The Impact of Refactoring Code Smells on the Energy Consumption of Java-based Open-source Software

Stephan Kok
stephan.kok@student.uva.nl

August 16, 2019, 53 pages

Research supervisor: dr. A. M. Oprescu, a.m.oprescu@uva.nl
Host/Daily supervisor: R. Genova, MSc.
Host organisation/Research group: KPMG Digital Enablement

UNIVERSITEIT VAN AMSTERDAM
FACULTEIT DER NATUURWETENSCHAPPEN, WISKUNDE EN INFORMATICA
MASTER SOFTWARE ENGINEERING
http://www.software-engineering-amsterdam.nl

Abstract

Code smells are code patterns that highlight code that is difficult to understand or that makes it more difficult to implement a new feature. 24 code smells have been identified by Martin Fowler and have gained the attention of software engineers. While code smells have shown to increase the maintainability of the software, we have to consider what the impact of refactoring these code smells is on the energy efficiency of the software. This work has shown an initial investigation of the impact of refactoring code smells on the energy consumption. It is followed by an empirical study on the impact of refactoring three code smells on the energy consumption. We conducted controlled experiments on three open-source Java projects. The projects have been refactored using JDeodorant as code smell detection tool. We have measured the energy consumption by using a power distribution unit and comparing the difference between the original and refactored version. The workload is created by running all the unit tests in the project. We found that refactoring long methods led to an increase in energy consumption of up-to 3.29%, refactoring large classes has shown an increase of 1.35% and when refactoring for duplicate code when have found both an increase and decrease in energy consumption. This work has shown that refactoring code smells has an impact on the energy consumption and further research is needed to investigate the impact of the other code smells.

Keywords– Energy Consumption, Power Measurements, Code Smells, Refactoring, Empirical Study, DAS, Refactor Tools, Green IT

Contents

Chapter 1

Introduction

Software is part of our daily life and it is in all of our daily products. It is in our cars, TVs, smart-watches, electronic doors, coffee machines and so on. The energy consumption of personal devices is growing with a rate of 5% per year [1]. Our daily use of software is not limited to the physical device. A lot of software is executed on the cloud as large data centers are becoming more popular [2, 3]. These large data centres consume a lot of energy: 270 TWh in 2012 [1] and around 40% of the energy consumption in large data centers is used by the servers [1]. The global greenhouse gas emission from ICT is believed to be 3% - 3.6% by 2020 [4].

Global warming is often in the news and economical and political measures are taken to limit the total energy consumption such as the 'Code of Conduct for Energy Efficiency in Data Centres' [5]. Research has shown an exponentially growing energy consumption caused by the usage of ICT [4], emphasising the need to direct research towards sustainability[6] and making Green IT [7] a hot topic.

Alongside the need to keep the planet healthy, energy consumption is of huge importance for embedded devices such as phones and laptops. A decrease in energy consumption translates to an increase in battery life. There have been numerous papers that focus on decreasing the energy consumption on a hardware level [8, 9]. However, on a software level there is far less research towards energy consumption. A lot of that research is focused mainly on the maintainability of the software.

Writing quality code makes the software easier to understand, helps to find bugs and helps to implement new features faster [10]. As such a lot of research is focused on the quality and maintainability of software. Researchers introduced so called 'code smells' [11], these are code patterns that make software difficult to maintain. These code smells have to be refactored, making the code easier to read and understand. However, these code smells are solely based on the human readability and changeability of code. Refactoring legacy code could also change the performance and energy consumption of the software, raising the question if refactoring the code smells would make the software consume more energy. In this study, we aim to show what the impact of refactoring code smells is on the energy consumption of open-source Java applications.

1.1 Problem statement

Refactoring code smells on existing code has impact on the internal structure of the code. So far only the impact on maintainability of the software has been thoroughly investigated [12, 13]. It is in by definition of refactoring that it does not alter the observable behaviour of the software. But this does not mean that refactoring will not have an effect on the energy consumption of the software. Since developer have to adhere to these rule patterns when writing code, it is important to know if these changes do not increase the energy consumption. When this would be the case, the compiler need to be optimised to transform the code patterns in energy efficient machine code.

As the field of Green Software is growing, research has been conducted to create a methodology of relating software change to energy consumption [14]. However, the research on refactoring code smells on general Java applications is still limited [15–17]. Most research is solely focused on mobile applications or embedded devices [18–21]. Research on the energy consumption of code smells and refactoring methods have been conducted on the C++ language [21–23]. However, research has shown the energy profile of the JVM is different then that of other languages such as C++ [24–27]. Since Java is still one of the most used programming languages [28]. It is important to know the impact of refactoring code smells on general Java application [23, 29].

1.1.1 Research questions

In this research the following research question will be answered:

Research Question What is the impact of refactoring code smells on the energy consumption of Java based open-source applications?

In order to answer this question we divided it in three sub questions. The reason we have selected these specific code smells is explained in Chapter 3. The sub-questions will be answered using an empirical investigation. The three sub questions are as follows:

RQ 1 What is the impact of refactoring 'Long Function' on the energy consumption?
RQ 2 What is the impact of refactoring 'Large Class' on the energy consumption?
RQ 3 What is the impact of refactoring 'Duplicate Code' on the energy consumption?

1.1.2 Research method

We conducted an empirically study on the impact of refactoring code smells on the energy consumption. For our measurement setup we used the proposed methodology from Abram Hindle [14].

The subjects under study are three open-source Java applications. We selected these codebases based on their relevance in other research as well as having a solid testing framework. Using the testing framework we can make sure that refactoring the code smells does not change the outside behaviour and it also provides us with the means of executing the code for energy measurements. The three selected codebases are shown in Table 4.5.

Energy consumption of software is often strongly correlated with performance, however research has shown that this is not always the case [15, 30]. This forces us to measure the energy consumption of software, since we can not calculate the running time or CPU cycles. There are tools available that simulate the energy consumption of software. However, these tools are based on strong assumptions as research is still evolving in the field of modelling energy consumption. For example, it is shown that power consumption is not linearly correlated with CPU utilisation [31]. We have chosen to measure the physical energy consumption of the device. In order to be able to measure the actual energy consumption, we used the Distributed ASCI Supercomputer (DAS). Some of these nodes are powered through a Power Distributed Unit (PDU). Which makes it possible to measure the energy consumption of the software on a system level [32]. A schematic view of the measurement setup is shown in Figure 4.3.

Refactoring code smells is a time consuming process. For this reason automated refactoring tools have been developed [33–39]. We used JDeodorant for the automated detection of code smells. Besides detection JDeodorant even supports automated refactoring suggestions [40–42]. Using JDeodorant we created different refactored versions of the software, enabling us to measure the difference in energy consumption between the refactored and original version. From the obtained data, the energy consumption is visualised using boxplots and for significance testing we used Mann Whitney u Tests.

1.2 Contributions

Our research makes the following contributions:

1. **An energy consumption measurement set-up.** In this research we have created a structure for measuring energy consumption on the DAS. It makes it possible to run instructions on the DAS and measure their energy consumption. The setup was designed in collaboration with Lukas Koedijk and Kees Verstoep and is explained in Chapter 4. All the code is available on Github[1]
2. **Overview of current available code smell refactoring tools.** The accessibility of refactoring code smell tools has changed and as such previous literature is no longer up-to-date. In Chapter 3 we show which free-to-use tools for refactoring code smells in Java are still working and available online. With exception of JSpirit, all of the tools are being maintained regularly and are believed to be operational for the coming years.
3. **Impact of refactoring code smells on the energy consumption.** In Chapter 3 we have proposed assumptions on what the impact of refactoring the 24 code smells (identified by Fowler) will be on the energy consumption.
4. **Empirical investigation in the energy consumption of code smells.** We have shown how the refactoring of 'Long Method', 'Large Class' and 'Duplicate Code' affected the energy consumption for open-source Java applications. The results from the investigation are shown in Chapter 5 and discussed in Chapter 6.

1.3 Outline

In Chapter 2 we describe the needed knowledge and background for this thesis. Chapter 3 describes guidelines for selecting a codebase to investigate the impact of refactoring code smells on the energy consumption. Additional, we will show our assumption on the impact of refactoring code smells on the energy consumption and we have created an overview of the current available code smell refactoring tools. Chapter 4 describes the experimental setup. The results are shown in Chapter 5 and discussed in Chapter 6. Chapter 7 contains the work related to this thesis. In the last Chapter 8, we conclude the work done, alongside with some future work.

[1] https://github.com/stephankok/green-software

Chapter 2

Background

This chapter will present the necessary background information. First, we define some basic terminology that will be used throughout this thesis. Next, we will explain the meaning of refactoring, followed by what code smells are, including a short summary for each smell. For clarifications and to avoid common mistakes, we explain what energy is, how we calculate it from the power readings and how we can properly formalise it. In order to be able to understand what is happening on the machine, we will give an introduction to the Java compiler, Java Virtual Machine and the Just In Time compiler. We will finish with a short introduction on statistical significance.

2.1 Terminology

Refactoring: a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.

Code smell: a pattern in the code that indicates a deeper problem in the system that either decreases readability or adjustability of the code.

DAS: The Distributed ASCI Supercomputer that is used for energy measurements.

Power (Watt): the rate of energy, per unit time.

Energy (Joule): The total work done.

JIT: Just In Time compiler used by Java to optimise code.

PDU: Power Distribution Unit, a device for monitoring and distribution of the electrical power.

P-value: The probability of obtaining a result given that the null hypothesis is true.

2.2 Code refactoring

Refactoring is changing software while keeping its external behaviour the same. The goal of refactoring is to improve the internal structure of the code, making it more readable and easier to implement a new feature. Over time, code tends to be altered and the original code design will slowly disappear [10]. This is why code refactoring should be a fundamental part of the software process.

Martin Fowler gives a concrete definition of refactoring in his book 'Refactoring: improving the design of existing code' [10]. This defnition is as follows:

Refactoring: a change made to the internal structure of software to make it easier to understand and cheaper to modify without chancing its observable behaviour.

There are two important conclusions to be made from this definition. The first conclusion is that a change in the software is only considered a refactoring if it either improves the readability or makes it easier to add a

new feature. The second conclusion is that the change in the software can not alter the functionality. Optimising software with regards to energy consumption can have tremendous impact on the external behaviour of the software. For example using black as a background colour allows pixels to be turned off and save energy [43]. Since the external behaviour is not preserved, such a change in the software can not be considered a refactoring.

Refactoring code is not easy and not without risks. For all the changes that are made, you have to validate if the change does not alter the external behaviour of the application. Additionally, one should not forget the main purpose of the refactoring, namely increasing the maintainability of the software. In order to be able to validate that the changes made did not alter the external behaviour, you need a solid testing framework [10]. Even when using an automated refactoring tool, a full testing framework is needed to make sure that the created refactored code has the same external behaviour.

2.3 Code Smells

Martin Fowler and Kent Beck first came up with the term 'code smell' in 1997 [10]. They created an informal definition of 22 code smells and discussed how to refactor them. Their message was that if something smells bad, change it. However, the naming of 'code smell' could have something to do with Beck's children being in diapers at that time [11].

The purpose of code smells is to give an indication when a certain piece of code is likely to reduce code readability or makes it more difficult to implement a change. As maintainability is getting more attention, Fowler revised his old book and published a new version in 2018. In his new book Fowler explains 24 code smells [11] shown in Table 2.1.

Category	Code smell	Summary
Bloaters	Mysterious Name	Code needs to be straightforward and clear
	Long Function	Long functions should be decomposed
	Large Class	Class has to many methods, field or parameters
	Primitive Obsession	Too many (primitive) variables that belong in a class
	Long Parameter List	Too many parameters in method
	Loops	Loops are depreciating and are being replaced with first-class functions such as filter and map
	Data Clumps	A method requires a lot of variables that could be in a class
Object-Orientation Abusers	Repeated Switches	Makes it more difficult to add features and should be replaced with polymorphism
	Temporary Field	Variables in class are rarely used
	Refused Bequest	Wrongly inherited subclass
	Alternative Classes with Different Interfaces	Classes with similar behaviour should have a common superclass
Change Preventers	Divergent Change	A Class should have only one reason to change
	Shotgun Surgery	A single change should not affect multiple classes
	Global Data	Global data can be modified from anywhere without knowing what caused it, this makes it error prone and more costly to add features
	Mutable Data	Changes to data can lead to unexpected consequences
Dispensables	Comments	Improper comment
	Duplicate Code	Code semantics already exists
	Lazy Element	Rarely used class
	Data Class	Only data in class
	Speculative Generality	Unused method, field or parameter in a class (often premature methods)
Couplers	Feature Envy	Extensive usage of another class
	Insider Trading	Modules should be localised decreasing the messaging between modules as much as possible
	Message Chains	Classes who request a class that requests a class, etc
	Middle Man	One class that only dedicates other classes

Table 2.1: Code smells identified by Martin Fowler [11].

The code smells are grouped in 5 subcategories. Bloaters typically decrease code readability and grow over time. Object-Orientation Abusers make incorrect use of the advantages of Object-Oriented languages. Change Preventers make it difficult to change the code. Dispensables have no purpose in the code and should be removed. Couplers cause additional unwanted message traffic between classes.

Refactoring code smells has shown to improve the maintainability of software [44] and even shown to be correlated with class errors [45]. In order to refactor these code smells, Fowler and Beck give an extensive list of 88 possible refactoring possibilities [11]. For each refactoring method there is a simple example available online[1] created by Martin Fowler. The names of the refactoring methods are shown in List 2.1.

