Empirical Evaluation of the Energy Impact of Refactoring Code Smells

Verdecchia, Roberto; Saez, Rene' Aparicio; Procaccianti, Giuseppe; Lago, Patricia

published in
ICT4S2018. 5th International Conference on Information and Communication Technology for Sustainability 2018

DOI (link to publisher)
10.29007/dz83

document version
Publisher's PDF, also known as Version of record

EPiC Series in Computing

Volume 52, 2018, Pages 365–383

ICT4S2018. 5th International Conference on Information and Communication Technology for Sustainability

Empirical Evaluation of the Energy Impact of Refactoring Code Smells

Roberto Verdecchia[1,2], René Aparicio Saez[2], Giuseppe Procaccianti[2], and Patricia Lago[2]

[1] Gran Sasso Science Institute
L'Aquila, Italy
roberto.verdecchia@gssi.it
[2] Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
g.procaccianti@vu.nl, p.lago@vu.nl

Abstract

Software energy efficiency has gained the increasing attention of the research community. How to improve it, however, still lacks evidence. Specifically, the impact of code smell refactoring on energy efficiency has been scarcely investigated. In the exploratory study here reported, we investigate the impact on performance and energy consumption of refactoring well-known code smells on Java software applications. In order to understand if software metrics can be used as indicators of the energy impact of refactoring, we also measured the variation caused by refactoring on a set of well-established software metrics. We conducted a controlled experiment using state-of-the-art power measurement equipment. Statistical hypothesis testing and effect size estimation were performed on the experimental results, which show that in one out of three applications, refactoring each smell significantly impacted power- and energy consumption. E.g., refactoring Feature Envy and Long Method smells led to a 49% energy efficiency improvement. No software metric, however, significantly correlated with execution time, power or energy consumption. In conclusion, refactoring code smells resulted to be a viable process to significantly improve software energy efficiency. The magnitude of the impact may depend on application properties, e.g. size or age. Further research is needed to understand the relationship between software metrics and energy efficiency.

1 Introduction

Computing devices have become a major part of our everyday life. The number of these devices is predicted to globally increase in the coming years. Not only do people own more devices themselves, they also increasingly rely on services provided by Cloud providers, which are typically hosted in large-scale data centers. This brings up the issue of their environmental impact: the carbon footprint of Information and Communications Technology (ICT) accounts for 2% of global emissions [26] and is expected to keep on growing [29]. Data centers alone account for around 1.1% to 1.5% of global energy consumption according to a report from 2010 [24].

B. Penzenstadler, S. Easterbrook, C. Venters and S.I. Ahmed (eds.), ICT4S2018 (EPiC Series in Computing, vol. 52), pp. 365–383

These numbers indicate the global scale reached by ICT, and shows the need for energy efficient ICT solutions. While hardware solutions have been thoroughly researched, the same cannot be said for software. As shown by Pinto et al. [32], only since recent years the energy efficiency of application software is taken into consideration [47]. It is expected that increasing software energy efficiency can cause major changes in the energy consumption of ICT, thanks to the global scale of software usage [22]. Therefore, the next step in finding significant energy efficient improvements in ICT will most likely be software related.

There are a number of empirical studies showing how software engineering best practices can improve energy efficiency [34, 2]. Code refactoring is probably the most common approach to re-engineer software applications in order to improve non-functional attributes. Refactoring activities are typically aimed at removing *code smells* [18], that can be defined as "certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality" [43]. Detection and refactoring of some code smells can be automated by using special-purpose tools, among which the Eclipse plugin JDeodorant [14] results to be the one which is most commonly utilized [44].

In many studies, refactoring code smells was found to have a positive impact on software maintainability [30, 49]. However, the impact on energy efficiency is only marginally investigated [31, 48, 19, 38].

In this study, we aim to perform an exploratory analysis of the impact of code smell refactoring on energy consumption and performance in software applications. We selected five different code smells (Feature Envy, Type Checking, Long Method, God Class and Duplicated Code) that we automatically detected and refactored in three open-source, ORM-based Java software applications. The refactoring was applied both in isolation (i.e. on all the occurrences of a single smell) and in combination (i.e. on all the occurrences of all smells). We then performed a benchmark of each version of the application by means of an automated test scenario and we collected energy consumption and performance metrics in a controlled environment. We also investigate whether well-established object-oriented software metrics can be used as indicators of the impact of refactoring the smells. This would allow developers to use such metrics as proxies for identifying and refactoring code smells with a high energy impact, thus removing the need of performing dedicated measurements and benchmarks. To the best of our knowledge, this is the largest study on the energy impact of code smells.

The structure of the paper is as follows. Section 2 will present related work on this topic. Section 3 and 4 discusses the definition and planning of our empirical experiment, along with subject selection, hypothesis formulation and instrumentation. Section 5 describes our experiment design and execution. Section 6 provides the results of the experiments, further discussed in Section 7. Section 8 elaborates on the most relevant threats to the validity of our experiment and how we mitigated them. Section 10 concludes the paper.

2 Related Work

As mentioned in Section 1, the current evidence regarding the energy impact of code smells is quite scarce. From an inspection of the related literature, the following empirical studies related to energy characteristics of code smells were identified:

- Perez-Castillo & Piattini [31] investigate the energy impact of refactoring a specific code smell, namely "God Class". They use a tool to detect and refactor multiple occurrences of the smell in two open-source Java software applications. The authors additionally analyze the architectural impact of the refactoring by extracting relevant software metrics. The

results of this work show an increase in maintainability but also an increase in power consumption and execution time due to a drastic increment of messages exchanged after the refactoring process.

- Vetrò et al. [48] investigate the energy impact of refactoring nine known performance smells in special-purpose C functions on a single embedded system. Five smells resulted to statistically increase power consumption, although quite limited in absolute values (less than 1%). However, none had a noticeable impact on execution time.

- Gottschalk et al. [19] investigate the energy impact of refactoring five code smells from two small size Android apps. The smells adopted are specifically related to mobile energy consumption, i.e. targeting third-party advertising and management of specific mobile resources (Wi-Fi, GPS, display). Hence they greatly differ from the smells described by Fowler [18] adopted in this paper. The impact evaluation of the smell shows that most refactorings reduce energy consumption, albeit with varying figures (from 5 to 30%).

