Summary

The topic of this master thesis is the reduction of energy within software as a result of handling code smells in code bases.

For this purpose, previous papers are brought forth which highlight a connection between the inclusion of code smells and a higher overall energy consumption of a program. Many of these tests however are made with different programming languages and system platforms in mind. The first contribution of the paper thus is to gather smells found to have an impact within different languages and platforms, and test whether they also contribute to an increased energy consumption when ported to C# on a desktop system.

Initially, microbenchmark tests are setup to test individual smells. This is done by creating two semantically equivalent programs, where one contains the desired smell and the other is a smell free version. When tested, this then highlights the performance impact of the smell itself, while limiting the influence of outside factors.

Through the micro benchmark tests, it is found that most of the found smells reduce overall energy consumption when removed from the program, some individual smells by as much as 60%. However, some of the smells induced an increase in energy consumption when refactored, in contrast to the literature. Given that the language used in this report is different from those used in the literature, this may indicate that different constructs have varying energy consumption across different languages.

To further test the smells a "Super Smell" is created, the Super Smell has the ability to include various smells in each of its test runs. This makes it easy to combine smells within a more coherent and larger program. Previous studies suggest that some combinations of smells may, when fixed, increase total energy consumption more than if only one or none of the smells had been fixed. The Super Smell helps test this theory for C# by potentially identifying smells which provide larger energy savings when fixed individually as opposed to together.

The Super Smell itself is first run with every smell implemented. For the sake of scale, the 5 smells with the highest impact on energy consumption are then run in sets of 2, 3, 4 and 5 combined. This allows for a comparison between the individually run smell and the performance of multiple smells run at once.

Through the Super Smell tests, it was found that some smells do indeed have this behavior and with certain combinations, handling a smell can have a negative impact on overall energy consumption. However, in all cases, the best performing combination of smells in regards to energy consumption, was always a smell free program.

Lastly a linter was created which helps identify smells within any C# program, the purpose being to inform developers of areas within their code where a potential

reduction in energy can be achieved by removing/handling certain smells.

To test the linter, first it is run on the previously created tests for both the microbenchmark tests and the Super Smell. Both of these contain cases for every smell mentioned in the paper and handled by the linter. As such a list of the amount of expected smells to be found are created for each. Running the linter on these programs proved to find not only the expected smells, but also some which we had inadvertently created when making the smells themselves.

The linter itself has not been tested on real world applications yet, however this will happen at a later date before the defense of the paper at which point these results will be presented and whether the tool has practical use in the wild is revealed.

An Energy Aware Linter

Helping programmers make more energy efficient code on the fly

Master Thesis

Rasmus Hartvig og Jonas Krogh Hansen

Aalborg University
Computer Science

Computer Science 9th semester
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
An Energy Aware Linter
Helping programmers make more energy efficient code on the fly

Theme:
Programming Technology

Project Period:
Spring Semester 2022

Project Group:
cs-21-pt-9-01

Participant(s):
Jonas Krogh Hansen
Rasmus Hartvig

Supervisor(s):
Bent Thomsen
Thomas Bøgholm

Copies: 1

Page Numbers: 91

Date of Completion:
June 9, 2022

Abstract:

Energy consumption of software has become a prominent focus area for researchers and developers. Much work has gone into researching patterns and constructions that consume more energy than others, across programming languages and platforms. Code smells have been examined to a large extent, however not much research has gone into examining if the results carry over to other languages or platforms. In addition to this, not many tools for detecting these issues exist. This report examines a collection of code smells that have been proven to have an impact in Java on desktop systems and C++ on embedded systems to see if they have the same impact in C# on a desktop system, in both a micro- and macrobenchmarks. Initial tests show that removing a number of code smells from a macro system can save up to 23% in energy consumption. To ease the process of such refactoring, an effort is made to create a linter-like tool that can detect such patterns using the analysis engine CodeQL.

Contents

Preface

This master thesis is a student project on the 4th semester of the Master of Software at Aalborg University. This project spans from 1st of February 2022 to 10th of June 2022. The project is supervised by Bent Thomsen[1] and Thomas Bøgholm[2].

Aalborg University, June 9, 2022

Rasmus Hartvig
<rhartv17@student.aau.dk>

Jonas Krogh Hansen
<jh17@student.aau.dk>

[1] https://vbn.aau.dk/da/persons/110568
[2] https://vbn.aau.dk/da/persons/112525

Chapter 1

Introduction

Since 2010 the demand for energy in the information and communications technology (ICT) sector has been increasing at a steady pace, and the demand is projected to further increase in the future. As of writing, the ICT-sector accounts for more than 2% of global energy consumption, of which data-centers account for 0.3%[5].