[1] https://refactoring.com/catalog/

1. Change Function Declaration
2. Change Reference to Value
3. Change Value to Reference
4. Collapse Hierarchy
5. Combine Functions into Class
6. Combine Functions into Transform
7. Consolidate Conditional Expression
8. Decompose Conditional
9. Encapsulate Collection
10. Encapsulate Record
11. Replace Record with Data Class
12. Encapsulate Variable
13. Encapsulate Field
14. Extract Class
15. Extract Function
16. Extract Method
17. Extract Superclass
18. Extract Variable
19. Introduce Explaining Variable
20. Hide Delegate
21. Inline Class
22. Inline Function
23. Inline Method
24. Inline Variable
25. Inline Temp
26. Introduce Assertion
27. Introduce Parameter Object
28. Introduce Special Case
29. Introduce Null Object
30. Move Field
31. Move Function
32. Move Method
33. Move Statements into Function
34. Move Statements to Callers
35. Parameterize Function
36. Parameterize Method
37. Preserve Whole Object
38. Pull Up Constructor Body
39. Pull Up Field
40. Pull Up Method
41. Push Down Field
42. Push Down Method
43. Remove Dead Code
44. Remove Flag Argument
45. Replace Parameter with Explicit Methods
46. Remove Middle Man
47. Remove Setting Method
48. Remove Subclass
49. Replace Subclass with Fields
50. Rename Field
51. Rename Variable
52. Replace Command with Function
53. Replace Conditional with Polymorphism
54. Replace Constructor with Factory Function
55. Replace Constructor with Factory Method
56. Replace Control Flag with Break
57. Remove Control Flag
58. Replace Derived Variable with Query
59. Replace Error Code with Exception
60. Replace Exception with Precheck
61. Replace Exception with Test
62. Replace Function with Command
63. Replace Method with Method Object
64. Replace Inline Code with Function Call
65. Replace Loop with Pipeline
66. Replace Magic Literal
67. Replace Magic Number with Symbolic Constant
68. Replace Nested Conditional with Guard Clauses
69. Replace Parameter with Query
70. Replace Parameter with Method
71. Replace Primitive with Object
72. Replace Data Value with Object
73. Replace Query with Parameter
74. Replace Subclass with Delegate
75. Replace Superclass with Delegate
76. Replace Inheritance with Delegation
77. Replace Temp with Query
78. Replace Type Code with Subclasses
79. Extract Subclass
80. Return Modified Value
81. Separate Query from Modifier
82. Slide Statements
83. Consolidate Duplicate Conditional Fragments
84. Split Loop
85. Split Phase
86. Split Variable
87. Remove Assignments to Parameters
88. Substitute Algorithm

List 2.1: Catalogue of refactoring methods to refactor code smells, by Martin Fowler [11]. Examples for each refactoring method is available online[1].

2.4 Power & Energy Consumption

Terms such as power and energy can be confusing and it is really important that they are used correctly. We define the two terms as follows:

Power is the rate energy, per unit time, at which electric energy is transferred through a circuit. Power is usually measured in Watt (W) and is calculated as the voltage (V) times the current (I).

$$P[W] = \{\text{Work done per unit time}\} = V[V] \cdot I[A] \tag{2.1}$$

Energy is the total work done over a time period. Energy is usually measured in Joules (W s).

$$E[J] = \int_0^{T[s]} P[W]\, dt \tag{2.2}$$

In theory, if the power would be constant in time it could be calculated as:

$$E[J] = P[W] \cdot \Delta T[s] \tag{2.3}$$

But this is usually not the case in practice, we will calculate the energy as the sum of the average power between two data-points multiplied by the time between the data-points.

$$E[J] = \sum_n \frac{P[W]_n + P[W]_{n+1}}{2} \cdot (T[s]_{n+1} - T[s]_n) \tag{2.4}$$

kWh (kilo-Watt hour) should not be mistaken with energy (J). Joule is Watt seconds (W s). kWh is not a SI-unit and it will therefore not be used in the paper. However, since it is still used in practice we do provide the conversion from Joule to kWh:

$$1 \text{ kilo-Watt hour } [kWh] = 1,000 \text{ Watt hour } [Wh] = 3,600,000 \text{ Joule } [Ws] \tag{2.5}$$

2.5 Java

Java is a general-purpose object-oriented programming language. It is not tied to any operating system because it is compiled to Java byte code which can be run on different operating systems as is shown in Figure 2.1. Unlike many other compilers, the Java compiler (Javac) makes negligible optimisations. The JVM interprets the Java byte code and translates it to machine-code, the different JVMs make it possible to run Java on different platforms. However, for some architectures the byte code is translated in another way to machine-code such as in Android devices. Inside the JVM there is the Just In Time compiler (JIT) that makes all the optimisations. The JIT compiles byte-code to machine-code while running, rather then prior to execution. The disadvantage is that during the first run you will have extra overhead of compiling the code, instead of pre-compiled code. But after a few runs the compiler can do a lot more optimisations based on the executed code. Since the JVMs are different it matters which package is being used. From all the Java versions there are only two that have long term support: Java 8 and Java 11, whereby Java 8 is dominating the current market [28].

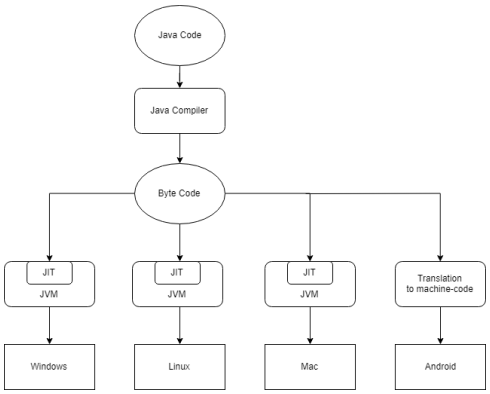

Figure 2.1: Overview of Java

2.5.1 JIT Optimisation's

The JIT makes all the complex optimisations. We will discuss some of the important optimisations for this research: Dead Code elimination, Native Code and Method Inlining. Other known optimisations are: Loop

unrolling, Lock Coarsening, Lock Eliding, Escape Analysis, Perf Sinks (Memory accesses, Calls and Locks), Volatile (Cache per CPU), Call Site, On-Stack Replacement and Code Hoisting. Since the code is optimised at run-time, the compiler knows what code is run often (hotspots) and is able to optimise that particular code with run-time information.

Calculating all the possible optimisations is an expensive process. Therefore JIT compilers usually have two modes: Client Mode (C1) and Server Mode (C2). C1 is meant for short run-times and therefore makes a lot less optimisation's then C2. C2 is a lot more aggressive and makes educated guesses to optimise to a maximum. Making educated guesses means that the optimisations can be wrong and can decrease the performance. When this happens the JIT will deoptimise (bail out) the Native code and make sure that no function will be using this piece of optimised code. In order to get the best of both there is a Tiered mode where both C1 and C2 are combined. The Tiered mode is the most commonly used one.

Dead Code elimination As JIT compiles code while running, code that is never called will never be compiled. However, even if the code is not called the function or class will be held in a registry. Additionally, the JIT compiler can remove code that adds no functionality, such as code after a return statement or a local variable that is never used.

Native Code Native compilation is the most basic optimisation. Interpreting a language is a costly operation since the Java byte code has to be translated to machine code each time it is executed. When a certain piece of code is run often, the code can be pre-compiled from Java byte code to Native code and keep the Native code in memory. This section of memory is also called the 'JVM Code Cache'.

Inlining If certain sequences or trees of functions are called a lot, the JIT compiler will merge these functions together in one large function. An example of this is shown in Listing 2.1. Whether or not the compiler in-lines a certain piece of code depends on a lot of factors. Two important factors are function size and function depth. If a function is too large or the recursion is too high it will not be in-lined because it will cost too much memory. Some of these thresholds are adjustable by the programmers, however this is not common practice. The most used thresholds are: -XX:+MaxInlineLevel, -XX:+MaxRecursiveInlineLevel, -XX:+MaxInlineSize, -XX:+InlineSmallCode and -XX:+FreqInlineSize.

```
int sumUntil(int max) {
    int accum = 0;
    for (int i = 0; i < max; i++) {
        accum = add(accum, i);
    }
    return accum;
}

int add(int a, int b) {
    return a + b
}
```

```
int sumUntil(int max) {
    int accum = 0;
    for (int i = 0; i < max; i++) {
        accum = accum + i;
    }
    return accum;
}
```

Listing 2.1: Example of inlining a function call

2.6 Statistical Significance

Checking if the data is statistically significant excludes the fact that the difference is obtained by random chance. In statistical significance tests there are two hypothesises. The null hypothesis (H_0) and the alternative hypothesis (H_1). If we are able to reject the the null hypothesis we can say that the alternative hypothesis is statistically significantly true. Whether or not we can reject the null hypothesis is given by the Significance Level (α) and the P-value (p). We can say the data is statistically significant when

$$p < \alpha. \tag{2.6}$$

The P-value is the probability of obtaining a result given that the null hypothesis is true. The Significance Level is the probability of rejecting a null hypothesis by the test when it is really true. The significance level must be set before executing the test and is usually set to 0.05 [46, 47].

Chapter 3

Research

This chapter describes guidelines for selecting a codebase to investigate the impact of refactoring code smells on the energy consumption. Furthermore, we discuss the impact of refactoring code smells on the energy consumption and we provide an overview of the available code smell refactoring tools

3.1 Guidelines when looking for codebase

When selecting a codebase to investigate the impact of refactorings on the energy consumption, there are two options: A codebase with existing refactored versions or a codebase without refactored versions. When selecting a codebase with existing refactored versions, there are two requirements for the codebase, shown in List 3.1.

1. **Reference version.** There must be an original version of the software. This will be there reference point for the other versions.
2. **Component unique refactorings.** Each different version must be refactored for a single component. When a version is refactored for multiple components or if it has any other modifications, it will be impossible to measure the impact of refactoring the component.

List 3.1: Requirements for codebase with existing refactored versions

There are not many codebases online available that fit these strict requirements. The few codebases we could find either did not have the data available anymore or did not respond to our messages. For selecting a codebase without refactored versions, there are some different requirements. We set three requirements shown in List 3.2.

1. **Solid testing framework with high code coverage.** As mentioned in Chapter 2, we need high code coverage to ensure that refactoring the code smell does not change the behaviour of the code.
2. **Open-source codebase, known in literature.** To be able to present our data, we need to use open-source software. Additional, for repeatability researchers should be able to access the refactored data.
3. **Size of the codebase.** The codebase can not be too large nor too small. Large codebases will require a lot of refactorings in order to have significant impact on the energy consumption of the application. Additionally, small codebases may not have any code patterns to refactor.

List 3.2: Requirements for codebase without existing refactored versions

We found two online software databases that are most commonly used to gather codebases in emperical research, The Quality Corpus[1] and Github[2]. The Quality Corpus is a collection of Java software packages, in special to be used for repeatable empirical studies [48]. However the latest date of release is 2013, making all the available software quite old. High code coverage was not popular back then, making it difficult to account for requirement 1. Additionally, in order to provide relevant research we want the software to be at least of Java version 8. For these reasons we have recommend to use Github as online software database.

3.2 Impact of refactoring code smells on the energy consumption

In this section we will discuss our assumption with regards to the influence of refactoring code smells on the energy consumption in Java. In order to support our assumptions, we have employed our knowledge of the Java compiler (Section 2.5) and searched literature for similar changes in code patterns. Additionally, research has shown that the computational cost in energy consumption is mainly due CPU processing, memory access and I/O operations [49] and a model for energy consumption on embedded Java devices has shown that 70% of the energy consumption is consumed by memory access on the JVM [50].

Fowler has introduced 24 unique code smell patterns (Section 2.3). From the 24 code smells, there are two that will have no impact on the energy consumption:

Mysterious Name Changing the name of a function will not affect the energy consumption, since function names will be removed on compilation. Additionally this is shown by Park et al. [23] for the C++ language.

Comments Comments will be removed on compilation and changing them will not affect the energy consumption.

In Table 3.1 an overview is shown of 22 code smells and their corresponding refactoring techniques which are identified by Fowler [11]. The column 'Energy' is our assumptions on what the change in energy consumption would be given the refactoring techniques in association with the code smell.

Code smell*	Refactoring Method*	Energy**
Long Function	Extract Function	By extracting a function an extra function call will be added on the stack. This is assumed to increase the energy consumption. However, there are compiler optimisations that can remove this overhead by inlining the code. This leads us to assume that refactoring long function will either have no impact or increase the energy consumption. Dhaka et al. [16], used a modelling tool to investigate the energy consumption of refactoring a long method on Java and found an increase in energy consumption from 0.5% to 7.3%. Additional similar effects have been found in the C++ language, Park et al. [23] found that 'Extract Function' refactoring increased the energy consumption. Contradictory with our assumption, Verdecchia et al. [15] found a single occurrence where energy consumption was decreased by 49.9% after refactoring for long methods in Java.

[1] http://qualitascorpus.com/
[2] https://github.com/

Large Class	Extract Class Extract Subclass Extract Interface Duplicate Observed Data	When extracting a class we have to account for two extra effects: Loading class in memory and creating a new object. On first usage, a class will be loaded in memory. The size of the extra memory depends on the class, but every class will have at least contain some overhead for the optimisations performed by the JIT. Since there is an overhead, splitting a class into smaller classes will always require more memory. As it must be possible to create new object from this class, the class has to stay in memory during the entire execution. This leads us to assume that the energy consumption will increase. Besides the class loading, there is object creation. Since class logic is split, only the required objects have to be created, making it possible to require less memory. This is assumed to decrease the energy consumption. However, if the object is created in combination with the class (which is usually the case) there is an extra overhead from the extra object, increasing the memory being used and increasing the energy consumption. Also there is message traffic. Since class logic is split, there will be additional message traffic. Logic can still be performed locally in the class, but the data has to be accessed from the class somehow. This will add instructions on the stack and is assumed to increase energy consumption.
Primitive Obsession	Replace Data Value with Object Replace Type Code with Class Replace Type Code with Subclass	Boxing is the wrapping of a primitive type in an object. Since Java 1.5 the autoboxing feature has been added to the compiler. However, continuous boxing and unboxing is very expensive and can lead to a significant slower execution time [51]. This is assumed to increase the energy consumption. Bloch explained this line of reasoning by an example shown in Figure A.1. Instead of adding a long value, the program will creates around 2^{31} Long objects induced by autoboxing. From a memory point of view, the wrapping of object requires a lot of extra space. For example an int requires 32 bits, but an Integer requires 128 bits[3]. Increasing the memory required and thus assumed to increase the energy consumption.
Long Parameter List	Introduce Parameter Object Preserve Whole Object	The amount of parameters can be reduced by combining them in an object. This leads to a trade-off between the cost of parsing parameters versus the creation of an object. The creation of an object will increase energy consumption and reducing paramaters will decrease energy consumption. We have not found any research concerning the trade-off analysis, but we can study the cost of Java opcode instructions provided by Lafond et al. [50]. We see a ratio of 1 to 5 when comparing the cost in energy consumption of creating a java.lang.Object against parsing a parameter[4]. By preserving the object one will wait with unwrapping the object until after it is parsed. This will reduce the amount of parsed parameters and it is assumed to decrease the energy consumption.
Loops	Replace Loop with pipeline	Streams are not embedded in Java. However, since Java 8 a stream programming framework has been added. Changing from loops to streams completely changes the underlying execution. Since the logic between conditionals and pipelines are different, it depends on the usage to which extent it affects the performance [52]. This makes it unable to make an assumption on how the energy consumption will change.
Data Clumps	Extract Class	Data clumps behave similar to large classes since logic has to be split by extracting a class. Thus it is assumed to increase energy consumption.