- Rodriguez et al. [38] investigate the energy impact of refactoring two smells (namely, God Class and Brain Method) from three Android apps. The authors utilize JDeodorant in order to carry out the refactoring process. Their results indicate that refactoring smells resulted in an overall significant increase of energy consumption (up to 70%) depending on the specific application. Authors argue this is because refactoring the smells results in a higher quality design, but also in an increase of object creations and message exchanges. They also conduct specific energy-oriented refactoring, without refactoring code smells, which led to a significant decrease of energy consumption (7 to 35%). Authors conclude there is a trade-off between quality of design (i.e. refactored code smells) and energy efficiency.

- Sahin et al. [40] investigate the energy impact of commonly performed refactorings. Authors describe an empirical experiment conducted by applying six different refactorings to nine Java applications. Refactorings were chosen w.r.t. their popularity among developers using the Eclipse IDE, and were not directly linked with code smells. In almost 30% of the cases, refactorings significantly influenced energy consumption. However, the impact was not consistent, as both negative and positive impacts were observed.

Other empirical studies focused on the energy impact of software constructs or programming best-practices:

- Hindle [23] investigates the impact of software change on power consumption, and the relationship with software metrics. Subjects are three applications: Firefox, Vuze, rTorrent. For each application a set of different versions and releases is selected. Main findings show that power consumption is not consistent among versions but no link is found with software evolution. No significant correlation was found between static software metrics (e.g. coupling, cohesion, fan-in/fan-out) and power consumption.

- Tonini et al. [45] evaluate the energy impact of two best-practices proposed by Google to improve performance in Android applications. Both best-practices resulted to provide significant improvements in energy efficiency and performance among the different devices.

- Linares-Vásquez et al. [27] aim at identifying energy-greedy API calls, and if sequences of API calls repeat themselves frequently causing energy consumption anomalies. The study analyzed execution traces of 55 Android apps, looking for the most energy consuming API

calls. Results showed that APIs related to *GUI & Image Manipulation* and *Database* are more energy-consuming. Using getters and setters to access internal class fields caused high energy consumption. This led to the identification of a trade-off between information hiding and energy efficiency.

From this overview, the state-of-the-art suggests a significant energy impact of code smells. However, the available evidence is limited in terms of number of smells and representativeness of the tested applications.

3 Experiment Definition

Following the Goal-Question-Metrics (GQM) template [4], our goal is defined as follows:

Analyze **Code Smells**
for the purpose of **evaluation of impact**
in terms of **Performance and Energy Efficiency**
from the point of view of **Software Developers**
in the context of **Open Source ORM-based Java applications**

In order to achieve our goal, the following three *research questions (RQs)* must be answered:

RQ1 What is the impact on Energy Consumption of refactoring Code Smells in Open Source Java ORM-based applications? - By answering this question, the impact of code smells refactoring on energy consumption can be determined. This will be done by looking at the Energy Consumption (EC) for each code smell per tested application.

RQ2 What is the impact on Performance of refactoring Code Smells in Open Source Java ORM-based applications? - This question will provide insight into whether the refactoring will have any impact on performance. Performance (Perf) is measured in terms of execution time per each code smell.

RQ3 Which object-oriented software metrics can be good indicators of Performance or Energy Consumption impacts of Code Smells in Open Source Java ORM-based applications? - This question is aimed at assessing whether or not certain software metrics can be used as indicators of the energy consumption or performance impact of refactoring code smells. This will be done by looking at the correlation index between metrics and the measured performance and energy consumption for each code smell. Therefore this question can be split up as follows:

RQ3.1 Which object-oriented software metrics show a correlation with Energy Consumption when refactoring Code Smells in Open Source Java ORM-based applications?

RQ3.2 Which object-oriented software metrics show a correlation with Performance when refactoring Code Smells in Open Source Java ORM-based applications?

The metrics required to answer the above questions are:

- **M1**. **Energy Consumption (Watt-Hours)**. This metric will be derived as the average Power consumption (**M2**) over Time (**M3**).

- **M2**. **Power Consumption (Watts)**.
- **M3**. **Time (seconds)**.
- **M4**. **Afferent Couplings (CA)** [28]. This metric is an indicator of the responsibility of the class. Refactoring can change certain responsibilities, thus affecting this metric.
- **M5**. **Efferent Couplings (CE)** [28]. The efferent couplings show an indication of the external dependencies of a class. Changes in a class could result in changes to the value of this metric.
- **M6**. **Depth of Inheritance Tree (DIT)** [7]. This metric will indicate if inheritance is changed after refactoring.
- **M7**. **Lack of Cohesion in Methods (LCOM)** [7]. Indication of the Cohesion between methods. Refactoring of methods could change cohesion between them.
- **M8**. **Weighted Methods per Class (WMC)** [7]. Metric to specify the sum of the complexities of all class methods. As refactoring might change structure and content of methods, change in this value is to be expected.
- **M9**. **Number of Classes (NOC)** [7]. Metric that indicates the number of sub classes of a class. Refactoring can change classes or even remove them, affecting this value.
- **M10**. **Total Lines of Code (TLOC)** Showing the difference in code size, could indicate the size impact of the refactoring operation on the code. Since the refactored code will be functionally equivalent and in the same language, Total Lines of Code are a valid metric to be used (as compared to e.g. Function Points, used to compare the same functionality in different applications).

4 Experiment Planning

In this section we introduce the characteristics of the empirical experiment conducted for this study. In particular, in Section 4.1 the variables of the experiment are reported, Section 4.2 documents the underlying hypotheses, Section 4.3 describes the subjects of the experiment, and Section 4.4 details the setting in which the experiment was conducted.

4.1 Variable Selection

From the GQM, the following dependent variables can be extracted: *Energy Consumption*, *Power consumption*, *Time*, *CA*, *CE*, *DIT*, *LCOM*, *WMC*, *NOC* and *TLOC*.