Meanwhile, actions are already being taken to improve the infrastructure and lower the energy consumption within the ICT sector. For example, in [15] A. Shehabi et al. uses the term 'hyperscale-shift' to describe the shift towards the opening of big data-centers that are optimized for energy and cooling efficacy. Concurrently, smaller, less energy efficient data centers are being shut down, to make way for larger, more energy efficient data-centers.

However, while changes to the infrastructure of various technologies within the sector are helpful, changes can still be made to make software itself run more efficient, and with lower energy consumption. For example, [13] W. Oliveira et al. showcases the possibility of great reducing overall energy consumption, by using different collections within the Java programming language.

A found cause of increased energy consumption is that of code smells, where as shown by Verdecchia et al. some smells have been found to increase the power consumption of their corresponding programs by up to 49%. These tests were made in Java and in this report, we wish to see if the same power increases can also be seen in systems written in C#, and as such set up various test smells for this purpose.

In the same paper, Roberto Verddecchia et al. also showcase how fixing multiple smells at once, can potentially provide less of an energy saving than merely fixing one of them individually. To further test this theory in this report, the smells found to have the highest impact on energy performance are tested in pairs to see if resolving them in various combinations might provide a greater energy saving than simply fixing every smell present. [4]

Lastly, with the found smells in mind, a linter is created, to identify the smells,

in any sort of program written in C#, which have an impact on energy consumption. This way, developers can easily use said linter to identify and fix potential smells within their C# programs, which would otherwise have a negative impact on overall power consumption.

Chapter 2

Related Work

This chapter builds on top of the related works chapter of our previous report [12] by including a couple of papers regarding code smells and a wide variety of linter frameworks that allow for extension and customization.

2.1 Prerequisite knowledge

This section will briefly introduce linters as an important point of prerequisite knowledge.

2.1.1 Linter

A linter is a static code analysis tool which can be used to flag bugs, stylistic errors and suspicious constructs in a codebase. In other words, a linter is a tool which programatically scans your code with the purpose of finding issues that can lead to potential bugs or inconsistencies with your code. One of the big advantages of a linter is that it uses static analysis, this allows errors to be found without having to execute the code beforehand. This is especially useful in dynamically typed languages, as these do not enforce as many or as strict rules on the programmer prior to execution.

Aside from finding bugs and syntactic errors, linters can also be used to improve on constructs with the objective of improving the run-time of the program. This might be done by having the linter suggest an alternative and more time-efficient collection. For example, should the user attempt to use a list for a collection which they frequently perform look ups within, the linter might suggest using a dictionary or hash map instead to improve performance. [17]

Another purpose of a linter is that it can condition programmers to stylize their code in a certain way, this can create uniformity in the code written among a group of programmers and make it easier to both review and edit parts of the code

which they did not originally work on. Additionally this can help especially newer programmers who might not be as into the workflow as a more veteran developer, and thus can have additional use of the linter's features.

This concept also extends to linters which attempt to improve run-time or other performance metrics. As developers are not only informed when an improvement can be made to their codebase, but can bring that knowledge to future code they write, as they have now been shown a better alternative to their original way of handling the error, thus allowing for the creation of "better" code even in cases where the linter is not in use.

2.2 Energy Code Smells

This section will summarize relevant related papers that have measured the energy consumption of code smells in software.

2.2.1 Definition, Implementation and Validation of Energy Code Smell: an Exploratory Study on an Embedded System

Vetro et al. has conducted an exploratory study on the energy consumption of code smells [1]. They selected nine common code smells that they tested on an embedded device, namely the Waspmote[1]. For each code smell they wrote a function with and without said code smell, and for each 50 samples of one million executions were collected.

They found that five of the nine code smells had a statistical significant difference in energy consumption once the code smell had been refactored, albeit the difference is very minor in the order of roughly 30-250 µW. This is because the tests performed were on very small functions that perform only a few operations, meaning they can be classified as microbenchmarks. They do however recognize this fact, and note that the impact would likely be larger in a more computationally complex scenario.

2.2.2 Empirical Evaluation of the Energy Impact of Refactoring Code Smells

Verdecchia et al. conducted an empirical experiment examining what impact refactoring code smells have on energy consumption [4]. They selected five common code smells that are feasible to refactor automatically. For their experiments they selected three open-source **ORM**-based (Object-Relational Mapping) Java applications, namely `JTrac` (14,000 **LOC**)(Lines of Code), `CashManager` (2,000 LOC), and `PetClinic` (2,000 LOC).