[3] https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html
[4] http://users.abo.fi/slafond/javacosts/

Repeated Switches	Replace Condition with Polymorphism	When using polymorphism there is a constant cost of two pointers that have to be accessed. However, when creating large nested conditionals the cost will keep increasing as there is a need for extra checks. This means that we would expect the energy consumption to decrease when refactoring large switch statements to polymorphism, but the energy consumption to increase when refactoring small switch statements. Similar effects have been found in the C++ language by Serge Demeyer [22]. He found that, depending on the compiler, the polymorphism would be faster after zero to five if-statements.
Temporary Field	Extract Class	Temporary Field behaves similar to large classes since logic has to be split by extracting a class. Thus it is assumed to increase energy consumption.
Refused Bequest	Push down Field Push Down Method Replace with Delegate	Refactoring a Refused Bequest can be compared with removing dead code. This because one removed class functionality that is not being used. When fields are removed, it will decrease the size of the class, reducing the memory needed to load the object and thus reducing the energy consumption. However, Java has dead code eliminations optimisations. This leads us to assume that refactoring Refused Bequest will either have no impact or decrease the energy consumption.
Alternative Classes with Different Interfaces	Change Function Declaration Extract Superclass	Just as with 'Mysterious Name', renaming a method will not impact the energy consumption. But, when two or more classes perform identical logic they can be refactored with extracting a single superclass. When the logic for both classes is in the superclass there are less methods to be compiled and loaded in memory, which is assumed to decrease the energy consumption.
Divergent Change	Extract Class	Divergent Change behaves similar to large classes since logic has to be split by extracting a class. Thus it is assumed to increase energy consumption.
Shotgun Surgery	Move Method Move Field Inline Class Combine into Class	Shotgun surgery is the opposite of 'Divergent Change' and it thus behaves opposite to refactoring large classes. This leads us to assume that refactoring shotgun surgery will decrease the energy consumption.
Global Data	Encapsulate Field	Java is specially build to encapsulate the data, it is therefore likely that the overhead will be optimised by the compiler. However, a method invocation (encapsulating field) is shown to be more expensive then direct data access [50]. While we assume one cannot measure any difference.
Mutable Data	Encapsulate Field Replace Derived Variable with Query Combine Functions into Class Combine Functions into Transform Change Reference to Value	When Encapsulating a field, we expect to see the same change as in 'Global Data' and see no change in the energy consumption. Data should not be pre-calculated (Derived Variable) and should only be calculated on request (Query). Logic will have to be moved to the right method. Since logic still has to calculated, we assume that this will not impact the energy consumption. Combing function in a new class, will behave as 'Large Class' and thus is assumed to increase the energy consumption. Combing functions into a transform is wrapping functions in another function. This will behave like 'Long Function' since an extra function is created and thus is assumed to have no impact or increase the energy consumption. Changing a reference to value, is creating an object from a primitive and thus behaves like 'Primitive Obsession'. This is believed to have no impact or increase the energy consumption.

Duplicate Code	Extract Function Pull Up Method	Extracting a function behaves like 'Long Function' and thus is assumed to have no impact or increase the energy consumption. However, in the case of duplicate code two or more functions will be extracted to one function. Meaning that the total amount of code will decrease. Since there is less code to be compiled from byte-code to machine-code, This is assumed to decrease the energy consumption. It could be that a function becomes 'hot' because it is now used a lot more. When it becomes hot it will be optimised by the compiler, such as for example code being cached in the JVM Code Cache. This is assumed to decrease energy consumption.
Lazy Element	Inline Function Inline Class Collapse Hierarchy	Refactoring a lazy element is the opposite of 'Long Function' or 'Large Class' refactoring and it is thus assumed to decrease the energy consumption.
Data Class	Move Function	Localising the data will decrease the amount of function calls between classes. Since there are less function calls on the stack, this behaves opposite of 'Long Function' and is thus assumed to have no impact or reduce the energy consumption.
Speculative Generality	Inline Function Inline Class Collapse Hierarchy Remove Dead Code	Premature implemented features should be removed. When class or method functionality is not being used it can be inlined. Similar as 'Lazy Element', this behaves in opposite to the refactoring of 'Long Function' and 'Large Class', assuming to see a decrease in energy consumption. The removal of dead code behaves similar to 'Refused Bequest', leading to assume that it will have no impact or see a decrease in energy consumption.
Feature Envy	Move Function	When a function requires more functionality from another class or makes more sense in the other class, it should be moved there. This reduces the amount of traffic between classes, reducing the amount of function calls between classes. This is assumed to reducing the energy consumption.
Insider Trading	Move Function Move Field Hide Delegate Replace Subclass with Delegate Replace Superclass with Delegate	Moving functions behaves as 'Feature Envy' and thus is assumed to reduce the energy consumption. When using Delegation instead of inheritance, the size of the class is reduced and the class only provides access to the other class. Since the purpose of insider trading is to minimise the message traffic between two classes, we assume there will not be a significant overhead caused by the extra function calls between classes. Leading to assume that refactoring will reduce the energy consumption.
Message Chain	Hide Delegate Move Function	Reducing chain of messages will decrease the amount of function calls needed to reach the data. This is assumed to reduce the energy consumption.
Middle Man	Remove Middle Man Inline Function	Sometimes a middle man is created by mistake when refactoring 'Message Chains'. It hides the internal details and is only delegating. By removing the middle man the object can be accessed directly, decreasing the amount of function calls needed and thus assumed to decrease the energy consumption.

Table 3.1: How refactoring code smells impacts the energy consumption in Java

* Code smells and code refactoring techniques identified by Fowler, Section 2.2 and 2.3
** Assumptions on what what the change in energy consumption would be given the refactoring technique.

In order to validate our assumption we have followed with an empirical study. In choosing the code smells which are to be investigated we have considered the following aspects:

1. Refactoring the code smell is assumed to have a impact on the in energy consumption.
2. The code smell is known by developers and thus actually refactored.
3. The code smell is supported by a code smell detection tool.

From the 24 code smells we have shown that two of them will have no impact on the energy consumption. This still leaves us with 22 code smells.

Yamashita et al. investigated the popularity of code smells and anti-patterns by developers [53]. After involving 85 professional software developers, three code smells and one anti-pattern have gotten an observable higher rank: 'Duplicate Code', 'Long Function', 'Accidental complexity' and 'Large Class'. Other code smells that are mentioned but are lower on the rank are: 'Lazy Class', 'Feature Envy', 'Long Parameter List' and 'Mysterious Name'. This reduces the 22 code smells to 6 code smells.

In order to reduce our influence when refactoring we have used tools for refactoring. How the tool has been selected is explained in Section 3.3. From the remaining 6 code smells, JDeodorant can detect 4 code smells: 'Long Function', 'Large Class', 'Feature Envy' and 'Duplicate Code'. However, as explained in Section 4.5, we have not detected 'Feature Envy' in the codebases and thus we have conducted the empirical investigation on 3 code smells: 'Long Function', 'Large Class' and 'Duplicate Code'. This is all explained in Chapter 4.

3.3 Code smell detection tools

There are several tools for automated detection of code smells. There has been research conducted on available tools and what code smells they are able to detect [35–38]. Additional, research has shown which tools are best to use [39]. However, these papers are now outdated. The tools are no longer maintained, no longer working or do not exist any more. We have tried to install 11 of these tools and in the following subsections we described our experience with using these tools on code smell detection in existing codebases.

We require a tool that can refactor code smells from existing Java code for Java version 8. This is why we choose to investigate the following features:

- Code smell support
- Release data
- Covered by Literature
- Automated refactoring
- Detection technique
- Type of tool

The results are shown in Table 3.2.

Fernandes et al [37] found 84 Code Smell detection tools in 2016. 29 of these tools were online available for download. From these 29 tools online available tools there are 8 tools that are free-to-use and detect code smells on Java applications: Checkstyle[5], inFusion[6] , iPlasma[7], JDeodorant[8] , PMD[9], JSpIRIT[10], Stench Blossom[11], and TrueRefactor[12]. Another free-to-use refactoring tool, discussed by Fonatana et al [36], is DECOR[13]. We have found two additional tools namely FindSmells[14] and Sonarqube[15]. FindSmells has been released in 2017 by Sousa et al [54] and Sonarqube [55, 56] is a refactoring tool that is widely used in Industry, but less in research.

We found some difficulties in using these tools. The first problem is that some of these tools are no longer operational. Infusion is developed by Intooitus. However, the company Intooitus does not exist anymore and the tool is no longer available online. DECOR is developed by Ptidej. The tool is not open-source and only available for download as a compressed zip file. However the current version online contains a data compression fault and can not be unzipped.

Secondly there is a severe problem into the availability of documentation of the tools [37]. iPlasma is able to detect 11 code smells [36]. However, the tool is complex in usability and there is no community online nor documentation on how to use the tool. This could be because iPlasma is further developed as Incode, which

[5] http://checkstyle.sourceforge.net/
[6] http://intooitus.com/products/infusion
[7] http://loose.cs.upt.ro/index.php?n=Main.IPlasma
[8] https://users.encs.concordia.ca/ nikolaos/jdeodorant/
[9] https://pmd.github.io/
[10] https://sites.google.com/site/santiagoavidal/projects/jspirit
[11] https://github.com/DeveloperLiberationFront/refactoring-tools/wiki/Stench-Blossom
[12] https://app.assembla.com/spaces/truerefactor/wiki/TrueRefactor
[13] http://www.ptidej.net/research/designsmells/
[14] https://github.com/BrunoLSousa/FindSmells
[15] https://www.sonarqube.org/

is a commercial refactoring tool developed by Intooitus and it no longer exist. Another tool that could not be used because of lack of documentation is TrueRefactor.

Lastly there is Stench Blossom, which identifies code smell real-time and does not provides a list of recommended refactorings. This makes it difficult to refactor an existing software project. Since we will be refactoring existing code we decided to leave Stench Blossom and the above mentioned troubled tools out of our paper. This leaves five working and available tools that are free-to-use for refactoring code smells namely: Checkstyle, JDeodorant, PMD, JSpIRIT and Sonarqube.

Name	Code Smells	Release	Fully covered by Literature	Automated refactoring	Detection technique	Type
JDeodorant	Long Function Large Class Feature Envy Type Checking Duplicate Code*	Version: v5.0.68 05/06/2018	Yes	Yes	ASTParser	Eclipse Stand-alone
Checkstyle	Long Function Large Class Long Parameter List	Version: 8.20 29/4/2019	No	No	Metrics	Stand-alone Third-party plugins
PMD	Long Function Large Class Long Parameter List Feature Envy Data Class Duplicate Code	Version: 6.15.0 26-05-2019	No	No	Metrics	Stand-alone Third-party plugins
JSpIRIT	Large Class Feature Envy Refused Bequest Shotgun Surgery Disperse Coupling Intensive Coupling Data Class Brain Class Brain Method Tradition Breaker	Version: unkown 08/10/2014	Yes	No	Metrics	Eclipse-plugin
Sonarqube	Duplicate Code	Version: 7.7 20-03-2019	No	No	Metrics	Stand-alone Third-party plugins Cloud (git)

Table 3.2: Free-to-use, Online available tools on detecting Code Smells in Java applications.

* Can only show refactor recommendation's on Duplicate Code, it still needs a separate tool for Duplicate Code detection

While Fowler and Beck have described code smells in detail [11] they are not formally defined [35]. This makes it important to know where the tools base their detection on. We encountered some difficulties in finding what code smells are being identified by the tools: Checkstyle, PMD and Sonarqube. It is possible to customise the detection parameters, but this makes it harder to connect them to a code smell in literature. A more thorough investigation shows some references inside the source code of the tools. However, this was not the case for all supported code smells. In contrast to the previously mentioned tools, JDeodorant and JSpIRIT are both developed by research institutes. Both of them have published several papers were they explain how they detect the code smells and on what literature it is based on. This makes them more reliable and transparent for research purposes.

Most tools use metrics to determine if a piece of code is considered a code smell. However, since the code smells are not formally defined it can differ what metrics are used and what their thresholds are. JDeodorant has a different approach. JDeodorant uses the ASTParser from the Eclipse Java Development Tools (JDK) to analyse the properties of statements and calculate possible refactoring possibilities.

JDeodorant and JSpIRIT lack documentation and especially active online communities compared to Checkstyle, PMD and Sonarqube. But JDeodorant and JSpIRIT are relatively easy to use. They provide a well-shown list of the detected code smells with a clickable links to the actual code location. Additionally, JDeodorant supports refactoring suggestions of the detected code smell, which the developer can review before applying

the refactoring. Checkstyle, PMD and Sonarqube have a much more extensive documentation. Futheremore, these tools are used a lot in industry which induces a large online community. This is because these tools have a much brother scope. Besides detecting code smells, they additionally detect patterns such as style flaws, security flaws, possible bugs and performance issues.

JDeodorant and JSpIRIT are both solely developed for the Eclipse IDE, making it harder to implement them when using a different development IDE. PMD and Checkstyle are officially a stand-alone application executed on the command line. There are third-party plugins but the documentation and communities of these applications are lacking. Sonarqube does provides a stand-alone GUI and they offer an online environment integrated with Github, making it easy to add to an existing development cycle.

After considering all the above mentioned characteristics of the available tools we recommend to use JDeodorant as refactoring tool. JDeodorant and JSpIRIT are the best tool supported by literature, but JDeodorant is more thoroughly documented than JSpIRIT. Furthermore, JDeodorant is still being maintained and it offers automated refactoring suggestions which will speed up the refactoring process.

Chapter 4

Experimental Setup

In this chapter we will explain the setup of our experiments. The setup has been created in collaboration with Lukas Koedijk and with help from Kees Verstoep. First we will give information of DAS where the experiments are executed on in Section 4.1. Next we will explain the workflow of the experiments in Section 4.2 followed by a detailed explanation how we obtain the energy readings in Section 4.3. We will show what codebases we have selected in Section 4.4 and how we extracted the application for measurements in Section 4.6. Finally, we will explain how we refactored the codebases in Section 4.5 and how we classify how much refactoring is done in Section4.7.