These variables are influenced by changing the only independent variable for this experiment, namely *Refactored Code Smells* (RCS). Since JDeodorant allows for searching and refactoring of five different code smells, it is possible to refactor each of them individually. It is however also possible to refactor multiple types of code smells in one single combination. This gives us a total of $2^5 = 32$ possible combinations. However, it seems more appropriate to mainly look at each individual code smell in order to find the impact of each code smell in isolation. In addition, we will determine the impact of a complete refactoring of all five code smells, to simulate an "aggressive" refactoring strategy. This leads us to the following list of 7 considered refactoring values:

- **RCS-0**: No refactoring of any code smells (original code). This will provide the baseline with all possible code smells still existing in the source code of the applications to be tested.
- **RCS-1**: Refactoring code smell Feature Envy.
- **RCS-2**: Refactoring code smell Type Checking.
- **RCS-3**: Refactoring code smell Long Method.
- **RCS-4**: Refactoring code smell God Class.
- **RCS-5**: Refactoring code smell Duplicated Code.
- **RCS-6**: Refactoring all 5 code smells.

The order of refactoring will be randomized to make sure that it does not influence the results.

Below we provide a brief description of each code smell, as well as the related findings presented in the literature w.r.t. their potential impact, which led to their inclusion[1]:

- **Feature Envy** is defined by Fowler [18] as a method that is placed in the wrong class. This happens when a method is invoking multiple 'get' functions, which indicates that the method is more "interested" in another class than the one it is currently located in. The solution to overcome this is placing the method in the most invoked class instead. In a 2010 paper by D'Ambros et al. [10], Feature Envy turned out to be the most occurring code smell in their six tested application. This indicates that this code smell is likely to have a noticeable impact on energy consumption and performance.
- **Type Checking** is explained in detail by the creators of JDeodorant [46]. Although it was not defined by Fowler, the original concept comes from his original code smells work. Type Checking occurs in 'switch' statements and 'if/else if' blocks, where a so called *typefield* is checked for different values and the according branch is then executed. Refactoring is done by changing the typefields with a state class, as defined by Fowler [18]. Such refactoring might potentially lead to drastic energy and performance optimizations.
- **Long Methods** are methods that have gotten too large, impacting negatively readability and maintainability [18]. The solution is to refactor the single long method into smaller methods. Results show that Long Method code smells occur more often in later version of applications [5]. If Long Methods are refactored the number of method calls will increase, since a single method is split up into multiple ones. If Long Methods are refactored, a noticeable performance or energy consumption impact might be measured.
- A **God Class** is a class that implements a large share of the application logic or controls a significant part of the architecture [37]. This primarily leads to maintainability issues. God Classes should be refactored into smaller classes. Perez-Castillo and Piattini [31] examined the impact of God Classes on power consumption. They came to the conclusion that refactoring God Classes resulted in an increase of power consumption due to an increase in messages exchanged between the new classes created. As the experiment utilized exclusively two applications, we decided to verify their findings by adopting different subjects.

[1]The smell selection process was also influenced by the feasibility of refactoring the smells automatically through tools, as a manual approach resulted to be exceptionally time-consuming, error prone and difficult to replicate.

- **Duplicated Code** refers to the duplication of the same code structure in more than one place of the code, mostly occurring due to copy and pasting [18]. Refactoring is usually carried out by unifying the separate structures into one. An approach to do so, independent of coding language, is proposed by Ducasse et al. [11]. Their results show that duplicated code is quite frequent. No studies seem to investigate the impact on performance or energy consumption of Code Duplication. However, as evidence shows that code duplication occurs frequently, refactoring of this smell is likely to have an impact.

Other independent variables are related to the software and hardware configuration of the test bed. To prevent these variables from acting as confounding factors, our experimental environment has been fully controlled (see Section 4.4). Another possible confounding factor is the application workload. Since each application is different, the workload was simulated with a test scenario testing the main functionality of each application. However, we ensured that the workload was exactly the same for each experimental unit by scripting the execution of the test scenario for each application.

4.2 Hypothesis Formulation

RQ1: What is the impact on Energy Consumption when refactoring Code Smells in Open Source Java ORM-based applications?

- $H1_0$: Null hypothesis. For every possible comparison of refactoring, the energy consumption remains the same.
- $H1_1$: Alternative hypothesis. For every comparison of refactoring, there exists a pair where the energy consumption is not the same.

RQ2: What is the impact with respect to Performance when refactoring Code Smells in Open Source Java ORM-based applications?

- $H2_0$: Null hypothesis. For every possible comparison of refactoring, the execution time remains the same.
- $H2_1$: Alternative hypothesis. For every comparison of refactoring, there exists a pair where the execution time is not the same.

RQ3.1: Which object oriented software metrics show a correlation with Energy Consumption when refactoring Code Smells in Open Source Java ORM-based applications?

- $H3.1_0$: Null hypothesis. The correlation index between the metric values and the energy consumption value is the same for each application.
- $H3.1_1$: Alternative Hypothesis. There exists a correlation index for one of the applications between the metrics values and the energy consumption value that differs from the other applications.[2]

RQ3.2: Which object oriented software metrics show a correlation with performance when refactoring Code Smells in Open Source Java ORM-based applications?

[2]For example, if two applications show a significant positive correlation while the third has a significant negative correlation, the Null hypothesis will be rejected.

- $H3.2_0$: Null hypothesis. The correlation index between the metrics and the execution time is the same for each application.
- $H3.2_1$: Alternative Hypothesis. There exists a correlation index for one of the applications between the metrics and the execution time value that differs from the other applications.

4.3 Subject Selection

Our subjects are Open Source, ORM-based Java applications. The choice of Open Source is a prerequisite as we need access to the source code for performing refactoring operations. Within this scope, we chose to focus specifically on ORM-based applications for multiple reasons: first of all, they represent a large share of business software applications. Moreover, large-scale ORM-based applications are typically running in datacenter environments, which would definitely benefit from an improvement in software energy efficiency. In previous work [35] we have already shown that ORM frameworks have a significant impact on energy consumption. As such, we expect that refactoring ORM-based applications will have an impact on energy consumption too. The most popular Java ORM framework is Hibernate [21], which is developed and maintained by Red Hat Inc. [36]. Therefore, all applications that will be tested are making use of the Hibernate framework. Our choice of the Java language is motivated by the tool we selected to refactor code smells, JDeodorant, which is only available for Java.