[1] `https://www.libelium.com/iot-products/waspmote/`

Their results showed that `JTrac` saw a significant decrease in energy consumption of up to 49% for certain code smells. The other two however, only saw a small to negligible difference. They attribute this difference to the fact that `JTrac` has seven times as many lines of code, and is far older than the other applications. For the latter, this is backed up by Fowler who argues that the age of an application may have an impact on the amount of code smells present [9].

2.3 Existing Linter Frameworks

This section will explain some of the widely used linting frameworks that allow for extending their ruleset.

2.3.1 Roslyn analyzers

Roslyn analyzers is a set of analyzers that examines a C# code base for various issues such as coding style and maintainability. Roslyn is the .NET compiler platform, and the Roslyn analyzers have been included in this platform since .NET 5. As such, once a C# project is compiled using Roslyn, the code will also be analyzed underway, and any errors or warnings found during this process will be reported. [11]

Roslyn analyzers allow for some level of customization through an `EditorConfig`, a config file in which settings such as indentation size, tab width, charset, and more can be overwritten from the defaults. In addition to this, the Roslyn analyzers are open source[2] and as such it is possible to define custom rules as well. However, rules are embedded in the analyzers themselves and in order to add any rules it is necessary to manually build the analyzers rather than just downloading a binary from the official source.

2.3.2 SonarQube

SonarQube is an open-source platform developed by SonarSource[3], that can be used to perform static analysis to find bugs, code smells as well as estimate code debt. SonarQube by default includes a large library of rules, which can individually be toggled as seen fit to fit the purpose of the current project, these rules are designed to help identify the aforementioned issues.

When run, SonarQube initially sets up a server to communicate found errors to the developer, this server can be accessed by default at `localhost:9000`. Once the server is running, SonarQube can be used to make a static analysis of the program we desire to analyze. Once the analysis is complete, the results are sent to the

[2] dotnet/roslyn-analyzers

[3] `https://www.sonarqube.org/`

SonarQube server, where they are presented through SonarQubes' own interface, after which the found bugs and code smells can be fixed. To make this process easier, SonarQube also provides an IDE extension tool called SonarLint, this helps apply the changes to the code base after the results are sent to the server.

Custom rules can be created to identify user defined problems with the code. The process of implementing these however, is dependent on what language is being analyzed. The implementations of the custom coding rules are split into 3 categories [16].

- **XPathj 1.0:** Supported by SonarQube's own web interface, here templates can be found to create custom rules through the server side of SonarQube.
- **Java:** New rules can be written in Java which uses SonarQubes API
- **Roslyn rules:** Issues can be imported from third-party Roslyn analyzers

A single language can have support for multiple of these custom rule implementation methods, as well as some of the languages not having any form of support for custom rule sets. It should be noted that in order for either Java or Roslyn rules to be added, these have to be written seperately from the tool set given by SonarQube and implemented thereafter.

2.3.3 Semgrep

Semgrep[4] is a static analysis tool built around rules for detecting specific patterns in source code. Semgrep supports a wide variety of languages, among these are C#, Java, Python, and many more. Rules are written in a syntax that resembles the code written in the given language, with the addition of several generic patterns.

Semgrep exists as a CLI that is easily installed using the Python package manager `pip`. To analyze a file, semgrep can be invoked as e.g.,

```
semgrep -config /path/to/rules /path/to/source/file
```

In addition to this, there exists third-party extensions for Intellij IDEA, Visual Studio Code, and Vim.

Semgrep provides a community-sourced registry of more than 1800 rules and more than 50 rulesets for various languages, as well as categories such as security, maintainability, and correctness. Rules are written in YAML files with a simple syntax, making it easy to implement your own rules catered for your own project. An example rule designed to detect passing parameters by value to a function can be seen in Snippet 2.3.1.

[4] `https://semgrep.dev/`

```yaml
rules:
- id: pass_by_value
  patterns:
    - pattern: |
        $F(..., $VALUE, ...);
    - metavariable-regex:
        metavariable: $VALUE
        regex: ([0-9]+\.?[0-9]*)|("[a-zA-Z0-9]*")
  message: Pass by value detected in function
  languages: [csharp]
  severity: WARNING
```

Snippet 2.3.1: Semgrep rule for detecting passing parameters by value to a function in C#.