4.1 DAS

The Distributed ASCI Supercomputer (DAS) [57] is a six-cluster wide-area distributed system designed by the Advanced School for Computing and Imaging (ASCI)[1]. The six clusters are located at:

- VU University, Amsterdam (VU)
- Leiden University (LU)
- University of Amsterdam (UvA)
- Delft University of Technology (TUD)
- The MultimediaN Consortium (UvA-MN)
- Netherlands Institute for Radio Astronomy (ASTRON)

Figure 4.1: The Distributed ASCI Supercomputer located at the VU

In this thesis we will be referring to two systems: DAS-4[2] and the newer system DAS-5[3]. From each of the systems we only used the clusters located at the VU. On the VU cluster there are 6 nodes available for Energy measurements which are connected to an PDU. Each cluster on the DAS has one head node which is used to start measurements on the system.

[1] https://www.asci.tudelft.nl/
[2] https://www.cs.vu.nl/das4/
[3] https://www.cs.vu.nl/das4/

4.1.1 Hardware

ASCI started with DAS-1 in 1995 and is build on two principles. The first principle is that the system should be homogeneous. All of the nodes run on the same operating system CentOS[4] and other system software. While most of the nodes have the same hardware, shown in Table 4.1, there are some exceptions on the nodes available for energy measurements, see Table 4.2. The second principle is that the DAS is focused on research. It is optimised for short interactive experiments. The DAS implemented a SLURM queue [58] to optimise fairness between different measurements.

	DAS-5
Year	2015
Clusters	6
Cores	3252
CPU	Dual Eight-core Xeon E5-2630v3
Interconnect	FDR Infiniband
WAN	Light paths

Table 4.1: Overview of DAS-5 system [32]

Node	GPU
node024	TitanX
node025	TitanX
node026	Titan, TitanX-Pascal
node027	RTX2080Ti
node028	K20, Xeonphi,michost
node029	GTX980, Titanx-Pascal

Table 4.2: The nodes attached to a PDU have special GPU's attached.

The six nodes available for energy measurements are shown in Table 4.2. There are some quite advanced GPUs connected to these nodes. In our measurements we will not be using these GPUs, but there is no option to remove the GPUs from the cluster. For all our measurements we only used node029 to minimise the influence from different hardware. Since we compare the relative difference in energy consumption which is executed on the exact same machine, the background noise and we assume that the additional noise from the GPU will cancel each other out.

Due to historic reasons the PDU is only accessible through the DAS-4 cluster. Since the nodes available on the DAS-4 are running on old machines and are already used a lot, we will be running the jobs on the DAS-5 cluster and use DAS-4 to obtain the Energy readings.

[4] https://www.centos.org/

4.2 Setup

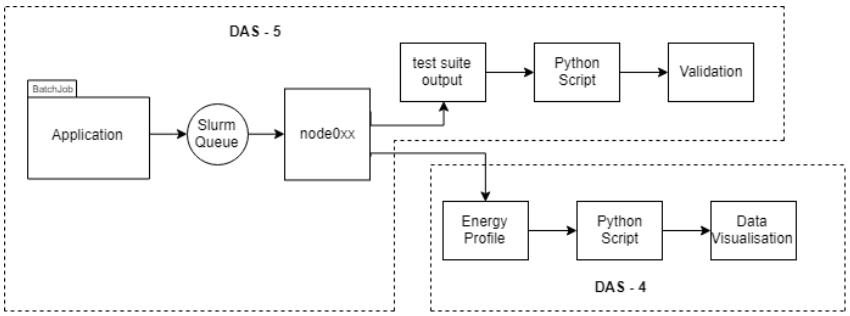

Figure 4.2: Experimentation workflow

Figure 4.2 shows the schematic workflow for the executing of one sample. The first step is to send a BatchJob[5] to the SLURM queue. A BatchJob contains information for needed for resource allocating by the DAS and it contains the run-time instructions. An example of such a BatchJob is shown in Appendix C. When the job has been assigned to the correct node it waits until all the right resources are allocated. From this moment nobody else has access to the node. With our created setup the job will have two outputs. One output on DAS-5, which will contain all the run-time information, and one output on DAS-4, which contains the energy profile. These outputs are later used for data validation and data analysis. In this thesis we executed the applications in an OpenJDK environment using version 'OpenJDK Runtime Environment (build 1.8.0_161-b14)'.

4.2.1 BatchJob

When the resources are allocated we can start with the actual measurements. First we start with a 60 seconds idle measurement. This means that we let the node sleep for 60 seconds, while measuring the energy consumption. The second step is to execute the application while measuring the energy consumption. To obtain enough energy reading we execute the applications for roughly 8 minutes, obtaining around 6000 readings per sample.

4.3 PDU

On the DAS-5 server there are 6 nodes that are individually connected to the PDU - Racktivity[6]. The Racktivity PDU has an error margin of 1%. We can access the PDU through the Simple Network Management Protocol (SNMP), but we can only access the PDU from the DAS-4. This creates a rather complex setup which is shown in Figure 4.3. We have created an automated shell script that starts to continuously read the data from the PDU and writes the output to a file. Additionally, we created a script that will stop the script reading the data from the PDU. Examples of these scripts are shown in Appendix C.

[5] https://slurm.schedmd.com/sbatch.html
[6] http://www.racktivity.com/

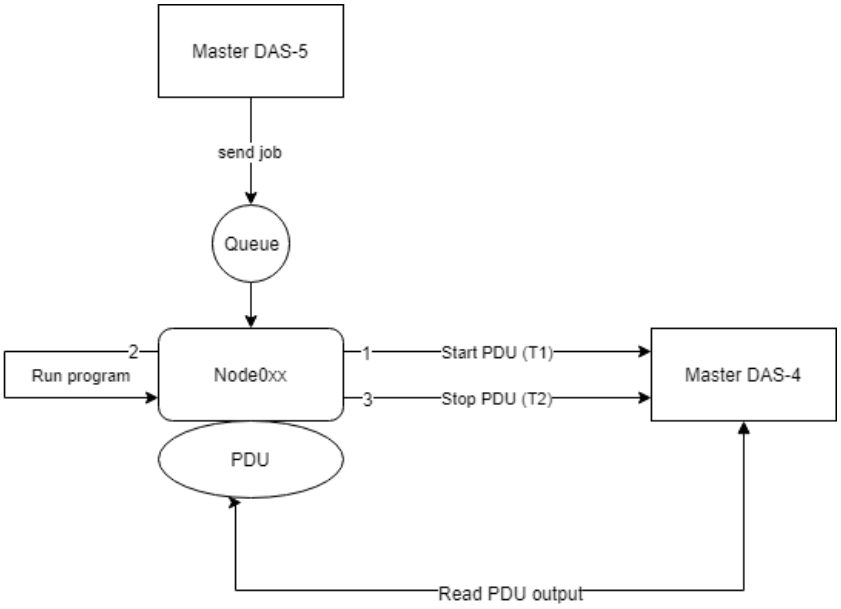

Figure 4.3: DAS setup overview

In order to start the created scripts we send a signal through ssh. The signal introduces two latency errors T_1 and T_2, shown in Figure 4.3. T_1 is the time between successfully starting the energy readings on DAS-4 and starting the execution of the application on DAS-5. T_2 is the time between the application termination on DAS-5 and stopping the energy readings. In order to know the impact of this error, we have measured the latency 1000 times. The results are shown in Table 4.3. This data shows that the delay at start of the measurement T_1 is around 4.019 ms. Stopping the measurement script, however takes significantly more time, around 280.5 ms. This is roughly the time of 1-2 data-points. Since one sample has around 6000 data-points, the delay from the ssh is negligible.

	Mean (ms)	Standard Deviation (ms)
T_1	4.019	0.1635
T_2	280.5	18.78

Table 4.3: Latency between running the measurement and running the actual application. Test with 1 thousand samples.

For making the connection to the PDU we have used snmpwalk[7]. We will obtain two values from the requests: time in milliseconds and power in Watt. Snmpwalk is not a constant connection and the time between obtained values is different. In order to find the difference between samples we made 1.5 million snmpwalk requests and measured the time between readings. This is shown in Table 4.4. Generally, there is a short time between measurements of less than 0.01 seconds. But in rare occasion there is 9 seconds between a reading. The time difference between readings are accounted for by calculating the average power between two readings multiplied by the time between the readings, see Equation 2.4.

	Mean	Standard Deviation	Max	Min
Time between samples (ms)	78	52	9089	67

Table 4.4: Time between samples using snmpwalk. Test with 1 million samples

[7] https://linux.die.net/man/1/snmpwalk

4.4 Software

When searching for software that fit all the set requirements, Section 3.1, we found a paper on regression testing [59]. We selected 3 software repositories that contained at least 90% covered instructions. The covered instructions have been tested by using a tool called EclEmma[8] in Eclipse. The selected software is shown in Table 4.5. All three of the selected software repositories are libraries. We have chosen library implementations because they are commonly imported in other projects, increasing the reach when it the energy consumption would be affected by the refactoring. In selecting the version of the software we made sure that the software used JUnit4[9]. The internal runnable core of JUnit4 makes it easy to run all the available unit test without the need for external packaging.

Software	Version	Classes	NOM	LOC	Covered Instructions
FasterXML/java-classmate[10]	1.5.0 (23-3-2019)	39	378	2789	93.9%
apache/commons-lang[11]	3.8 (19-8-2018)	258	3150	27643	95.8%
apache/commons-configuration[12]	2.5 (27-4-2018)	367	3026	29942	90.2%

Table 4.5: Used Software

FasterXML/java-classmate will be referred to as 'Classmate'. It is a library for introspecting generic type information. It can realiably resolute generic type declaration for classes types, member methods, static method, fields and constructors. Apache/commons-lang will be referred to as 'Lang'. It is a library that provides additional manipulation of core classes. Using helper utilities it provides String manipulations, Object reflections, serialisation and much more. Apache/commons-configuration will be referred to as 'Configuration'. It is a library that allows easy implementations of configurations in codebases. Using properties files as XML or .properties one can change variables, removing the need for hard-coded settings in the codebase.

4.5 Refactoring with JDeodorant

The different versions of the software will be created using a code smell detection tool called JDeodorant. Most code smell detection tools work with external input from the user. Although the customisation of parameters offers more flexible detection, it decreases automation and it requires expertise. JDeodorant does not require any set of parameters, but identifies and suggests all possible and behaviour preserving refactoring opportunities [40].

JDeodorant, original proposed by Tsantalis and Chatzigeorgiou [42], makes use of an adopted form of Program Dependence Graphs (PDG). In order to increase precision and correctness they implement four additional code analysis techniques within the PDG [40]: Alias analysis [60], Polymorphic method call analysis [61, 62], Handling of branching statements directly in the PDG [63, 64] and Handling of try/catch blocks and throw statements directly in the PDG [65]. The control flow graph makes use of Block-Based Slicing. Using the concept of Maruyama [66] JDeodorant is able to create multiple block-based slices which can be used to produce multiple refactoring possibilities.

JDeodorant can detect 'Feature Envy', 'Large Class', 'Long Method' and 'Type Checking' code smells. Besides detecting the code smells, it additionally displays refactoring suggestion for these code smells and for 'Duplicate Code'. However, both 'Feature Envy' and 'Type Checking' code smells were not detected by JDeodorant for the selected three applications. Limiting us to work with the three available Code Smells: 'Long Method', 'Large Class' and 'Duplicate Code'. In the following paragraphs we will explain how JDeodorant detects the code smells.

[8] https://www.eclemma.org/
[9] https://junit.org/junit4/
[10] https://github.com/FasterXML/java-classmate/tree/13bc9c549c47209bfe8f6b5cc21a732c78ed656e
[11] https://github.com/apache/commons-lang/tree/9801e2fb9fcbf7ddd19221e9342608940d778f8c
[12] https://github.com/apache/commons-configuration/tree/dc00a04783ea951280ba0cd8318f53e19acb707f

Long Function When and how long methods should be extracted is based on two algorithms in JDeodorant [40, 42]. The first algorithms detects complete computation slices (a slice that makes a computation of a local variable) and the second algorithm detects object state slices (a slice that affects the state of an object). Besides the algorithms JDeodorant has rules to account for edge cases. These rules are: Behaviour Preservation, Indispensable Statements, Duplicate Code, Anti-dependencies, Output-dependencies and some Extreme-cases [40].

Large Class Large Class refactoring algorithm is applied to every class regardless of the cohesion. This way it will not require thresholds. The methodology of detecting Large Classes is based on two aspects [67–69]: the distance between class members based on the Jaccard distance and the cohesive groups of entities that can be extracted. Additional to these algorithms they provide a set of rules to limit the refactored code to have at least a certain functionality (at least one entity and at least one method) and that it will preserve the same observable behaviour as refactoring requires.

Duplicate Code As mentioned in Table 3.2, JDeodorant requires an additional tool for the detection of 'Duplicate Code'. The tools supported by JDeodorant are: CCFinder, ConQat, Deckard, NiCad and ClondeDR. In this thesis we used NiCad[13] developed by Chanchal Roy [70] as our duplicate detection tool. Nicad can detect type 1, type 2 and type 3 clones [71]. NiCad is text-based, but uses the benefits from tree-based structures, pretty-printing, code normalisation, source transformation, code filtering and it is able to detect Near-miss clones. JDeodorant continues with the output and provides three features [72–76]. First, it can group the detected clones. On import the clones are checked if they are valid and are categorised in groups of sub-clones. Second it can visualise the duplicate code, highlighting differences of the same type, unmapped statements (Clone Gaps) and semantically equivalent statements. And finally, using extract method, extract and pull up method and introduce utility method it can provide a refactoring suggestion.

4.6 Workload

In order to create a workload on the application, we have two options: we can either create a simulated Load test designed for the application or we can execute all the JUnit tests. JUnit test checks the functionality of a given system. A load test, usually occurs after functionality has been tested, checks how a system behaves under a given load.

If we were to simulate a load test on the applications we would obtain a real-life example of the energy consumption for that specific software under that specific load. This means that we investigate the impact of refactoring code smells on the energy consumption only for that specific application and scenario. Additionally, it could be that the refactored code is not even executed.