Our experiment will test three different applications. During the subject selection process, applications that were too large resulted very unpractical to apply code smell refactoring as JDeodorant can only refactor a single code smell at the time. Since this is done in an automatic fashion, there is no guarantee that the refactoring does not compromise the application functionality. Hence, after each refactoring we had to perform an assessment of the application before refactoring another code smell or rolling back to the previous refactoring. Therefore, the selected subjects have to be relatively limited in terms of project size.

The first selected application is called CashManager [39]. This is a small Open Source application available on Github, composed by roughly 1900 total lines of code. It implements an environment to manage multi-currency money transactions and withdrawals. Information about currencies and transactions has to be input manually, making the application very adaptable, yet less user-friendly. The provided application makes use of Apache Tomcat 6 as application server [3].

The second application is called JTrac [1]. JTrac is a highly customizable issue-tracking web-application typically deployed by small-scale projects. JTrac falls under the Apache Project License Version 2.0. The JTrac application has roughly 14000 total lines of code, which makes it the largest application under test.

The third selected application is Spring-PetClinic [33]. This application showcases some possibilities of using the Spring Framework [42], which provides elements for the UI of the application. It is an actively maintained project used by many beginning developers to create their own applications from. The application runs on Apache Tomcat 7 [3]. PetClinic has around 2000 lines of code, which makes it comparable in size to CashManager.

4.4 Instrumentation

Refactoring was performed by means of the JDeodorant [14] plugin for Eclipse IDE. JDeodorant has been deemed as "the only software currently available able to provide code smell detection and then to suggest which refactoring to apply to remove on the detected smells" [17]. The developers of JDeodorant provided extensive documentation on how the detection of certain

code smells has been implemented [46, 15, 16]. In order to refactor 'Duplicate Code' smells a third party tool is needed to detect clones. For this purpose, an Eclipse plugin called ConQAT [9] was used. In order to realize the test scenarios for our experimentation, we made use of a tool called LoadComplete [41]. This latter capture and replay tool supports semi-automated load, stress, and scalability testing for websites and web applications. User scenarios were recorded using the FireFox browser.
The subject applications ran on virtual machines with Windows Server 2008 installed. All the VMs were deployed on a single server, namely an HP DL360 G7 with an Intel Xeon CPU at 2.53GHz and 36GB of RAM, managed via the VMware vSphere 6.0 hypervisor. The hypervisor was also used to gather data about the CPU usage of the virtual machines. Power consumption of the server was measured via a Wattsup PRO Meter, which is capable of measuring power values once per second with an accuracy of 1.5% [12].

5 Experiment Design and Execution

The first step of our experiment is performing the refactoring for each application and treatment. For the individual code smell refactoring, the procedure was as following: i) select the application and smell, then ii) refactor all the detected occurrences of the smell using JDeodorant and ConQAT (for clone extraction) one by one, and finally iii) assess if the resulting application is able to execute the test scenario correctly.

For the combined refactoring of all code smells a different procedure was devised. In order to make sure that our results would not get affected by the order of refactoring, combinations were chosen at random for each trial. To perform this randomization we used the Fisher-Yates algorithm [13].

The number of code smells refactored per application can be found in Tables 1, 2 and 3. The names of the code smells have been abbreviated as follows: Feature Envy (FE), Type Checking (TC), Long Method (LM), God Class (GC) and Duplicate Code (DC). Note that the tables report the number of code smells that could be refactored without impairing the functionality of each application. In most cases, more code smells could be refactored automatically but with side effects and loss of functionality as a result. Note that for the PetClinic refactoring, no 'Type Checking' or 'Duplicate Code' code smells were identified. In total we collected 70 samples for CashManager, 70 samples for JTrac and 75 samples for Spring-petClinic. The total amount of samples is 215, which ensures adequate statistical power.

Table 1: Number of code smell refactorings for the CashManager Application

Combination order	**FE**	**TC**	**LM**	**GC**	**DC**
Single Refactoring	1	3	10	2	1
FE-TC-DC-GC-LM	1	3	10	2	0
FE-GC-TC-DC-LM	1	3	0	2	1
FE-GC-LM-DC-TC	1	3	10	2	0
TC-FE-GC-DC-LM	1	3	9	2	1
LM-GC-DC-FE-TC	1	3	10	2	0
LM-DC-GC-FE-TC	1	3	10	2	0
GC-FE-LM-TC-DC	1	3	10	2	0
GC-LM-FE-DC-TC	1	3	10	2	0
GC-DC-FE-LM-TC	1	3	9	2	1
DC-LM-TC-GC-FE	1	3	9	2	1

Table 2: Number of code smell refactorings for the JTrac Application

Combination order	FE	TC	LM	GC	DC
Single Refactoring	8	3	43	11	3
FE-TC-DC-LM-GC	8	3	30	10	3
FE-GC-LM-TC-DC	8	3	31	11	3
FE-GC-DC-TC-LM	8	3	32	11	3
TC-FE-GC-LM-DC	8	3	28	10	3
LM-TC-GC-FE-DC	8	3	43	9	3
LM-TC-GC-DC-FE	8	3	43	9	3
LM-GC-DC-FE-TC	8	3	43	8	3
GC-TC-LM-DC-FE	8	3	30	11	3
DC-FE-LM-GC-TC	8	3	28	9	3

Table 3: Number of code smell refactorings for the PetClinic Application

Combination order	FE	TC	LM	GC	DC
Single Refactoring	2	-	10	2	-
FE-LM-GC	2	-	10	1	-
FE-GC-LM	2	-	10	2	-
LM-FE-GC	5	-	10	1	-
LM-GC-FE	5	-	10	1	-
GC-FE-LM	5	-	10	2	-
GC-LM-FE	5	-	10	2	-

The software metrics that are linked to the refactoring of individual code smells are found by using the Eclipse Metrics 1.3.8 Plugin[3]. The values for the measured metrics per application and treatment can be found in Tables 4, 5 and 6 below. Note that the total lines of code metric (TLOC) metric does not count blank lines and comment lines inside the code.