The rules in the ruleset are enclosed in the `rules` scope. Each rule has an identifier, as well as a message that is to be displayed when the rule is violated. A rule must also defined which languages it applies to, as well as its' severity level. Finally, you can define a number of conditional or non-conditional patterns that causes this rule to trigger.

```
1  var some_int = 123;
2  var some_float = 123.0;
3  var some_string = "some_string";
4
5  Foo(12, "some_string", 12.2);
6  Foo(some_int, some_string, 12.2);
7  Foo(some_int, some_string, some_float);
8
9  void Foo(int i, string str, float f) {}
```

Snippet 2.3.2: Example C# code used to detect passing parameters by value to a function.

Using semgrep with the aforementioned rule on the code in Snippet 2.3.2, we find that lines 5 and 6 are caught by the linter tool, while line 7 is not.

2.3.4 InsiderSec

InsiderSec[5] is a static analysis tool with a primary focus on security. Specifically they aim for covering the OWASP top 10 security vulnerabilities[6], and it supports

[5] https://github.com/insidersec/insider
[6] https://owasp.org/Top10/

Java, JavaScript, C#, Android, and iOS. Rules are written directly in the code and as such it is required to re-compile the tool for them to take effect, unlike tools such as Semgrep that allows defining rules in a config-style separate from the application. An example rule can be seen in Snippet 2.3.3.

```
var CsharpRules []engine.Rule = []engine.Rule{
    Rule{
        ExactMatch:
        ↪ regexp.MustCompile(`.*=\s+new\sProcess\(\);(?:.*)*(.*\.StartInfo\.Arguments\s+=\s+.*".*\+)`)
        CWE:           "CWE-78",
        AverageCVSS:   3,
        Description:   "The dynamic value passed for the execution of the
        ↪ command must be validated.
        ↪ https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo?view=netframew
        Recomendation: "",
    }, ...
}
```

Snippet 2.3.3: Example rule included in the official InsiderSec ruleset

½ As seen in the snippet, rules are matched based on regex. Because of this, rules will primarily be limited to specific expressions under a given language's syntax.

The developers of InsiderSec provide pre-compiled binaries of a command-line tool for Linux, macOS, and Windows, making it straightforward to use regardless of platform. It can be invoked as follows

./insider –tech csharp –target <directory>

in order to analyze a project folder.

2.3.5 CodeQL

CodeQL[7][8] is an analysis engine developed by the GitHub team, and currently supports several languages, including C, C#, Python, and Java. CodeQL uses the query language `.ql` in order to detect various errors. To do this, a database representing a codebase must be created as a prerequisite step. Genereating a database is done through an *extractor* included in CodeQL for all supported languages. Once the database is generated, queries are executed in order to detect issues. Included in CodeQL are a large variety of queries written by the GitHub team as well as the

[7] `https://github.com/github/codeql`
[8] `https://codeql.github.com/docs/`

community spanning several categories, such as bugs, performance, bad practice, and security. Once the codebase is analyzed the issues, if any, are reported in a manner defined by metadata included in a query. If CodeQL is run in a supported IDE, e.g., Visual Studio Code, the issues are displayed directly in the code much like a linter.

A CodeQL query includes metadata, imports, and a `select` clause. It is also possible to define classes and predicates for a query. An example query for detecting empty catch blocks in C# can be seen in Snippet 2.3.4.

```
1  /**
2   * @name Poor error handling: empty catch block
3   * @description Finds catch clauses with an empty block
4   * @kind problem
5   * @problem.severity recommendation
6   * @precision very-high
7   * @id cs/empty-catch-block
8   * @tags reliability
9   *       readability
10  *       exceptions
11  *       external/cwe/cwe-390
12  *       external/cwe/cwe-391
13  */
14
15 import csharp
16
17 from CatchClause cc
18 where
19   cc.getBlock().isEmpty() and
20   not exists(CommentBlock cb | cb.getParent() = cc.getBlock())
21 select cc, "Poor error handling: empty catch block."
```

Snippet 2.3.4: `.ql` query for empty catch blocks [10]

The block comment on lines 1-13 contains the metadata for the query, and must be present in order to get reports about this issue. The metadata should contain general information about the query, such as the name, a description, its' type, the level of severity, the level of precision (i.e., a rate of true positives), a unique id, as well as some tags that help group queries together in categories.

Followed by the metadata are the imports. At least one import must exist, namely the given language the query is written for - in this case, `csharp`. Also included are a large number of libraries[9] for each language, that may be imported

[9] `https://codeql.github.com/codeql-standard-libraries/csharp/`

for various purposes, such as classes for control flow structures, frameworks, types, and more.

The imports may be followed by a number of classes and predicates necessary to heighten readability of, or simply to perform the query.