However, by executing all JUnit tests in combination with the tool EclEmma, we will know exactly what code is executed. This way we can insure that the refactored code is actually executed and obtain the impact of refactoring code smells on the energy consumption. For this reason we will be executing all the JUnit tests and not using a simulated load test. In order to be able to execute all the JUnit tests we created a 'driver' class that consist of a main function that executes all available set of tests, similar as done before by Dhaka et al. [16]. An example of a driver file is shown in Appendix B. We refactored all the three software packages using JDeodorant and we exported the software as runnable JAR from Eclipse so it can be executed on the DAS.

However, for Configuration there were some problems with exporting as runnable JAR and the driver class had to be called manually from the command line. Since this is a codebase specific issue and it has no impact on the results the process of exporting, Configuration is explained in more details in Appendix B.

[13] https://www.txl.ca/txl-nicaddownload.html

4.7 Classifying the refactorings

In order to be able to provide some insight into how much we changed the code by refactoring the software, we will explain corresponding code metrics should be changed after refactoring.

The Software Improvement Group (SIG) conducts research in classifying maintainable software [77]. Using their framework, referred to as BCH[14], we are able to show how much the refactoring altered the software. Their framework shows the percentage of methods of a certain McCabe index [78] and the percentage of methods of a certain amount of lines. Refactoring long methods should decrease these values. Additionally, for Duplicate Code they provide how much percentage of duplicate code the project contains, which should decrease aswell.

However for the Large Class refactoring we had to resort to other measures. A commonly used metric for determining the complexity of a class is Weighted Method Complexity (WMC). How WMC is calculated can be different, but we are discussing the sum of the McCabe's complexity as discussed by Lanza et el [79]. Additionally, Lanza et al. computed thresholds for what a high WMC value is, these values are shown in Table 4.6. Refactoring a large class should decrease the amount of methods with a high WMC value.

	low	average	high	very high
WMC	5	14	31	47

Table 4.6: Derived thresholds for the WMC metrics for Java languages, by Lanza et al. [79]

Additional metrics that are used are Number of Methods (NOM), Number of Classes (NOC) and Lines of Code (LOC), where LOC are the non-commented lines of Java code. In order to calculate the metrics we used a plugin from IntelliJ IDEA[15] named MetricsReloaded[16]. In order to calculate the LOC we calculated the non-commented Lines of code.

[14] https://bettercodehub.com/
[15] https://www.jetbrains.com/idea/
[16] https://plugins.jetbrains.com/plugin/93-metricsreloaded

Chapter 5

Results

In this chapter, we present the findings of our experiments. First we show how refactoring the code smells impacted the energy consumption. First we show a table with the obtained mean energy consumption, followed by boxplots to show if there is any variance in the data. Furthermore, we show how the refactoring altered the corresponding software metrics. The data is analysed using python. The code and the raw data are available on Github[1].

5.1 Energy Consumption

All the jobs were run on Node029 on DAS-5 VU cluster. For each of the software package we show boxplots of the energy consumption of the executed applications. How boxplots can be interpreted is explained in more detail in Appendix D.

The energy consumption is calculated by Equation 2.4. For each obtained power reading, the idle power is subtracted. The idle power is calculated by taking the mean power of all the idle measurements of the executed software package. We thus have the following equation

$$\mu = \frac{\sum \mu_i \cdot n_i}{N}, \tag{5.1}$$

where μ is the subtracted idle power, μ_i is the mean idle power of measurement i with n_i measurement-points and N is the total amount of measurement-points.

In the boxplots we also plotted the idle energy consumption 'IDLE' to show if there are any variations in the idle energy consumption at the different starting times. The energy consumption of 'IDLE' should be zero since we have subtracted the idle power. Each boxplot contains 240 samples of 1 minute of measuring the idle energy consumption. Next, we show the Original version (Or), 'Long Function' refactored version (LF), 'Large Class' refactored version (LC) and 'Duplicate Code' refactored version (DUP). Each refactored version contains 60 samples.

Software		Mean energy consumption (Joule)	difference
Classmate	Original	$1.20 * 10^4$	-
	Long Function	$1.21 * 10^2$	+1.06%
	Large Class	$1.20 * 10^3$	+0.30%
	Duplicate Code	$1.20 * 10^4$	-0.26%
Lang	Original	$1.08 * 10^4$	-
	Long Function	$1.10 * 10^4$	+1.52%
	Large Class	$1.10 * 10^4$	+1.35%
	Duplicate Code	$1.11 * 10^4$	+2.12%
Configuration	Original	$2.80 * 10^3$	-
	Long Function	$2.89 * 10^3$	+3.29%
	Large Class	$2.83 * 10^3$	+1.07%
	Duplicate Code	$2.74 * 10^3$	-1.89%

Table 5.1: Mean energy consumption for the different software versions and the relative difference compared to their original version.

Table 5.1 shows the mean energy consumption per executed version for each of the codebases. Additionally it shows the relative difference between refactored versions and the original.

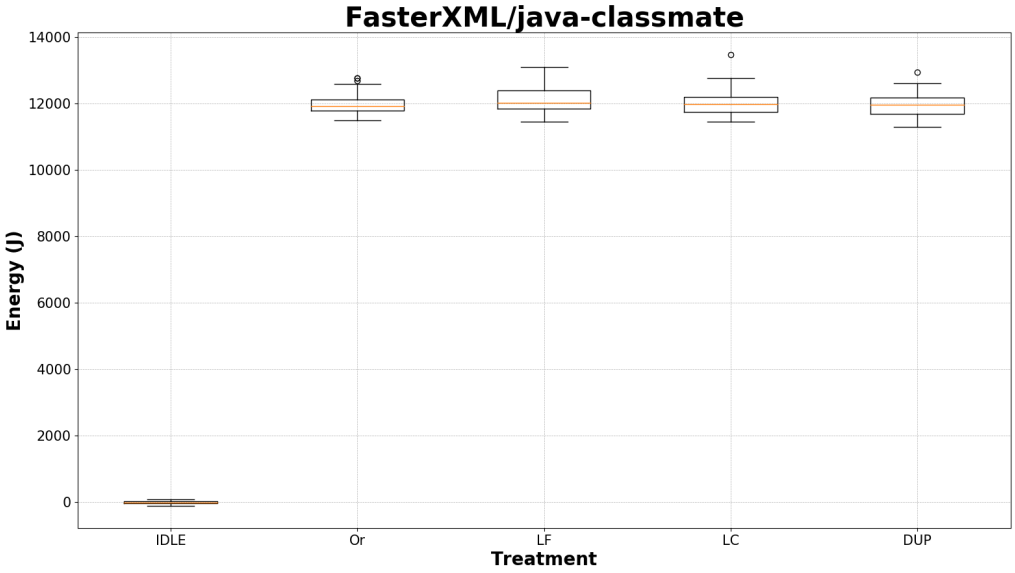

Figure 5.1: Impact on the energy consumption in Joule when refactoring code smells in the Classmate codebase.

In Figure 5.1 we have plotted the energy consumption when executing the Classmate application. Classmate was the smallest package and contained 226 unit tests. The JUnit testing framework was repeated 150,000 times to obtain sufficient samples. After validating the execution logs, we have found no errors and each unit test has finished successfully. The mean idle energy consumption found is $1.79{\cdot}10^{-2}$ Joule and the mean energy consumption for the application are shown in Table 5.1. We observed an increase in energy consumption for 'Large Class' with 1.06% and 'Long Method' with 0.30%, but we observe a decrease in energy consumption when refactoring for 'Duplicate Code' with 0.26%.

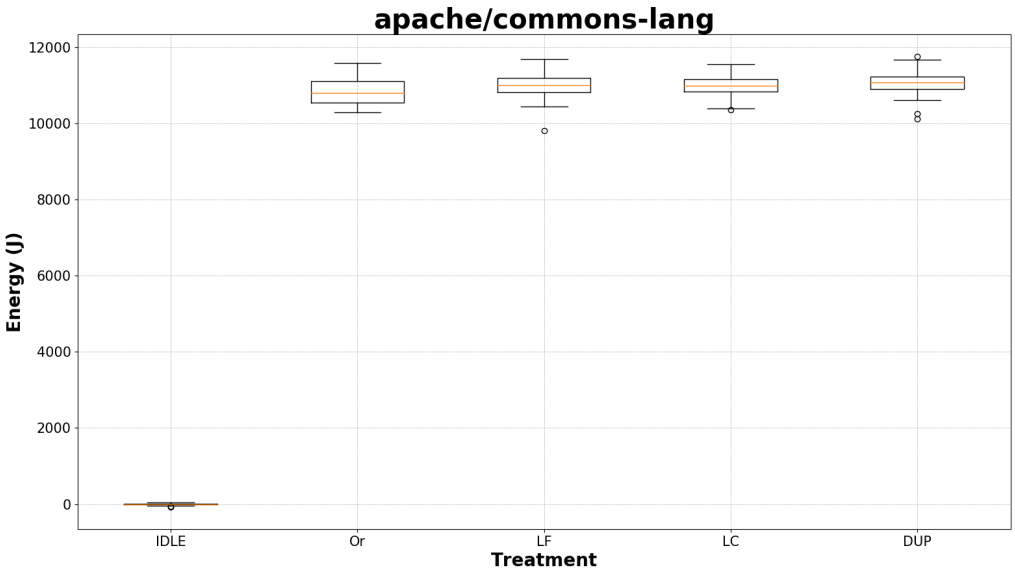

Figure 5.2: Impact on the energy consumption in Joule when refactoring code smells in the Lang codebase.

In Figure 5.2 we have plotted the energy consumption when executing the Lang application. The Lang application contained 4114 unit tests and the JUnit testing framework was repeated 40 times for each sample. After validating we found that a few of our measurement had errors. After investigating we found that there was one unit test that would fail 1 in a thousand runs. These measurements have been excluded from the plot and have been re-run without errors. The data of the excluded measurements are available on Github[1]. The process is explained in more details in the Appendix B.2. The mean idle energy consumption found is $-4.79\dot{1}0^{-1}$ Joule and the mean energy consumption for the application are shown in Table 5.1. We observe an increase in energy consumption for all three of the refactorings. After refactoring the energy consumption increased with 1.52% for 'Long Function', 1.35% for 'Large Class' and 2.12% for 'Duplicate Code'.

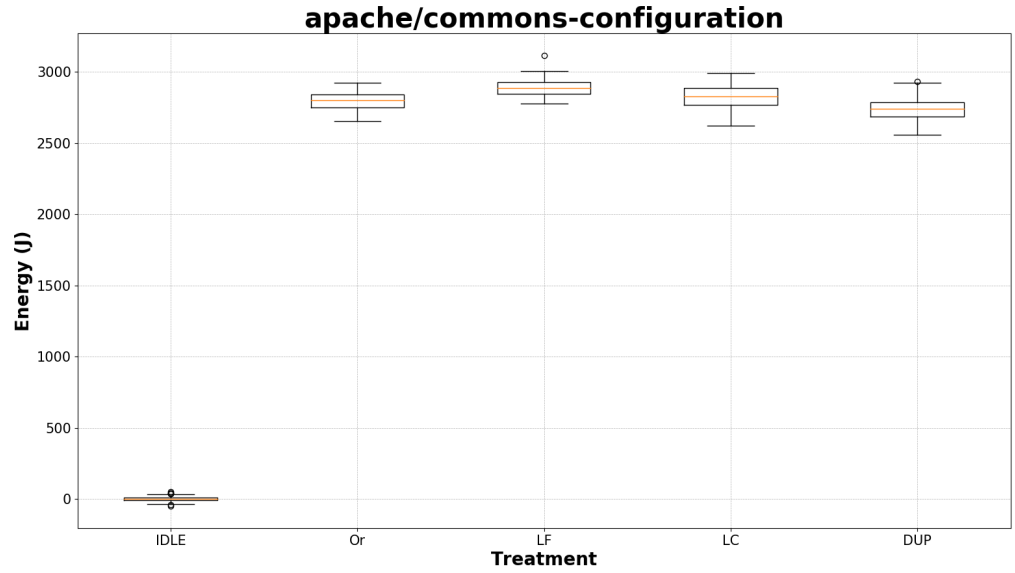

Figure 5.3: Impact on the energy consumption in Joule when refactoring code smells in the Configuration codebase.

In Figure 5.1 we have plotted the energy consumption when executing the Configuration application. The Configuration application contained 2806 unit tests and the JUnit testing framework was repeated 25 times for each sample. After validating the execution logs we have found no errors and each unit test has finished successfully. The mean idle energy consumption found is $-2.68 \cdot 10^{-1}$ Joule and the mean energy consumption for the application are shown in Table 5.1. After refactoring we observe an increase in energy consumption with 3.29% for 'Long Function' and 1.07% for 'Large Class'. We observe a decrease in energy consumption when refactoring for 'Duplicate Code' of 1.89%.

5.2 Impact of Refactoring

In the following section we show how the refactoring affected corresponding software metrics.

5.2.1 Long Function

Software		LOC	NOM	BCH[14] - McCabe index				BCH[14] - Lines of Code			
				< 5	5-10	10-25	> 25	< 15	15-30	30-60	> 60
Classmate	ori	2789	378	72.03 %	17.7 %	5.54 %	4.73 %	69.29 %	15.14 %	10.85 %	4.73 %
	ref	2895	410	80.34 %	15.62 %	0 %	4.04 %	78.92 %	11.81 %	5.24 %	4.04 %
Lang	ori	27646	3199	64.74 %	21.75 %	9.81 %	3.7 %	59.32 %	26.21 %	8.34 %	6.12 %
	ref	27976	3287	67.13 %	20.37 %	9.36 %	3.14 %	61.41 %	25.18 %	8.11 %	5.3 %
Configuration	ori	29962	3083	81.07 %	11.93 %	4.82 %	2.18 %	58.57 %	24.34 %	12.42 %	4.67 %
	ref	30118	3157	83.32 %	10.87 %	3.64 %	2.16 %	61.72 %	23.03 %	10.61 %	4.65 %

Table 5.2: Change of metric when refactored for 'Long Function'

In Table 5.2 we have shown the change of the metrics values after refactoring for Long Methods. We can find an overall decrease in McCabe complexity per Unit and an overall decrease in Unit size. As is stated in Table 5.2, there are still some long function in each of the languages. These long methods were not refactored because either JDeodorant did not detect them or changing them would alter the behaviour of the software.

For Classmate we observed an increase of 32 methods, for Lang we find an increase of 88 methods and for Configuration we find an increase of 74 methods. In Classmate we have shown the largest increase of short functions, namely by 9.7%. As expected we find an increase in Lines of code in all three of the codebases after refactoring for Long Methods.