Table 4: Software Metrics values for CashManager

	CA	CE	DIT	LCOM	WMC	NOC	TLOC
Orig.	6.667	3.5	1.267	0.265	8.5	5	1859
FE	8.5	4.167	1.267	0.265	8.533	5	1900
TC	8.5	5.333	1.405	0.189	6.476	7	1964
LM	6.667	3.5	1.267	0.267	8.833	5	1901
GC	7.667	3.833	1.25	0.272	8.094	5.333	1903
DC	6.667	3.5	1.267	0.265	8.467	5	1849

In terms of statistical hypothesis testing, for hypotheses H1 and H2 we applied the Analysis of Variance (ANOVA). Hypotheses H3.1 and H3.2 were tested by means of Pearson and Spearman correlation coefficients. We also estimate the magnitude of the observed differences using the Cliffs Delta (or d), a non-parametric effect size measure for ordinal data [20]. For all of our test, a significance level $\alpha = 0.05$ was adopted.

Table 5: Software Metrics values for Jtrac

	CA	CE	DIT	LCOM	WMC	NOC	TLOC
Orig.	6	5.714	3.988	0.241	11.257	8.143	14372
FE	6.048	5.714	3.988	0.241	11.263	8.143	14378
TC	6	8.048	3.511	0.185	8.879	10.619	14736
LM	6	5.714	3.988	0.246	11.708	8.143	14598
GC	6.143	5.952	3.828	0.233	10.829	8.619	14546
DC	6.048	5.81	4.035	0.238	11.15	8.238	14377

Table 6: Software Metrics values for PetClinic

	CA	CE	DIT	LCOM	WMC	NOC	TLOC
Orig.	4.818	4.364	1.465	0.219	4.884	3.909	2009
FE	4.818	4.364	1.465	0.219	4.884	3.909	2005
LM	4.818	4.364	1.465	0.227	5.116	3.909	2047
GC	5.182	4.636	1.435	0.223	4.804	4.182	2086

Table 7: Full overview of our energy consumption results

Application (Watthours)		**Treatment**						
		Orig.	FE	TC	LM	GC	DC	All
JTrac	Median	11.68	5.85	11.12	5.85	10.10	9.71	10.43
	% change	-	-49.9	-4.9	-49.9	-13.5	-16.9	-10.7
	Eff. Size (d)	-	large	large	large	large	large	large
CashManager	Median	8.33	8.67	8.52	8.25	8.47	8.47	8.57
	% change	-	+4.2	+2.3	-0.9	+1.7	+1.7	+2.9
	Eff. Size (d)	-	small	negl.	small	negl.	negl.	negl.
PetClinic	Median	9.91	9.99	N/A	9.87	9.83	N/A	9.79
	% change	0	+0.7	N/A	-0.4	-0.8	N/A	-1.2
	Eff. Size (d)	-	negl.	negl.	negl.	negl.	negl.	negl.

6 Results

6.1 RQ1: Energy consumption impact of code smells

ANOVA testing on the energy consumption data shows a significant difference ($p < 2.2 \cdot 10^{-16}$) in the JTrac application. Figure 1 shows that such application consumed approximately half the energy when the code smells 'Feature Envy' and 'Long Method' were refactored. For JTrac the Null hypothesis $H1_0$ is hence rejected, as no significant change in energy consumption was observed. For CashManager and PetClinic, the Alternative hypothesis $H1_1$ has instead to be rejected.

6.2 RQ2: Performance impact of code smells

The only significant difference regarding performance was observed on the JTrac application ($p < 2.2 \cdot 10^{-16}$). Such trend is depicted in Figure 2. The individual refactoring of the code smells 'Feature Envy' and 'Long Method' differed noticeably in terms of energy efficiency if compared to the original and the other code smell refactoring options for the JTrac application. The Null hypothesis $H2_0$ is hence rejected for JTrac, as significant change in execution time

[3] http://metrics2.sourceforge.net/

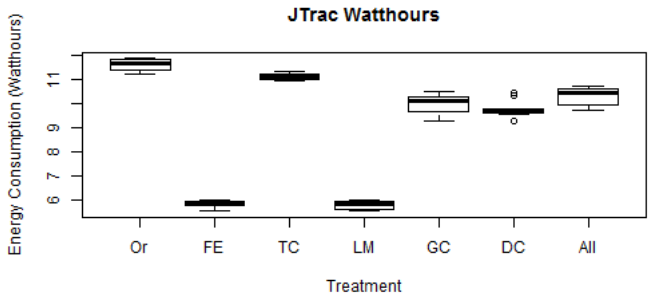

Figure 1: Boxplot showing Energy consumption in Watts for JTrac experiment per treatment.

is present. The Alternative hypothesis $H2_1$ is instead rejected for both the CashManager and Petclinic application.

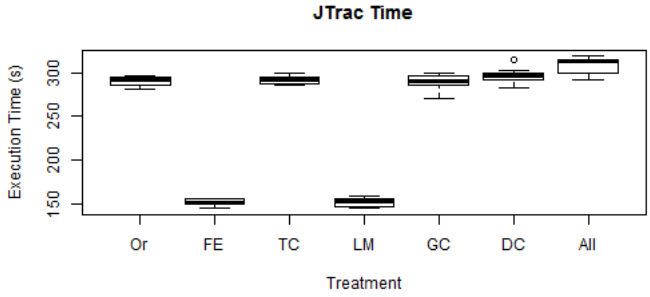

Figure 2: Boxplot showing execution time in seconds for JTrac experiment per treatment.

6.3 RQ3: Software Metrics as indicators

We calculated the correlation coefficients between each metric and Power consumption, Performance, Energy consumption and CPU usage. Such coefficients were found significant only for the CashManager application, specifically for the correlation between performance and Afferent Coupling (CA) ($p = 0.0062$), Efferent Coupling (CE) ($p = 0.0206$) and Total Lines of Code (TLOC) ($p = 0.0428$). In all cases, the correlation coefficient indicated a quite strong positive correlation between the variable and the metric (see Table 8). However, we did not observe consistent values emerging across all the applications for the same metrics. We can therefore confidently reject the alternative hypotheses $H3.1_1$ and $H3.2_1$.