Finally on lines 17-21 is the query that is to detect the issue. These queries are akin to database languages such as SQL in their structure. The `from` clause includes variable declarations used in the query. The `where` clause includes logical statements that are the conditions for the selection to happen. The `select` clause denotes what should be returned in the case the `where` clause holds true. In this case, the `CatchClause` variable `cc` with the accompanied string as an error message.

Chapter 3

Methodology

This chapter will go over the methodology behind the experiments that will be performed. Firstly the test environment will be explained, including the test machine and the room it is situated in. Then a brief explanation of the approach to measuring energy consumption of the experiments, and the structure of the implementation of the code smells that will be used for the experiments. Finally, the iterations in the code smells and the sample size of the experiments will be explained.

3.1 Test Environment

This section will explain the test environment that has been used to test the various code smells. A test system located in a group room at the university has been used for every test. One uncontrollable factor in this situation is the temperature of the group room. Temperature has previously been shown to have an impact on the performance of a computer, and as such the temperature of the CPU will be measured along with the power consumption during the tests.
The hardware specifications of the test system is listed below:

- Computer type: Optiplex 5050 (07A2)
- Motherboard: Dell 0WWJRX
- RAM: 2x 8GB; 2400 MHz; DDR4; DIMM; Micron Technology (8ATF1G64AZ-2G3B1) and Hynix Semiconductor (HMA81GU6AFR8N-UH)
- CPU: Intel i7-6700; Skylake; 3.4GHz (overclock: 4.2GHz); 64bit; clock 100MHz
- GPU: integrated
- Disk: 256GB SSD; INTEL SSDSC2FK25

In addition, the software specifications are listed below:

- OS: Ubuntu Server 21.10, minimal install
- Login manager: `lightDM`
- Window manager: `i3-wm` as window manager (and as such, the systems runs X11)

A minimal installation of Linux, as well as the login and window managers were selected as they are light weight and as such to avoid as much unnecessary overhead as possible. In addition to this, Linux was also selected in order to ease the process of measuring power consumption using RAPL, as this process is more complicated on operating systems such as Windows given that there is no access to the RAPL driver.

3.2 Measuring energy consumption

In order to measure the energy consumption of a given code smell, a tool named `rapl.rs`[1] is used. `rapl.rs` was developed for benchmarks on our previous semester, and is a command-line benchmarking tool that measures the energy consumption of the system in the period that some input program executes. Once the program finishes executing various data about the energy consumption is printed to the terminal, including energy consumption in joules and watts, and the temperature of the CPU. This data is also saved to a csv file with a line for each poll. The energy is recorded by polling Intel Running Average Power Limit (**RAPL**) available on Intel CPUs since the Sandy Bridge generation [7]. Figure 3.1 shows the workflow for the `benchmark` tool in `rapl.rs`.

[1] cs-21-pt-9-01/rapl.rs

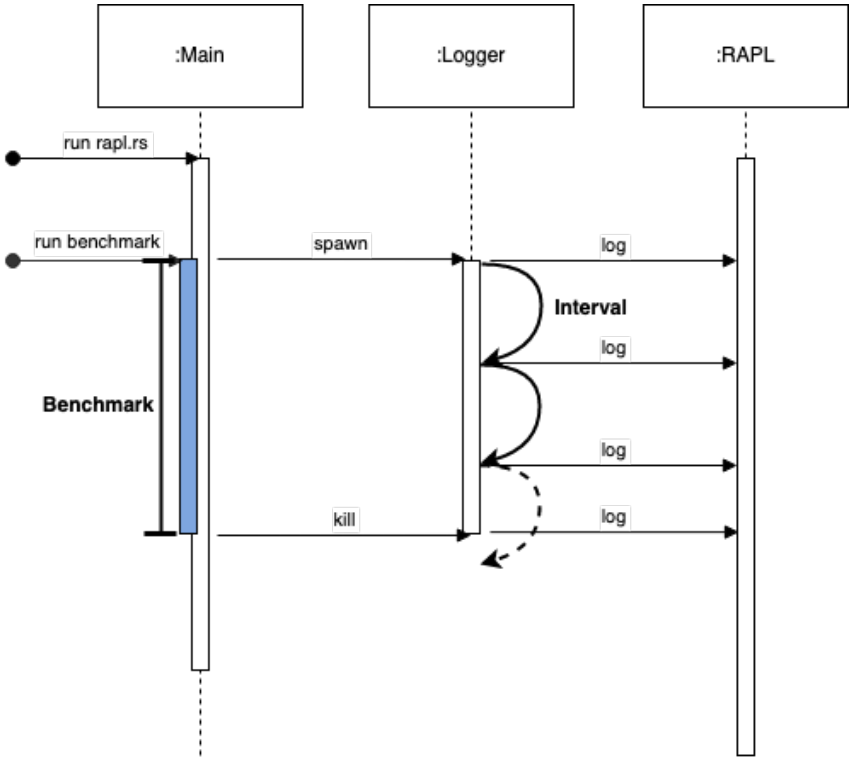

Figure 3.1: Flow diagram of the `benchmark` tool in `rapl.rs` [12]