5.2.2 Large Class

Software		LOC	NOC	WMC				
				very low <5	low 5-15	average 15-31	high 31-47	very high >47
Classmate	ori	2789	39	23.68 %	36.84 %	26.33 %	5.26 %	7.89 %
	ref	2943	47	26.09 %	36.96 %	26.09 %	4.35 %	6.51 %
Lang	ori	27646	258	38.54 %	24.39 %	14.64 %	7.8 %	14.63 %
	ref	29939	279	38.94 %	24.78 %	14.16 %	7.96 %	14.16 %
Configuration	ori	29962	367	33.47 %	29.08 %	19.52 %	7.57 %	10.36 %
	ref	30596	401	32.28 %	34.74 %	17.19 %	7.02 %	8.77 %

Table 5.3: Change of metrics when refactored for 'Large Class'.

In Table 5.3 we have shown the change of the metrics values of refactoring for Large Class Code Smells. As expected we observed an increase of number of classes, as-well as an increase in lines of code. Additionally, we observed that the WMC convert to a lower complexity per class. All the refactorings have been extract class refactorings. We have extracted 8 classes for Classmate, 21 classes for Lang and 34 classes for Configuration.

5.2.3 Duplicate Code

Software		LOC	NOM	BCH[14] - Duplicate Code
Classmate	ori	2789	378	3.79 %
	ref	2766	392	3.4%
Lang	ori	27646	3150	9.08 %
	ref	26107	3180	3.73 %
Configuration	ori	29962	3026	1.59 %
	ref	29898	3034	1.24 %

Table 5.4: Change of metrics after refactoring for 'Duplicate Code'.

In Table 5.4 we show the change in the codebases after refactoring for 'Duplicate Code'. We observed an overall decrease in duplicate code: Classmate decreased with 0.39%, Lang decreased with 5.35% and Configuration decreased with 0.35%. Additionally we observed a decrease in lines of code: Classmate decreases with 0.83%, Lang with 5.9% and Configuration with 0.21%.

Chapter 6

Discussion

In this chapter, we discuss the results of our experiments shown in Chapter 5. First we will discuss the statistically significance of the results. After which we discuss how the refactoring impacted the energy consumption for each of the refactored code smell. Furthermore, we build up to an answer for each of the posed research questions. The results from the experimentations are plotted in Figure 5.1, 5.2 and 5.3.

6.1 Statistically significance of the results

Since energy distributions typically do not follow a normal distribution, we used the Mann Whitney U test to test the significance of the data [80]. The Significance Level has been set to $\alpha = 0.05$. In order to know if there was an increase or decrease in energy consumption we created two sets of one-sided Mann Whitney U tests:

One-sided decrease in energy consumption:
H_0 Refactoring had no effect ($\mu_0 = \mu_1$)
H_1 Refactoring decreased energy consumption ($\mu_0 > \mu_1$)
One-sided increase in energy consumption:
H_0 Refactoring had no effect ($\mu_0 = \mu_1$)
H_1 Refactoring increased energy consumption ($\mu_0 < \mu_1$)

where μ_0 is the mean energy consumption of the original version and μ_1 is the mean energy consumption of the refactored version.

Software		P-value (decrease)	P-value (increase)	Sample size	U-val
Classmate	Long Function	0.954	0.0462	60	2121
	Large Class	0.664	0.338	60	1880
	Duplicate Code	0.317	0.684	60	1709
Lang	Long Function	1.00	0.00142	60	2369
	Large Class	1.00	0.00142	60	2327
	Duplicate Code	1.00	0.0000531	60	2539
Configuration	Long Function	1.00	$4.00 * 10^{-11}$	60	3039
	Large Class	0.989	0.0116	60	2233
	Duplicate Code	0.0000566	1.00	60	1064

Table 6.1: Statistically significance when comparing the energy consumption of the original codebase with the refactored version. The P-values are obtained by the Mann Whitney U Test.

In Table 6.1 we show the obtained P-values by using the Mann Whitney U test. We have compared the energy consumption of the original version with the energy consumption of the refactored version. The significance level (α) is set to 0.05.

In the Classmate codebase when refactored for 'Long Function' we have obtained a P-value of 0.0462 when doing a one-sided test to see if there is an increase in energy consumption. The obtained p-value ($0.0462 < 0.05$), makes it able to reject the null hypothesis of the one-sided increase significance test. Leading to the conclusion that there is a statistically significant increase in the energy consumption. However, we fail to reject both of the null hypothesis for 'Duplicate Code' and 'Large Class' refactoring, making us unable to say anything about the statistically significance of these two sets.

In the Lang codebase we are able to reject all three of the null hypothesises for one-sided increase in energy consumption. Showing a statistically significant increase in energy consumption for 'Long Function', 'Large Class' and 'Duplicate Code' refactorings.

In the Configuration codebase we are able to reject the null hypothesis for the one-sided decrease statistical test when refactored for 'Duplicate Code'. Obtaining our first and only measurement were we have shown a statistically significant increase in the energy consumption after refactoring. We are able to reject the null hypothesis for the one-sided increase in energy consumption, when refactored for 'Long Function' and 'Large Class', showing a statistically significant increase in the energy consumption.

6.2 Impact on the energy consumption

Refactoring \ Software	Classmate	Lang	Configuration
Long Function	+1.06%	+1.52%	+3.29%
Large Class	-	+1.35%	+1.07%
Duplicate Code	-	+2.12%	-1.89%

Table 6.2: Overview of relative change in energy consumption, which had a statistically significant impact.

Table 6.2 shows an overview of all the statistically significant changes in the energy consumption after refactoring for the code smells. For two of the nine measurements we were unable to reject any of the one-sided tests and they thus yielded no statistically significant result. With six of the measurements we obtained an increase of the energy consumption, where 'Long Function' refactored showed the largest increase in energy consumption of 3.29%. Additionally, we found one significant decrease in the energy consumption for refactoring 'Duplicate Code', which had an impact of 1.89%. In the following subsection we will discuss the how the three research questions have been answered in this work.

6.2.1 Long Function

To answer RQ 1, our initial hypothesis has shown that refactoring long methods would add a function on the stack and thus create an overhead. However, compilers could negate the effect by inline optimisations (Section 3.2). Our empirical analysis has shown that refactoring long methods leads to a statistically significant increase in energy consumption ranging from 1.06% to 3.29%, Table 6.2. This means that the overhead from the additional function calls led to a significant increase in the energy consumption and that the compiler was unable to negate the effect by inlining. This leads us to our first important Finding 1.

Finding 1: Refactoring for 'Long Function' shows an increases in the energy consumption

To validate our refactorings we have shown how many complex and long methods are still in the codebase after refactoring in Table 5.2. We see an overall decrease in complexity per method and an overall decrease

in amount of lines per method. However there are still some 'long' methods in the codebase. These long methods were either not detected by JDeodorant or changing them would alter the observable behaviour. We would expect that refactoring them would have led to an even larger increase in energy consumption.

6.2.2 Large Class

To answer RQ 2, our initial hypothesis has shown that refactoring large classes will increase the energy consumption induced by the additional required memory and the overhead in object creation (Section 3.2). This is validated by our empirical analyses, which has shown two statistically significant increases in the energy consumption when refactoring large classes, this is shown in Table 6.2. We observed an increase in energy consumption of 1.07% for the Configuration application and 1.35% for the Lang application. For the classmate application we were unable to see a statistically significant difference in energy consumption, the calculated p-values are shown in Table 6.1. This does not mean that refactoring had no impact, we just do not know if the increase in energy consumption of 0.30% was by random chance. The overall increase in energy consumption leads us to our second important Finding 2.

Finding 2: Refactoring for 'Large Class' shows an increase in the energy consumption

To validate our refactoring we have shown the change in weighted method complexity per class in Table 5.3. We see a small overall decrease in WMC. The reason that we only see a small decrease can be explained by the fact that all public methods have to stay accessible through the initial class. The logic and private methods are transferred to the new class, but there is still a quite a high WMC value from the large amount of methods in the class.

6.2.3 Duplicate Code

To answer RQ 3, we have showed our initial assumption on the effect of refactoring duplicate code in Section 3.2. It mentions that refactoring duplicate code could increase energy consumption caused by the additional function call or that it could decrease energy consumption since less code has to be compiled and since it is called more often, the code is more likely to be optimised. Our empirical analysis has shown a spread in energy consumption. We have found an statistically significant increase in energy consumption of 2.12% and we have found a statistically significant decrease in energy consumption of 1.89%. For the measurement from Classmate we could not reject the null hypothesis, thus we do not know if the obtained decrease in energy consumption of 0.26% was by random chance. This leads is to our third Finding 3.

Finding 3: Refactoring for 'Duplicate Code' shows an increase and a decrease in energy consumption.

In Table 5.4 we have shown that the detected duplicate has decreased after refactoring. Additionally, as expected, we have seen that the total amount of lines of Java in the codebase has decreased. We have assumed that less lines of code would reduce the energy consumption however, Lang had the largest decrease in lines of code and yet yielded the highest increase in energy consumption.

6.3 Threats to Validity

It is not always possible to remove all the code smells. This is because we do not remove them but refactor them. With refactoring you may not change the observable behaviour of the code (Chapter 2.2) making it unable to refactor a method if it changes the observable behaviour. Additionally, we have only refactored the code smells that where covered by the unit tests. Changing code smells that were not covered will not have affected the obtained results from the empirical investigation. Since we have chosen codebases with high code coverage (+90%) the amount of code that is not covered is limited.

For the measurements it is important that interference is minimised. While the interference is limited as much as possible by using the DAS-system, there will still be background noise. However, since we are dealing with a relative difference in energy consumption the interference is not a problem as long as it is constant. This way when comparing the different run times, the background noise is subtracted from each other. From Figure 5.1, Figure 5.2 and Figure 5.3 it becomes clear that the idle energy consumption is constant with little to no change. This means that the background noise does not have any significant impact on our results.

Refactoring was done by use of a refactoring tool call JDeodorant. While code smells are a well known concept, there is not a formal definition of code smells [11], making it theoretically possible for different tools to identify different code smells. While this can lead to different refactorings, the concept of the refactorings will be the same.

The experiments are conducted in a controlled environment on the DAS. We have measured the energy consumption of the whole system and shown the relative change in energy consumption. Different hardware can lead to a change in the energy consumption as the costs of bytecode type, native method and monitor operation can be different [81]. For example the Intel Turbo Boost has shown to increase performance, but also to increase energy consumption [82].

Chapter 7

Related work

A key aspect of refactoring is that the observable behaviour of the code is preserved. With this in mind we have divided the related work in different sections. All of the mentioned articles research the impact of the energy consumption by changing the code. There has already been some related work on code smell refactorings, but there is still a gap to be filled. While other research has focused on approximating the energy consumption, we have measured on a system-level. Furthermore, research has shown that the C++ compiler has a different energy profile than the Java JVM [24–27], requiring empirical research to be done on both languages. Additionally, we have limited the influence from background noise by using the DAS, removing any complications from virtualisation.

7.1 Refactoring Code Smells

Verdecchia et al. [15] measured the energy consumption of 6 code smells on ORM-based Java applications. But with their setup they are unable to know what code is executed, making it possible that the executed code does not contain the refactored code smells. Furthermore, they run their experiments on a single machine that is running multiple virtual machines. They fail to find any significant difference except for refactoring Long Methods. They found an energy reduction from up-to 49.9% when refactoring for Long Methods.

Additionally, Dhaka & Singh [16] investigated the impact of the refactoring sequences of code smells on the energy consumption for three open-source object-oriented software systems. They investigated the influence of the permutations of 'Long Function', 'Large Class' and 'Feature Envy'. They approximated the energy consumption using a tool called Jalen. Their result shows a wide variance in the energy consumption for each of the permutation of code smell refactorings. Their research implies that refactoring one code smell can introduce another one.

Serge Demeyer [22] investigated the influence on the performance by replacing switch statements and conditional logic with polymorphism in the C++ language. He found that polymorphism is slower in comparison with small case- or if-statements, but upon increasement of size polymorphism will be faster. How large they need to be depends on the compiler.

7.2 Refactoring Methods

Additionally, there has been research into the impact of refactoring methods on the energy consumption. Sahin et al. [29] investigated the influence of 6 common refactoring methods on 9 different Java applications. To obtain the energy profiles they measure the energy consumption of the running JUnit tests. They found a change in energy consumption from -7,50% to 4,54%. The places to apply a refactoring method were selected manually. Both within the application and across the different applications they could not find any consistency of an increase or decrease in the energy consumption after applying the refactor method.

Park et al. [23] investigated the influence of 63 refactoring methods proposed by Fowler [10]. These refactoring methods were investigated on the using a power estimation tool XEEMU that works with C and C++. Measuring on a small embedded device they found only 33 of the refactoring methods to be energy efficient.

7.3 Refactoring bad Code Design

Morales et al. [83] focus on patterns such as binding resources to early and using an ArrayMap rather than an expensive HashMap on Android mobile phones. In their article the authors propose a tool 'EARMO' that can remove up to 84% of these patterns and lead to an increase of battery life up to 29 minutes.

Vetrò et al. [21] identified code patterns that use hardware resources in a sub-optimal way to decrease energy consumption. They focus on embedded devices and investigated single methods using the C++ language. Inspired on the original work of Folwer [10] they introduce 'energy smells' as '*A Energy Smell is an implementation choice that makes the software execution less energy efficient.*'. They manually created two functions, one induced with the energy smell and one without it. They measured energy consumption in the order of μW. They found an energy difference of less than 1% and a difference in performance of less than 0,01%.

7.4 Removing bad Code Design

There has been quite some work in decreasing energy consumption in Android mobile-phones by removing power consuming processes such as screen brightness, screen colour and GPS. Gottschalk et al. [18] calls these power consuming processes 'energy code smells' and shows how to detect and remove them. The authors followed up with an empirical study [43] that showed a decrease in energy consumption of up to 27%.

Noureddine et al. [84] and Hao et al. [85] call the energy consuming processes 'Energy Hotspots'. They show that the execution time is not directly correlated with energy consumption. Furthermore, they each provide a tool to model the energy consumption on Android phones and additionally they show an energy map of energy hotspots in Android applications.

Pinto et al. [19] have shown that developers are aware of these energy hotspots. However, they also discuss that the developers answer to solve them are often flawed. As result of Pinto's reseach, Banerjee er al. [20] provide an automated test generation framework that detects energy hotspots in Android applications. Banerjee tested his framework using physical measurements and created an energy over utilisation metric that detects energy hotspots in the application. The frameworks shows developers where a lot of energy is consumed, but now how to solve them.