Table 8: Significant correlation coefficients observed in our experiment

Application	**Variable**	**Software Metric**	**Coefficient**
CashManager	Performance	Afferent Coupling (CA)	0.9350
CashManager	Performance	Efferent Coupling (CE)	0.7383
CashManager	Performance	TLOC	0.5182

7 Discussion

One of the main insights relevant for practitioners that emerge from our results regards the answer of RQ3 on the role of software metrics. We did not observe any significant correlation consistent across all our selected applications. As of today, empirical research is still unable to provide reliable indicators for software energy efficiency that might remove the need for empirical energy measurements collected from physical machines.

> **Finding 1.** No reliable indicators of the impact on energy consumption and performance of refactoring code smells could be identified. Physical measurements are hence needed to reliably assess and evaluate such impact.

However, if we do not consider the signif icance of the correlation coefficients, which might be impaired by the limited sample size, in some cases we did observe some patterns. For example, the DIT metric has a negative correlation with power consumption in each application, whereas the lack of cohesion in methods (LCOM) metric has a positive correlation with CPU usage. The patterns that emerged from this exploratory study should hence be investigated further by considering a larger number of applications.

A more detailed analysis of the results revealed a clear impact on the JTrac application, namely a significant reduction in energy consumption when refactoring any code smell compared to the original application. Table 7, reports the median energy consumption and the change percentage in energy consumption per treatment. We observed a maximum reduction of 49.9% obtained by refactoring code smells 'Feature Envy' and 'Long Method' smells. This provides novel evidence of the deep impact that code smell refactoring can have in energy optimization processes (see Section 2).

> **Finding 2.** Code smell refactoring processes resulted to significantly reduce the energy consumption of our selected software application.

Table 9: JTrac Power Consumption per treatment

JTrac	**Treatment**						
(Watts)	Orig.	FE	TC	LM	GC	DC	All
Median	143.92	138.00	137.37	137.45	124.90	118.20	120.10
% change	0	-4.1	-4.5	-4.5	-13.2	-17.9	-16.5
Eff. Size	-	large	large	large	large	large	large

We also observed a significant impact w.r.t. performance in the JTrac application. By inspecting the distributions depicted in Figure 2, we can see that indeed a large change in performance exists for refactoring either code smell 'Feature Envy' or code smell 'Long Method'. Nevertheless, based on the gathered data, we can conjecture that refactoring a combination of all the different code smells selected results on average in a slightly higher execution time. Table 10 shows the median execution time and the difference in percentages per treatment. ANOVA testing on the power consumption data indicates that only the JTrac application shows significant change among all refactorings ($p < 2.2 \cdot 10^{-16}$). The boxplot in Figure 3 illustrates the power consumption values of JTrac for all refactorings. By combining these results with those shown in Table 9, we can observe that the impact of each smell is different in nature:

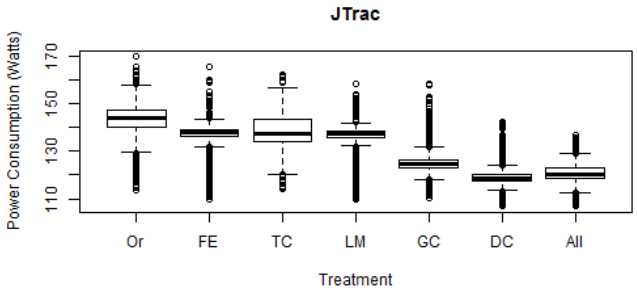

Figure 3: Boxplot showing Power consumption in Watts for JTrac experiment per treatment.

for some smells (namely Feature Envy and Long Method) the energy impact is mostly due to a performance-related improvement. However, for code smells God Class and Duplicated Code we see either a negligible decrease or even a significant increase in execution time, but a significant decrease in power consumption, which in turn determines the decrease in total energy consumption. This indicates that refactoring such smells might implicate a tradeoff between performance and energy efficiency. This also demonstrates that, when refactoring multiple smells at once, the interaction might have to be carefully monitored to avoid the effects to negate one another. This could ultimately result in neglecting any improvement, or even deteriorating the application performance or energy efficiency.

Finding 3. The impact of refactoring code smells can be different depending on the smells. Some resulted to introduce performance improvements, other to reduce power consumption. A combined refactoring of multiple code smells can hinder such improvements.

Table 10: JTrac Performance values per treatment compared to original values

JTrac	**Treatment**						
(Seconds)	Or	FE	TC	LM	GC	DC	All
Median	292.5	152.5	291.5	153.0	291.0	296.5	312.5
% change	0	-47.9	-0.3	-47.7	-0.5	1.4	6.9
Eff. Size	-	large	negl.	large	negl.	large	large

From the results we can observe that JTrac was the only application where a significant impact of refactoring code smells is present. This leads to the question of what makes this application different w.r.t. the other ones considered. A potential factor could be the *size* of JTrac, with around 14000 total lines of code compared to roughly 2000 total lines of code of the other applications. Size also impacts the probability of detecting code smells in the code: from Tables 1, 2 and 3 we can already see that JTrac contains significantly more smells compared to either the CashManager or PetClinic application. This could explain the higher impact those code smells have on the results.

Another factor could be the *age* of the application. As mentioned by Fowler [18], code smells manifest themselves more often in older code for various reasons. Since PetClinic is a showcase application, its code is very clean. CashManager is a small application that is approximately one year old. JTrac on the other hand was created around 2006, hence almost 10 years ago.

This can be a reason for the higher amount of code smells detected. This finding is also in accordance with previous studies [6] showing that the amount of smells seems to increase with the age of the project.

Finding 4. The impact of refactoring code smells depended on multiple factors, such as *size* and *age* of the application.

8 Threats to Validity

In this section we present the threats to the validity of our experiment and our strategy to mitigate them. The classification of the threats follows the one described by Cook et al. [8].