The diagram is divided into three components: the main thread, the logger thread, and the RAPL driver. The main thread starts once `rapl.rs` is invoked from the terminal. The purpose of the main thread is to perform some initial tasks, such as spawning the logger thread, after which it will execute the benchmark. The main thread will then wait for the benchmark program to terminate on its' own, which will trigger a kill signal to the logger thread. When the signal is sent, the main thread will wait for the logger thread to finish before continuing.

The logger thread is in charge of logging energy measurements to a csv file. Measurements will be logged at an interval specified through the command-line. When it is time to log, the logger thread will read the current consumption in microjoules from the RAPL driver, convert it to other metrics such as joules and watts, and then write it to the csv file. Once the main thread sends the kill signal, the logger thread will perform one last measurement and then die.

Once the main thread is done waiting for the logger thread, it will print the final results to the terminal and exit.

3.3 Test implementation

This section will give a brief explanation of the implementation of the code smells. This includes the language and framework used, the architecture surrounding the code smells, as well as the structure of the code smell implementations.

3.3.1 Language and Framework

The code smells and their surrounding architecture will be implemented in C#. This is done for two reasons. Firstly because much of the existing literature examines Java, and as such C# was chosen to see if the results found carry over to another, similar OOP language. Secondly, it was done because much of the existing literature examines Android on mobile devices, and as such a language mostly used for more conventional computers was chosen in order to see if these results carry over to desktops and laptops.

The code smells use .NET Core version 6, meaning C# version 10 is used.

3.3.2 Code smell Architecture

This section will go over the architecture surrounding the code smells as well as how they are executed internally.

All the code smells are implemented in a single program with a surrounding architecture that allows for the user to execute a specific variant of a specific code smell.

The code smell program accepts a set of command-line arguments denoting what smell to run. The program is executed as `./Smells <smell> <variant>`, where `<smell>` is the smell to run, and `<variant>` is the variant to run: *bad* for the smell, or *good* for the refactored version.

Dispatch

Once the command-line arguments are parsed, they are passed to the `Dispatch` class, which is in charge of selecting and executing the smell to run. Each code smell has its own method for execution, and a `switch` will make the selection based on the input.

```
private void RunLongMethod(string variant)
{
    int iterations = 5000000; // 5M

    LongMethodBase LongMethod;
    if (variant == "bad") LongMethod = new LongMethodBad();
    else LongMethod = new LongMethodGood();

    Console.WriteLine("Running code smell Long Method, variant " + variant);
    for (int i = 0; i < iterations; i++) LongMethod.Compute();
    Console.WriteLine("Done");
}
```

Snippet 3.3.1: Run method for the Long Method code smell

Snippet 3.3.1 shows the implementation for the run method for the Long Method code smell. The relevant class is instantiated depending on the value of `variant`, after which the `Compute()` method is run a number of times. The run methods for the remaining code smells follow the same structure.

Code smell structure

The implementations of the code smells all follow the same structure. A class is implemented for both the bad variant and the refactored version, and both of these inherit from the same base class. An example of this structure can be seen in Snippet 3.3.2.

```
namespace Smells.CodeSmellExamples
{
    public abstract class LongMethodBase
    {
        public abstract void Compute();
    }

    public class LongMethodBad : LongMethodBase
    {
        public override void Compute()
        {
            // implementation of the bad variant
        }
    }

    public class LongMethodGood : LongMethodBase
    {
        public override void Compute()
        {
            // implementation of the refactored variant
        }
    }
}
```

Snippet 3.3.2: Example of code smell implementation structure

The reason behind this structure is in part to ensure readability, but also to ensure that the run methods in the dispatch class are kept simple.

3.4 Iterations and Sample Size

Each code smell runs for its own fixed number of iterations. For example, as seen in Snippet 3.3.1 in the previous section, Long Method is executed five million times per run. This is done in order to allow a code smell to run for a few seconds, thereby providing results on a greater scale than if they were run only once, making the results easier to compare. In addition to this, every code smell is run for the same number of iterations in both the good and bad variants. A code smell running for this fixed number of iterations will be referenced as a singular run of the code smell for the remainder of this section.