Chapter 8

Conclusion

Researchers identified code patterns that should be refactored to improve human code readability and make it easier to implement a new feature. These code patterns are called 'code smells'. As software engineers it also our responsibility to minimise the energy consumption of software or at least to make sure that the guidelines we as developers have to follow do not increase the energy consumption. Hence, we posed the question RQ: what is the impact of refactoring code smells on the energy consumption? Some research has been conducted on the C++ language, but research has shown that the energy profile of JVM is very different from C++, requiring further investigation for the impact of refactoring code smells and refactoring methods in Java.

For as far as we know we are the first to introduce assumptions on how the energy consumption is affected by refactoring a code smell for each of the 24 code smells defined by Fowler. We have followed up with an empirical study on three of the code smells: 'Long Function', 'Large Class' and 'Duplicate Code'. Our setup consist of three open-source Java applications which are refactored using JDeodorant. The applications are executed by running all the provided unit tests in the codebase, at the same time we measured their energy consumption on a system level using a PDU on the DAS.

We have found a statistically significant difference in energy consumption in 7 out of the 9 measurements, shown in Chapter 6. We have seen an increase in energy consumption after refactoring for 'Long Function' and 'Large Class'. This is in agreement of our proposed assumptions in Chapter 3. Additionally, when refactoring for 'Duplicate Code' we have both seen an increase and decrease in energy consumption, which is also in agreement with our assumptions. The results from all the measurements are presented in Chapter 5.

The conclusion from this work is not to discourage developers from refactoring code smells. Writing quality code is important and should be a priority. Furthermore, it is not ment as guideline to create small energy optimisations in the code. As said by many developers: '*Premature optimisations are the root of all evil*'. What this work does show is that the Java compiler must increase its performance in recognising these code patterns developers have to follow and translating them to energy efficient machine-code.

8.1 Future work

We have shown that refactoring code smells to increase the energy consumption when refactoring three code smells in Java 8. Future work can be done too see if the same results are obtained using the newer version Java 11. When this would be the case, it is an important reason for developers to stop using Java 8 and switch to Java 11 and otherwise it pinpoints were the Java compiler should be better optimised with respect to energy efficiency. Additional, more empirical studies can be done by refactoring the other 21 code smells to validate our proposed assumption.

We presented an empirical study on 3 of the 22 code smells that are assumed to have an impact on the energy consumption. Future work should be done to investigate the impact of refactoring the other 19 code smells, focusing on the two other code smells commonly known by developers as 'Lazy Class' and 'Feature Envy'.

Acknowledgements

This thesis has been written for the Master Software Engineering at the University of Amsterdam. It has been written in collaboration with KPMG, in specific within the Digital Enablement team. We would like take this possibility to thank all support and input received from the Digital Enablement team and my fellow students from the Master Software Engineering. By name we would like thank dr. Ana Oprescu and Rali Genova for guiding us throughout the whole process, Lukas Koedijk for helping in setting up the DAS measurements, drs. Kees Verstoep for providing access to the DAS and with help setting it up and Joran van Weel for proofreading the work.

Bibliography

[1] W. Van Heddeghem, S. Lambert, B. Lannoo, D. Colle, M. Pickavet, and P. Demeester, "Trends in worldwide ict electricity consumption from 2007 to 2012", *Computer Communications*, vol. 50, pp. 64–76, 2014.

[2] J. G. Koomey, "Worldwide electricity used in data centers", *Environmental research letters*, vol. 3, no. 3, p. 034 008, 2008.

[3] T. Mastelic, A. Oleksiak, H. Claussen, I. Brandic, J.-M. Pierson, and A. V. Vasilakos, "Cloud computing: Survey on energy efficiency", *Acm computing surveys*, vol. 47, no. 2, p. 33, 2015.

[4] L. Belkhir and A. Elmeligi, "Assessing ict global emissions footprint: Trends to 2040 & recommendations", *Journal of Cleaner Production*, vol. 177, pp. 448–463, 2018.

[5] M. Avgerinou, P. Bertoldi, and L. Castellazzi, "Trends in data centre energy consumption under the european code of conduct for data centre energy efficiency", *Energies*, vol. 10, no. 10, p. 1470, 2017.

[6] A. Beloglazov, R. Buyya, Y. C. Lee, and A. Zomaya, "A taxonomy and survey of energy-efficient data centers and cloud computing systems", in *Advances in computers*, vol. 82, Elsevier, 2011, pp. 47–111.

[7] S. Murugesan, "Harnessing green it: Principles and practices", *IT professional*, vol. 10, no. 1, 2008.

[8] L. Benini, A. Macii, and M. Poncino, "Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques", *ACM Transactions on Embedded Computing Systems*, vol. 2, no. 1, pp. 5–32, 2003.

[9] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving gpu energy efficiency", *ACM Comput. Surv.*, vol. 47, no. 2, Aug. 2014.

[10] M. Fowler, "Refactoring: Improving the design of existing code", in *11th European Conference. Jyväskylä, Finland*, 1997.

[11] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[12] 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.

[13] 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 *2009 3rd international symposium on empirical software engineering and measurement*, IEEE, 2009, pp. 390–400.

[14] 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*, IEEE Press, 2012, pp. 78–87.

[15] R. Verdecchia, R. A. Saez, G. Procaccianti, and P. Lago, "Empirical evaluation of the energy impact of refactoring code smells", in *ICT4S2018. 5th International Conference on Information and Communication Technology for Sustainability*, B. Penzenstadler, S. Easterbrook, C. Venters, and S. I. Ahmed, Eds., ser. EPiC Series in Computing, vol. 52, EasyChair, 2018, pp. 365–383.

[16] G. Dhaka and P. Singh, "An empirical investigation into code smell elimination sequences for energy efficient software", *2016 23rd Asia-Pacific Software Engineering Conference*, pp. 349–352, 2016.

[17] R. Pérez-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.

[18] M. Gottschalk, M. Josefiok, J. Jelschen, and A. Winter, "Removing energy code smells with reengineering services.", *GI-Jahrestagung*, vol. 208, pp. 441–455, 2012.

[19] G. Pinto, F. Soares-Neto, and F. Castor, "Refactoring for energy efficiency: A reflection on the state of the art", in *Proceedings of the Fourth International Workshop on Green and Sustainable Software*, IEEE Press, 2015, pp. 29–35.

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

[21] A. Vetro, L. Ardito, and M. Morisio, "Definition, implementation and validation of energy code smells: An exploratory study on an embedded system", 2013.

[22] S. Demeyer, "Maintainability versus performance: What's the effect of introducing polymorphism", *Edegem, Belgium: Universiteit Antwerpe*, 2002.

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

[24] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere, "Method-level phase behavior in java workloads", *ACM SIGPLAN Notices*, vol. 39, no. 10, pp. 270–287, 2004.

[25] D. C. Lee, P. J. Crowley, J.-L. Baer, T. E. Anderson, and B. N. Bershad, "Execution characteristics of desktop applications on windows nt", in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*, IEEE, 1998, pp. 27–38.

[26] S. Hu *et al.*, "Cache performance in java virtual machines: A study of constituent phases", in *2002 IEEE International Workshop on Workload Characterization*, IEEE, 2002, pp. 81–90.

[27] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh, "Characterizing the memory behavior of java workloads: A structured view and opportunities for optimizations", in *ACM SIGMETRICS Performance Evaluation Review*, ACM, vol. 29, 2001, pp. 194–205.

[28] T. Group, *Tiobe index for ranking the popularity of programming languages*, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2019.

[29] 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*, ACM, 2014, p. 36.

[30] E. A. Jagroep, J. M. van der Werf, S. Brinkkemper, G. Procaccianti, P. Lago, L. Blom, and R. van Vliet, "Software energy profiling: Comparing releases of a software product", in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16, Austin, Texas: ACM, 2016, pp. 523–532.

[31] R. F. da Silva, A.-C. Orgerie, H. Casanova, R. Tanaka, E. Deelman, and F. Suter, "Accurately simulating energy consumption of i/o-intensive scientific workflows", in *ICCS: International Conference on Computational Science*, 2019.

[32] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, "A medium-scale distributed system for computer science research: Infrastructure for the long term", *Computer*, no. 5, pp. 54–63, 2016.

[33] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools", *Journal of Software Engineering Research and Development*, vol. 5, no. 1, p. 7, 2017.

[34] S. Singh and S. Kaur, "A systematic literature review: Refactoring for disclosing code smells in object oriented software", *Ain Shams Engineering Journal*, 2017.

[35] F. A. Fontana, E. Mariani, A. Mornioli, R. Sormani, and A. Tonello, "An experience report on using code smells detection tools", in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, IEEE, 2011, pp. 450–457.

[36] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment.", *Journal of Object Technology*, vol. 11, no. 2, pp. 5–1, 2012.

[37] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools", in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '16, Limerick, Ireland: ACM, 2016.

[38] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment.", *Journal of Object Technology*, vol. 11, no. 2, pp. 5–1, 2012.

[39] 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*, ACM, 2015, p. 7.

[40] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods", *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, 2011.

[41] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Ten years of jdeodorant: Lessons learned from the hunt for smells", in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, IEEE, 2018, pp. 4–14.

[42] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities", *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[43] M. Gottschalk, J. Jelschen, and A. Winter, "Refactorings for energy-efficiency", in *Advances and New Trends in Environmental and Energy Informatics*, Springer, 2016, pp. 77–96.

[44] M. V. Mantyla, "An experiment on subjective evolvability evaluation of object-oriented software: Explaining factors and interrater agreement", in *2005 International Symposium on Empirical Software Engineering, 2005.*, IEEE, 2005, 10–pp.

[45] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution", *Journal of systems and software*, vol. 80, no. 7, pp. 1120–1128, 2007.

[46] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering", *IEEE Transactions on software engineering*, vol. 28, no. 8, pp. 721–734, 2002.

[47] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering", in *2011 33rd International Conference on Software Engineering (ICSE)*, IEEE, 2011, pp. 1–10.

[48] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies", in *2010 Asia Pacific Software Engineering Conference*, IEEE, 2010, pp. 336–345.

[49] C. Seo, S. Malek, and N. Medvidovic, "An energy consumption framework for distributed java-based systems", in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ACM, 2007, pp. 421–424.

[50] S. Lafond and J. Lilius, "An energy consumption model for an embedded java virtual machine", in *International Conference on Architecture of Computing Systems*, Springer, 2006, pp. 311–325.

[51] J. Bloch, *Effective java*. Addison-Wesley Professional, 2017.

[52] A. Ward and D. Deugo, "Performance of lambda expressions in java 8", in *Proceedings of the International Conference on Software Engineering Research and Practice*, The Steering Committee of The World Congress in Computer Science, 2015, p. 119.

[53] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey", in *2013 20th Working Conference on Reverse Engineering (WCRE)*, IEEE, 2013, pp. 242–251.

[54] B. L. Sousa, P. P. Souza, E. Fernandes, K. A. Ferreira, and M. A. Bigonha, "Findsmells: Flexible composition of bad smell detection strategies", in *Proceedings of the 25th International Conference on Program Comprehension*, IEEE Press, 2017, pp. 360–363.

[55] R. Ferenc, L. Langó, I. Siket, T. Gyimóthy, and T. Bakota, "Source meter sonar qube plug-in", in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, IEEE, 2014, pp. 77–82.

[56] D. Taibi, A. Janes, and V. Lenarduzzi, "How developers perceive smells in source code: A replicated study", *Information and Software Technology*, vol. 92, pp. 223–235, 2017.

[57] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, "A medium-scale distributed system for computer science research: Infrastructure for the long term", *Computer*, vol. 49, no. 05, pp. 54–63, Mar. 2016.

[58] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management", in *Workshop on Job Scheduling Strategies for Parallel Processing*, Springer, 2003, pp. 44–60.

[59] K. Wang, C. Zhu, A. Celik, J. Kim, D. Batory, and M. Gligoric, "Towards refactoring-aware regression test selection", in *2018 IEEE/ACM 40th International Conference on Software Engineering*, IEEE, 2018, pp. 233–244.

[60] F. Ohata and K. Inoue, "Jaat: Java alias analysis tool for program maintenance activities", in *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, IEEE, 2006, 11–pp.

[61] L. Larsen and M. J. Harrold, "Slicing object-oriented software", in *Proceedings of IEEE 18th International Conference on Software Engineering*, IEEE, 1996, pp. 495–505.

[62] D. Liang and M. J. Harrold, "Slicing objects using system dependence graphs", in *Proceedings. International Conference on Software Maintenance*, IEEE, 1998, pp. 358–367.

[63] T. Ball and S. Horwitz, "Slicing programs with arbitrary control-flow", in *International Workshop on Automated and Algorithmic Debugging*, Springer, 1993, pp. 206–222.

[64] S. Kumar and S. Horwitz, "Better slicing of programs with jumps and switches", in *International Conference on Fundamental Approaches to Software Engineering*, Springer, 2002, pp. 96–112.

[65] M. Allen and S. Horwitz, "Slicing java programs that throw and catch exceptions", *ACM SIGPLAN Notices*, vol. 38, no. 10, pp. 44–54, 2003.

[66] K. Maruyama, "Automated method-extraction refactoring by using block-based slicing", in *ACM SIGSOFT Software Engineering Notes*, ACM, vol. 26, 2001, pp. 31–40.

[67] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander, "Decomposing object-oriented class modules using an agglomerative clustering technique", in *2009 IEEE International Conference on Software Maintenance*, IEEE, 2009, pp. 93–101.

[68] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems", *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, 2012.

[69] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: Identification and application of extract class refactorings", in *2011 33rd International Conference on Software Engineering*, IEEE, 2011, pp. 1037–1039.

[70] C. K. Roy, "Detection and analysis of near-miss software clones", in *2009 IEEE International Conference on Software Maintenance*, IEEE, 2009, pp. 447–450.

[71] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization", in *2008 16th iEEE international conference on program comprehension*, IEEE, 2008, pp. 172–181.

[72] G. P. Krishnan and N. Tsantalis, "Refactoring clones: An optimization problem", in *2013 IEEE International Conference on Software Maintenance*, IEEE, 2013, pp. 360–363.

[73] G. P. Krishnan and N. Tsantalis, "Unification and refactoring of clones", in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*, IEEE, 2014, pp. 104–113.