8.1 Internal validity

- **History**: History threats to validity occur when results depend upon the order or the specific moment when the tests were executed. To mitigate these threats, we configured LoadComplete so that all forms of cache and history were cleared upon each run. Additionally, when needed by the applications, a database reset was performed after each trial. In addition, when applying multiple treatments i.e. code smell refactorings, we performed a randomization of the treatments to average any possible effect resulting from a specific order of refactoring.

8.2 Conclusion validity

- **Reliability of treatment implementation.** As mentioned in Section 4.4, the code smell refactoring was performed on our application subjects by means of the JDeodorant tool. On one hand, this ensures replicability of the results. On the other hand, we cannot guarantee that using a different tool for refactoring the smells would give similar results. Nevertheless, the implementation of the refactoring processes is thoroughly reported in the tool documentation [46, 15, 16]. In addition, to the best of our knowledge, JDeodorant is to date the only tool that supports automated detection and refactoring of the smells we selected as treatments [17].

- **Random disturbances**: If during a test trial disturbances are caught by either LoadComplete or the tester, the test were removed from the results and repeated. In order to keep these disturbances to a minimum a closed environment was used, only allowing the tester to use the system. This mitigated most of the random disturbances that could interfere with the tests.

8.3 Construct validity

- **Construct definition**: our experimentation targets code smells, a well-established concept in Software Engineering. However, an exact, formal definition of code smell is not available in the general literature, and in general the identification of a smell is subject to expert opinion and human intuition [18]. Hence, we cannot claim that the code smell detection and refactoring operations performed are formally correspondent to a theoretical, if any, concept of code smell. To mitigate this threat, we refer to the most prominent

sources in the literature related to code smells, and we use the best tool available (to the best of our knowledge) to detect and refactor the smells.

- **Confounding levels of construct**: The treatment factor only takes into account refactoring all the possible refactoring of the selected code smell as a treatment. No testing is done on refactoring just one of the possibilities within one code smell (e.g. only refactor one single Feature Envy code smell instead of all the Feature Envy code smells). Doing so could give more insight in which type of the specific refactoring result in better results. Size of the application is also not taken into account (although partially mentioned in the form of total lines of code as a software metric). In addition it is possible to perform clone detection by using various third party applications. ConQAT was chosen as the application to do in this paper. It could be possible, even though unlikely, that different tools find different clones. This could alter results and cause a possible threat to validity.

8.4 External validity

- **Subject selection.** In our study we considered exclusively three software applications. Given our factorial design, selecting a higher number of applications would have multiplied our experimental units. In addition, functionality correctness testing (due to automated refactoring) and semi-automated load testing result to be rather time-consuming processes. Our necessary selection implies that we cannot guarantee that our subjects are representative of the whole population of Open-Source ORM-based Java software applications. To mitigate this risk, we chose our application subjects to be as representative as possible, by varying their application domain and properties. Nevertheless, due to the low number of subject considered, the results presented in this exploratory study have to be considered as promising introductory results on which future research activities will be based.

- **Experimental setting.** We conducted our experiments in a controlled environment. Hence, we cannot guarantee that our results would be the same in a different setting, e.g. the production environment of a company. However, our test machine (the HP ProLiant DL360 G7) and software environment (vSphere 6 hypervisor and VM configuration) were as up-to-date as possible and they are widely used in industrial settings as well. In addition, in order to mitigate possible threats related to the use of dedicated VMs, the load test executions were throughly monitored and baseline measurement were considered in order to rigorously isolate the impact of the treatments.

9 Verifiability

In this experiment, we targeted a largely un-investigated topic, namely the quantification of the the impact on performance and energy consumption of code smells. More in general, the pioneering field of software energy efficiency still requires much more empirical evidence for providing solid foundations and principles [47]. For this reason, we make all our raw data, scripts and results publicly available for replication or reproduction purposes[4]. Our goal is to strongly encourage other researchers to contribute to this field of research, by reproduction and replication of our results. We are confident this will stimulate the community towards new and interesting findings.

[4] www.s2group.cs.vu.nl/ICT4S2018ReplicationPackage

10 Conclusion

This paper presents an empirical experiment on Open Source Java ORM-based applications, aimed at investigating the performance and energy impact of code smell refactoring processes. Our results show that for the largest among the selected applications refactoring a set of prominent code smells resulted in up to 49.9% lower energy consumption as compared to the original code, whereas performance was improved up to 47.8% in the best scenario. Interestingly, refactoring all of the code smells resulted in a performance decrease of about 6.8% which still led to a 10.7% lower energy consumption. This provides evidence to support the thesis that performance and energy consumption are not always directly correlated. We also investigated whether software metrics could be used as indicators for such impact. The metrics we selected did not significantly correlate with energy consumption or performance values. We can hence conjecture that the selected software metrics cannot be used as reliable indicators of the impact of code smells. However, this result can be due to our reduced sample size. In the future, we will investigate further this aspect by selecting more applications and introducing additional variability (e.g. selecting different versions). In conclusion, we proved that refactoring code smells can lead to a significant reduction of energy consumption of software applications. Combined with the already available evidence on the benefits on code maintainability, our findings suggest that refactoring code smells positively impacts two important dimensions of sustainability [25]: the environmental dimension, by reducing software energy impact, and the technical dimension, by making software more maintainable over time. Based on the results gathered through this exploratory study, we plan to further investigate the impact of code smells on energy efficiency by considering a larger set of heterogeneous software applications and code smells.

References

[1] JTrac. http://www.jtrac.info/. Accessed: 04 April 2017.

[2] F. Alizadeh Moghaddam, G. Procaccianti, G. A. Lewis, and P. Lago. Empirical validation of cyber-foraging architectural tactics for surrogate provisioning. *The Journal of systems and software*, 138(4):37–51, Apr. 2018.

[3] Apache Tomcat. http://tomcat.apache.org/. Accessed 22 June 2017.

[4] V. R. Basili and H. D. Rombach. The tame project: Towards improvement-oriented software environments. *Software Engineering, IEEE Transactions on*, 14(6):758–773, 1988.