As for the sample size when running the benchmarks with `rapl.rs` 400 was chosen. This means for a given test, the code smell is tested on 400 runs. This number is chosen based on results found from Cochran's Formula for sample sizes,

as explained in our 9th semester report [12].

Chapter 4

Analysis

This chapter will explain the different code smells that have been selected, show the used implementation and the results gained from the experiments, and finally an analysis of the results, including an IL or assembly inspection where necessary.

In addition to this a "Super Smell" has been created that implements all the tested smells, which is then extensively tested in different ways in order to examine how the smells interact with each other with regards to energy consumption and execution time. The results will be displayed in tables and will be explained and analyzed.

4.1 Energy Costly Patterns

This section will explain a number of code smells that have been proven to have an impact on the energy consumption of software. Beck & Fowler define *Code Smells* as certain code structures that indicate a need for refactoring [8].

Given the source of the research, a lot of these have only been proven to be energy code smells when writing a system for Android mobile devices, not more conventional desktop applications. Therefore, tests are also created in this section to further analyze whether these code smells carry over to other languages, specifically C#.

In general, the tests are made by having two pieces of code with the same functionality. However one snippet contains the given code smell, while the other is written as the "correct" solution. Given that the examples are often rather small, they are for the most part repeated a number of times. This is to better accentuate the performance difference, with more readable and tangible numbers. As a single run executes too fast to get relatable information. Given both implementations are looped the same amount of times, it is deemed that the overhead for either implementation should be about the same, and not cause a discernible difference.

An initial set of 100 runs are given to each test, after this Cochran's formula is

used on this sample size to find the minimum amount of required tests to reach a 95% confidence interval. [6] [14]

The summary in the end of this section includes an overview of the results of the tests in Table 4.1.

4.1.1 Duplicate Code

Duplicated code appears as highly similar code snippets, with maybe a few differences. Beck & Fowler suggest amending this smell by finding a way to combine these snippets, e.g., by extracting the differences to a separate method.

In [4] Verdecchia et al. found that Duplicated Code can also have an impact on energy consumption. For `JTrac`, they found that refactoring this code smell reduced the energy consumption by 10.7%. In the paper the authors used a tool to automatically detect and refactor these smells, and as such it is unknown exactly what code was refactored in their experiments. For this reason, a new example of the duplicate code smell will have to be set up.

In order to test the effect of duplicate code two separate versions of a method are implemented. One implementation includes the code smell, and in the other the smell has been refactored. The implementation can be seen in Snippet 4.1.1 and Snippet 4.1.2.

```
1  public override void sumElements()
2  {
3      int sum_a = 0;
4      int sum_b = 0;
5
6      for (int x = 0; x < 4; x++)
7          sum_a += list_a[x];
8
9      int average_a = sum_a / 4;
10
11     for (int x = 0; x < 4; x++)
12         sum_b += list_b[x];
13
14     int average_b = sum_b / 4;
15 }
```

Snippet 4.1.1: Duplicate code code smell

```
1  public override void sumElements()
2  {
3      int sum_a = 0;
4      int sum_b = 0;
5      for (int x = 0; x < 4; x++)
6      {
7          sum_a += list_a[x];
8          sum_b += list_b[x];
9      }
10
11     int average_a = sum_a / 4;
12     int average_b = sum_b / 4;
13 }
```

Snippet 4.1.2: Duplicate code code smell, refactored

The snippets both sum the elements of two lists. The duplicate code example does this separately for both lists, while the non duplicated version sums the results

simultaneously in the same loop. [1]

