selected code smells, on energy consumption of software applications can be explored.

## REFERENCES


[1] S. Mingay, "Green IT: the new industry shock wave," Gartner I, 2007.

[2] M. Fowler, K. Beck, W. Brant, W. Opdyke, and D. Roberts, Refactoring: improving the design of existing code, Add.-Wes., 1999.

[3] M. Leppänen, S. Mäkinen, S. Lahtinen, O. Sievi-Korte, A. P. Tuovinen and T. Männistö, "Refactoring-a Shot in the Dark?," In IEEE Software, vol. 32, no. 6, pp. 62-70, Nov.-Dec. 2015.

[4] T. Sharma, G. Suryanarayana and G. Samarthyam, "Challenges to and Solutions for Refactoring Adoption: An Industrial Perspective," In IEEE Software, vol. 32, no. 6, pp. 44-51, Nov.-Dec. 2015.

[5] M. Gottschalk, J. Jelschen, and A. Winter, "Refactoring for Energy Efficiency," In Advances and New Trends in Environmental and Energy Informatics, pp. 77-96, 2016.

[6] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," Journal of Systems and Software, vol. 86, no. 10, pp. 2639-2653, 2013.

[7] A. Hindle, "Green mining: A methodology of relating software change to power consumption," In IEEE Working Conference on Mining Software Repositories, pp. 78-87, 2012.

[8] R. Perez-Castillo and M. Piattini, "Analyzing the harmful effect of god class refactoring on power consumption," IEEE Software, vol. 31, no. 3, pp. 48-54, 2014.

[9] A. Rodríguez, M. Longo, and A. Zunino, "Using bad smell-driven code refactorings in mobile applications to reduce battery usage". In Proc. of Argentine Symp. on Soft. Eng., pp. 56-68, 2015.

[10] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring-improving coupling and cohesion of existing code". In Proceedings of the Working Conference on Reverse Engineering, pp. 144-151, 2004.

[11] G. Pinto, F. Soares-Neto, and F. Castor, "Refactoring for energy efficiency: a reflection on the state of the art," In Proceedings of the Int'l Workshop on Green and Sustainable Software, pp. 29-35, 2015.

[12] J. J. Park, J. E. Hong, and S. H. Lee, "Investigation for Software Power Consumption of Code Refactoring Techniques," In Int'l Conf. on Software Engg. and Knowledge Engg., pp. 717-722. 2014.

[13] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?," In Proc. of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 1-10, 2014.

[14] J. Reimann and U. Aßmann, "Quality-Aware Refactoring for Early Detection and Resolution of Energy Deficiencies," In Proc. Int'l Conf. on Utility and Cloud Computing, pp. 321-326, 2013.

[15] A. Noureddine, R. Rouvoy, and L. Seinturier, "Unit testing of energy consumption of software libraries," In Proc. of the 29th Annual ACM Symposium on Applied Computing, pp. 1200-1205, 2014.

[16] Supporting Information: 2016. *www.pvsingh.com/g_dhaka*