[5] A. Chatzigeorgiou and A. Manakos. Investigating the evolution of bad smells in object-oriented code. In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pages 106–115. IEEE, 2010.

[6] A. Chatzigeorgiou and A. Manakos. Investigating the Evolution of Bad Smells in Object-Oriented Code. In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pages 106–115. ieeexplore.ieee.org, 2010.

[7] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

[8] T. D. Cook and D. T. Campbell. *Quasi-experimentation: Design & analysis issues for field settings*. Rand McNally, 1979.

[9] CQSE GmbH. Cqse conqat. https://www.cqse.eu/en/products/conqat/overview/. Accessed 23 June 2017.

[10] M. D'Ambros, A. Bacchelli, and M. Lanza. On the impact of design flaws on software defects. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 23–31. IEEE, 2010.

[11] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 109–118. IEEE, 1999.

[12] Electronic Educational Devices. Watts up? Meter. https://www.wattsupmeters.com/secure/products.php?pn=0. Accessed 22 June 2017.

[13] R. A. Fisher, F. Yates, et al. Statistical tables for biological, agricultural and medical research. *Statistical tables for biological, agricultural and medical research.*, (5th rev. ed), 1957.

[14] M. Fokaefs, T. Chaikalis, G. Panamoottil Krishnan, K. Dextras-Romagnino, R. Stein, and Z. Valenta. Jdeodorant, 2007.

[15] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou. Jdeodorant: Identification and removal of feature envy bad smells. In *ICSM*, pages 519–520, 2007.

[16] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. Jdeodorant: identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1037–1039. ACM, 2011.

[17] F. A. Fontana, M. Mangiacavalli, D. Pochiero, and M. Zanoni. On experimenting refactoring tools to remove code smells. In *Scientific Workshop Proceedings of the XP2015*, page 7. ACM, 25 May 2015.

[18] M. Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 2009.

[19] M. Gottschalk, J. Jelschen, and A. Winter. Saving Energy on Mobile Devices by Refactoring. In *EnviroInfo*, pages 437–444. enviroinfo.eu, 2014.

[20] K. J. Goulden. Effect Sizes for Research: A Broad Practical Approach. *J. Dev. Behav. Pediatr.*, 27(5):419, Oct. 2006.

[21] Hibernate. http://hibernate.org/. Accessed 24 June 2017.

[22] A. Hindle. Green mining: A methodology of relating software change to power consumption. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*.

[23] A. Hindle. Green mining: a methodology of relating software change and configuration to power consumption. *Empirical Software Engineering*, 20(2), 2015.

[24] J. Koomey. Growth in data center electricity use 2005 to 2010. *A report by Analytical Press, completed at the request of The New York Times*, page 9, 2011.

[25] P. Lago, S. A. Kocak, I. Crnkovic, and B. Penzenstradler. Framing sustainability as a property of software quality. *Communications of the ACM*, 58(10):70–78, Oct. 2015.

[26] J. A. Laitner and M. Berners-Lee. Smarter 2030: ICT Solutions for 21st Century Challenges. Technical report, Global e-Sustainability Initiative, July 2015.

[27] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 2–11, New York, NY, USA, 2014. ACM.

[28] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[29] S. Mingay. Green it: the new industry shock wave. *Gartner RAS Research Note G*, 153703:2007, 2007.

[30] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement*, pages 390–400. IEEE Computer Society, 2009.

[31] R. Perez-Castillo and M. Piattini. Analyzing the harmful effect of god class refactoring on power consumption. *Software, IEEE*, 31(3):48–54, 2014.

[32] G. Pinto, F. Castor, and Y. D. Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 22–31. ACM, 2014.

[33] I. Pivotal Software. Pet clinic. https://github.com/spring-projects/spring-petclinic. Accessed 04 May 2017.

[34] G. Procaccianti, H. Fernandez, and P. Lago. Empirical Evaluation of Two Best-Practices for Energy-Efficient Software Development. *J. Syst. Softw.*, 117(July 2016):185–198, 2016.

[35] G. Procaccianti, P. Lago, and W. Diesveld. Energy efficiency of orm approaches: an empirical evaluation. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM/IEEE, 2016.

[36] Red Hat Inc. http://www.redhat.com/. Accessed 24 March 2017.

[37] A. J. Riel. *Object-oriented design heuristics*, volume 335. Addison-Wesley Reading, 1996.

[38] A. Rodriguez, M. Longo, and A. Zunino. Using bad smell-driven code refactorings in mobile applications to reduce battery usage. *Simposio Argentino de*, 2015.

[39] Rubinatrix. Cash manager. Accessed 04 May 2017.

[40] C. Sahin, L. Pollock, and J. Clause. How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 36:1–36:10, New York, NY, USA, 2014. ACM.

[41] SmartBear Software. Loadcomplete. https://smartbear.com/product/loadcomplete/overview/. Accessed 22 June 2017.

[42] Spring Framework. https://projects.spring.io/spring-framework/. Accessed 22 June 2017.

[43] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann, 1 edition edition, 17 Nov. 2014.

[44] G. Szőke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy. FaultBuster: An automatic code smell refactoring toolset. In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pages 253–258, 2015.

[45] A. R. Tonini, L. M. Fischer, J. C. B. de Mattos, and L. B. de Brisolara. Analysis and evaluation of the android best practices impact on the efficiency of mobile applications. In *2013 III Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 157–158. IEEE Computer Society, 1 Nov. 2013.

[46] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 329–331. IEEE, 2008.

[47] R. Verdecchia, F. Ricchiuti, A. Hankel, P. Lago, and G. Procaccianti. Green ICT research and challenges. In *Advances and New Trends in Environmental Informatics*, pages 37–48. Springer, 2017.

[48] A. Vetro, L. Ardito, G. Procaccianti, and M. Morisio. Definition, Implementation and Validation of Energy Code Smells: an Exploratory Study on an Embedded System. In *ENERGY 2013 : The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 34–39. thinkmind.org, 2013.

[49] A. Yamashita and S. Counsell. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software*, 86(10):2639–2653, 2013.