[74] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones", *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1055–1090, 2015.

[75] D. Mazinanian, N. Tsantalis, R. Stein, and Z. Valenta, "Jdeodorant: Clone refactoring", in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion*, IEEE, 2016, pp. 613–616.

[76] N. Tsantalis, D. Mazinanian, and S. Rostami, "Clone refactoring with lambda expressions", in *Proceedings of the 39th International Conference on Software Engineering*, IEEE Press, 2017, pp. 60–70.

[77] J. Visser, S. Rigal, G. Wijnholds, P. van Eck, and R. van der Leek, *Building Maintainable Software, C# Edition: Ten Guidelines for Future-Proof Code*. " O'Reilly Media, Inc.", 2016.

[78] T. J. McCabe, "A complexity measure", *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[79] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[80] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other", *The annals of mathematical statistics*, pp. 50–60, 1947.

[81] C. Seo, S. Malek, and N. Medvidovic, "Estimating the energy consumption in pervasive java-based systems", in *2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom)*, IEEE, 2008, pp. 243–247.

[82] R. F. K. Mparmpopoulou, “Monitoring greenclouds evaluating the trade-off between performance and energy consumption in das-4”, 2013.

[83] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, “Earmo: An energy-aware refactoring approach for mobile apps”, *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1176–1206, 2018.

[84] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier, “Runtime monitoring of software energy hotspots”, in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ACM, 2012, pp. 160–169.

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

Appendix A

Autoboxing

```
// Hideously slow program! Can you spot the object creation?
public static void main(String[] args) {
    Long sum = 0L;
    for (long i = 0; i < Integer.MAX_VALUE; i++) {
        sum += i;
    }
    System.out.println(sum);
}
```

Figure A.1: The mixing of primitives with objects can significantly decrease performance, example from 'Effective Java' by Joshua Bloch[51]

Since Java 1.5 autoboxing has been implemented. Autoboxing allows the mixture of primitives and boxed primitives. An example of autoboxing out of the book 'Effective Java' by Joshua Bloch [51] is shown in Figure A.1. Since the variable 'sum' is declared as Long, the variable 'i' is automatically boxed to an Long object and replaced by $Long.valueof(i)$. This induces the creation of about 2^{31} Long objects and significantly decreases performance.

Appendix B

Unit tests as workload

Listing B.1 shows an example of a main function that executes all the unit tests in the codebase and terminates with an error when a unit test has failed.

```
// Import JUnit4 runner
import org.junit.runner.Result;

public class TestRunner {
  private final static int AMOUNTOFTESTS = /* Amount of test that have to finish successfully */;

  public static void main(String[] args)
  {
    int repeats = validateArgument(args);

    System.out.println("Repeating " + repeats + " times.");
    runTests(repeats);
    System.out.println("Exiting successful");
    System.exit(0);
  }

  public static int validateArgument(String[] args) {
    if (args.length != 1) {
      System.err.println("You must specify the amount of loops by parsing one argument");
      System.exit(1);
    }

    try {
      return Integer.parseInt(args[0]);
    }
    catch (NumberFormatException e) {
      System.err.println("Parsed argument is not an integer: " + e.getMessage());
      System.exit(2);
    }

    return 0;
  }

  public static void runTests(int repeats) {
    for (int i = 0; i < repeats; i++) {
      Result result = runtests();
      if (!result.wasSuccessful() || result.getRunCount() != AMOUNTOFTESTS) {
        printError(result, i);
        System.exit(3);
      }
    }
  }

  private static void printError(Result result, int run) {
    System.err.println("Failed to run tests all " + AMOUNTOFTESTS + " tests succesfull");
    System.err.println("Run: " + run);
    System.err.println("Fails: " + result.getFailureCount());
    System.err.println("Runtime: " + result.getRunTime());
    System.err.println("RunCount: " + result.getRunCount());
```

```
    System.err.println("IgnoreCount: " + result.getIgnoreCount() + "\n");
    System.err.print("Failed runs:");
    for (int i = 0; i < result.getFailures().size(); i++) {
        String indent = "    ";
      System.err.println(indent + result.getFailures().get(i));
      System.err.println(indent + result.getFailures().get(i).getMessage());
      System.err.println(indent + result.getFailures().get(i).getTrace());
      System.err.println(indent + result.getFailures().get(i).getTestHeader());
    }
  }

  private static Result runtests() {
    return org.junit.runner.JUnitCore.runClasses(
        TestClass1.class,
        TestClass2.class
        /* All the other classes in the testing framework */
        );
  }
}
```

Listing B.1: Example of a driver class

B.1 Creating a runnable JAR

In order to be able to run the software on the DAS we created a runnable JAR file. To achieve this, we have to create a main function that can execute all the tests, an example of such a class is shown in Listing B.1. Next we have exported the project using Eclipse. How we exporeted the runnable JAR is explained in more details below:

1. Create a Runnable-JAR file
   (a) Open Export
   (b) Select Java/Runnable JAR file
   (c) Set launch configuration corresponding with the java file that executes all the unit tests (Listing B.1)
   (d) Select Package required libraries into generated JAR
   (e) Name the JAR file
2. execute the JAR file:
   "java -jar name_of_jar_file.jar #"
   # is an integer, it is the amount of times the whole set of JUnit test have to be repeated.

Problems with apache/commens-configuration. When importing the Maven project in eclipse there were two dependency errors. The *avacc-maven-plugin:2.6:javacc* plugin and the *maven-antrun-plugin:1.8:run* could not be found. In order to repair these errors, we had to compile the project with maven from the command line. Using *"mvn generate-resources"* the right resources were downloaded. However, the project could still not run in eclipse because the files were generated in the target map and Eclipse could not find them. By copying the missing java files in the main folder we were able to remove the compile errors. This did not have any affect on the refactoring since we did not refactor any of the changed files.

Additional there were problems with extracting a runnable JAR. The JUnit tests from the software required access to testing resources. These testing resources were hard-coded in the code and thus were required to be in a certain directory relative from execution. In order to work around this we had to manually add the test resources to the correct location from execution, this is explained in more detail below:

1. Create a JAR file
   (a) Open Export
   (b) Select Java/JAR file
   (c) Select all resources in project

(d) Only select: "Export all output folders for checked projects"
(e) Name the JAR file

2. Copy the 'target' folder with the generated resources to the path from which the JAR will be executed
3. When executing, manually add all the maven dependencies to the classpath and execute the main runner:
"java -cp name_of_jar_file.jar:folder_with_mvn_deps/* TestRunner #"
 # is an integer, it is the amount of times the whole set of JUnit test have to be repeated.

B.2 Unit test problem with apache/commens-lang

There is an JUnit test that will fail 1 in 1000 times, this unit test is shown in Listing B.2. When a single test fails, the driver class is configured to stop the program and mark it as failed. This means that when such a test fails it will create a completely different energy consumption as output. But as is shown in Figure 4.2, we check all the executed programs. We have found that after 9600 repetitions 7 had failed. These 7 fails have been re-run without any complication.

```
package org.apache.commons.lang3

/**
 * Test homogeneity of random strings generated --
 * i.e., test that characters show up with expected frequencies
 * in generated strings.  Will fail randomly about 1 in 1000 times.
 * Repeated failures indicate a problem.
 */
@Test
public void testRandomStringUtilsHomog() {
    final String set = "abc";
    final char[] chars = set.toCharArray();
    String gen = "";
    final int[] counts = {0,0,0};
    final int[] expected = {200,200,200};
    for (int i = 0; i< 100; i++) {
       gen = RandomStringUtils.random(6,chars);
       for (int j = 0; j < 6; j++) {
           switch (gen.charAt(j)) {
               case 'a': {counts[0]++; break;}
               case 'b': {counts[1]++; break;}
               case 'c': {counts[2]++; break;}
               default: {fail("generated character not in set");}
           }
       }
    }
    // Perform chi-square test with df = 3-1 = 2, testing at .001 level
    assertTrue("test homogeneity -- will fail about 1 in 1000 times",
        chiSquare(expected,counts) < 13.82);
}
```

Listing B.2: JUnit test that will fail 1 in 1000 runs

Appendix C

Scripts

In Listing C.1 we show an example of a batchjob that gets send to the SLURM queue on DAS-5. The variables on the top of the script have to be adjusted to what programming you are trying to run. This batchjob is created for the execution of Java JAR files, but it can easily be adjusted to run any other kind of application on the DAS and measure the energy consumption.

```
#!/bin/sh
#SBATCH -N 1 -w "node029"
#SBATCH --output="results/%x-%j.out"

# Update accordingly
port=2                                  # see portmapping
repeats=150000                          # repeats of unit test framework
programname="classmate-duplicate"       # jarfile to execute (DAS-5)
measure_script="/home/user/scriptname"  # Location of measurement script (DAS-4)
kill_measurement="/home/user/scriptname" # Location of kill script (DAS-4)
output="/home/user/outputlocation/"     # Location of output (DAS-4)

pdu_measurement="$programname-$port-$repeats-$SLURM_JOB_ID.csv"
pdu_idle="$programname-$port-$repeats-$SLURM_JOB_ID-idle.csv"
command="java -jar programs/$programname.jar $repeats"

# Report running information
echo "Running on: `hostname` time: `date`"
echo "JOBID: $SLURM_JOB_ID"
echo "Measurement script (DAS-4): $measure_script"
echo "Kill script (DAS-4): $kill_measurement"
echo "Run: $command"

# Make sure no measurement is running
ssh skok@fs0.das4.cs.vu.nl $kill_measurement

# Start idle measurement
ssh skok@fs0.das4.cs.vu.nl $measure_script -o "$output$pdu_idle" -p $port &

sleep 60

# Stop idle measurement
ssh skok@fs0.das4.cs.vu.nl $kill_measurement

# Start measurement program
ssh skok@fs0.das4.cs.vu.nl $measure_script -o "$output$pdu_measurement" -p $port &

# run program
`$command`

# Stop program measurement
ssh skok@fs0.das4.cs.vu.nl $kill_measurement
```

Listing C.1: Example of a batch job. Running on DAS-5.

In Listing C.2 we show a script that is used to start power readings. The port (what node to measure) and the output location have to specified. The script will make a snmpwalk connection to the PDU (located at ip). The PDU gives a lot of information, but we are only interested in the timestamp and the power. Since it cost a lot less time to request less information we have limited the request to stop after 'pStatePortCur'. This has optimised the sampling by a factor of 25. The script will keep reading the energy consumption untill its process is killed.

```
#!/bin/bash

# This is the last known port to node mapping:
# port 0: unused
# port 1: unused
# port 2: DAS-5/node029
# port 3: DAS-5/node028
# port 4: DAS-5/node027
# port 5: DAS-5/node026
# port 6: DAS-5/node025
# port 7: DAS-5/node024

# o filename output on das-4
# p port to measure
while getopts "o:p:" opt; do
  case $opt in
    o)
      output=$OPTARG
      ;;
    p)
      port=$OPTARG
      ;;
  esac
done

if ! test "$output" ; then
    echo "-o is obligatory (OUTPUT NAME)"
    exit 1
fi

if ! test "$port" ; then
    echo "-p is obligatory (PORT NODE USING [2-7])"
    exit 2
fi

ip="10.141.0.195"
mib="/var/scratch/versto/Racktivity/ES-RACKTIVITY-MIB.txt"
node=".1.$port"

# Format
echo "Power(W), TimePDU(S), TimeNODE(mS)" > $output

# Continues measure
while true
do
# Local time node das4
querry_time=$(date +"%s%3N")

# SNMP connection PDU, Since we dont need al the information stop at pStatePortCur to optimise speed
result=$(snmpwalk -v 1 -CE pStatePortCur -c public -t 9 -M .:/usr/share/snmp/mibs -m $mib $ip EPowerEntry)

# Extract power
power=$(fgrep "pPower$node" <<< "$result")
power_number=$(echo "$power" | grep -o -E '[0-9]+'| tail -1)

# Extract timestamp PDU
timestamp=$(fgrep "pCurrentTime" <<< "$result")
timestamp_number=$(echo "$timestamp" | grep -o -E '[0-9]+' | tail -6 | head -n 1)

# Log to file
echo "$power_number, $timestamp_number, $querry_time" >> $output
done
```

Listing C.2: Continues read Power from specific Node. Running on DAS-4

In order to be able to kill to process we have created a script that will kill it. The script is shown in Listing C.3. The script works by requesting all the processes from a user (specified in the script) that have a script running with a certain name (specified in the script) and we send a kill signal to all of them.

```
#!/bin/bash
# You will need access to be to kill processes of that user

# The name of the file for reading the power from a node.
scriptname=""
# The name of the user running the script
username=""

# Get all process with name from user
process=$(pgrep -u $username $scriptname)

# Kill all processes
for i in $process
do
kill $i
done
```

Listing C.3: Kill all running measurement scripts. Running on DAS-4.

Appendix D

Boxplots

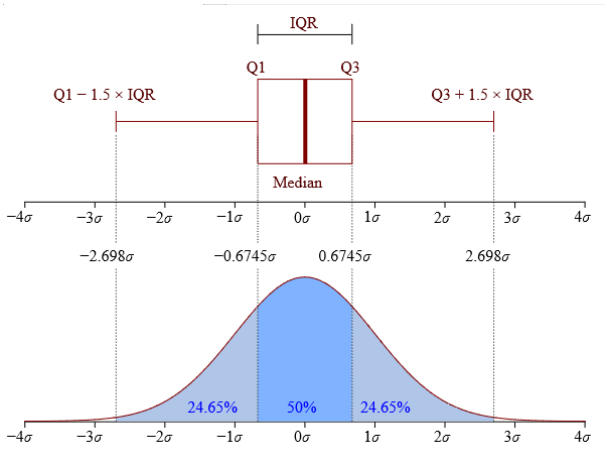

Figure D.1: Boxplot example, adopted from Jhguch[1]

Boxplots are especially useful for comparing multiple data-sets. The line in the middle of the boxplot is the independent median statistic. The upper (Q3) and lower (Q1) side of the box are called quartiles. The Interquartile Range (IQR) is the difference between the two. The quartiles are located respectively at median + 25% and median - 25%. The upper and under bound are called whiskers and are located at a distance of 1.5 times the IQR from their corresponding quartile. An example of a boxplot is shown in Figure D.1, alongside with a probability density plot of a normal distribution.

[1] https://en.wikipedia.org/wiki/File:Boxplot_vs_PDF.svg