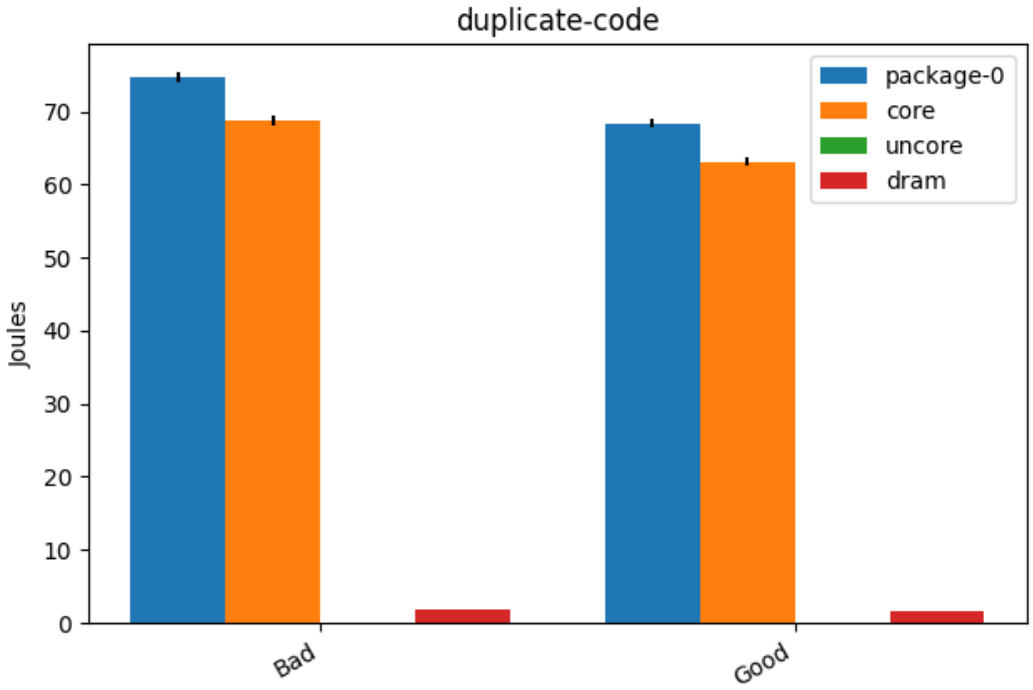

Figure 4.1: Duplicate code energy consumption

As seen in Figure 4.1, there is a slight decrease in energy consumption when the duplicate code smell is handled. On average, the smell variant uses 74.75 joules per run, while the "good" version uses 68.41 joules per run, resulting in a total energy saving of about 8.48%. This is somewhat smaller than the reported reduction of 10.7% from android tests, however is nevertheless a decent reduction. This change is rather expected, as the duplicate code is set up such that an extra loop is executed for the smell version, which also increases the average time spent on execution for the smell variant. Execution time for the smell variant was an average of 4.14 seconds while the non smell variant had an average run time of 3.72 seconds. This is roughly a 10.14% decrease in time spent for the refactored variant. This also means that even discounting run time, the good variant spends less overall energy per unit of time, as well as having a faster execution speed making it the clearly superior version among the two.

[1] CodeSmellExamples/DuplicateCode.cs

Analyzing results

As mentioned before, the refactored version shows a moderate decrease in energy consumption. The cause for this is evident by simply inspecting the code: the bad variant contains two loops, whereas the good variant contains one loop. Having two loops doubles the number of instructions performed, and it is therefore safe to assume that this will increase the energy consumption.

4.1.2 Feature Envy

Feature Envy is a code smell that occurs when a method is seemingly more interested in a class that is not its' parent class. For example, when a given method calls a large number of methods from the other class, indicating that this segment of code is better suited in that class, rather than its' actual parent.

Beck & Fowler suggest amending this smell by extracting the envious code into its' own method and moving it to the class it is envious of. [8]

It has been shown that Feature Envy can have a large impact on the energy consumption on software. Verdecchia et al. identified and refactored this code smell on three different Java applications, and found that for `JTrac` it reduced the energy consumption by 49.9% [4].

In order to test Feature envy, three seperate classes are made to help represent a user. The feature envy code smell version, *User*, uses "getters" from a seperate class *ContactInfo*. Meanwhile the non-smelly version has all functionality built into it's own class, *FullUser*, which contains it's own fields and data as opposed to relying on a seperate class for these. The implementations can be seen in Snippet 4.1.3 and Snippet 4.1.4.

```
class User
{
    private ContactInfo
    ↪ _contactInfo;

    User(ContactInfo contactInfo)
    {
        _contactInfo = contactInfo;
    }
    public string
    ↪ GetFullAddress(ContactInfo
    ↪ info)
    {
        return
        ↪ _contactInfo.streetName
        ↪ + ";" +
        ↪ _contactInfo.city + ","
        ↪ + _contactInfo.zip + ";"
        ↪ + _contactInfo.state +
        ↪ ";" +
        ↪ _contactInfo.country;
    }
}
```

Snippet 4.1.3: Feature envy code smell

```
class FullUser
{
    private string city = "Aalborg";
    private string state =
    ↪ "Jylland";
    private string streetName =
    ↪ "Frederik Bajers Vej";
    public string zip = "9000";
    public string country =
    ↪ "Danmark";

    public string GetFullAddress()
    {
        return streetName + ";" +
        ↪ city + ";" + zip + ";" +
        ↪ state + ";" + country;
    }

}
```

Snippet 4.1.4: Feature envy code smell, refactored

Both classes return a string containing an address. [2]

The results from the tests of the feature envy smell can be seen in Figure 4.2.

[2] CodeSmellExamples/Featureenvy.cs

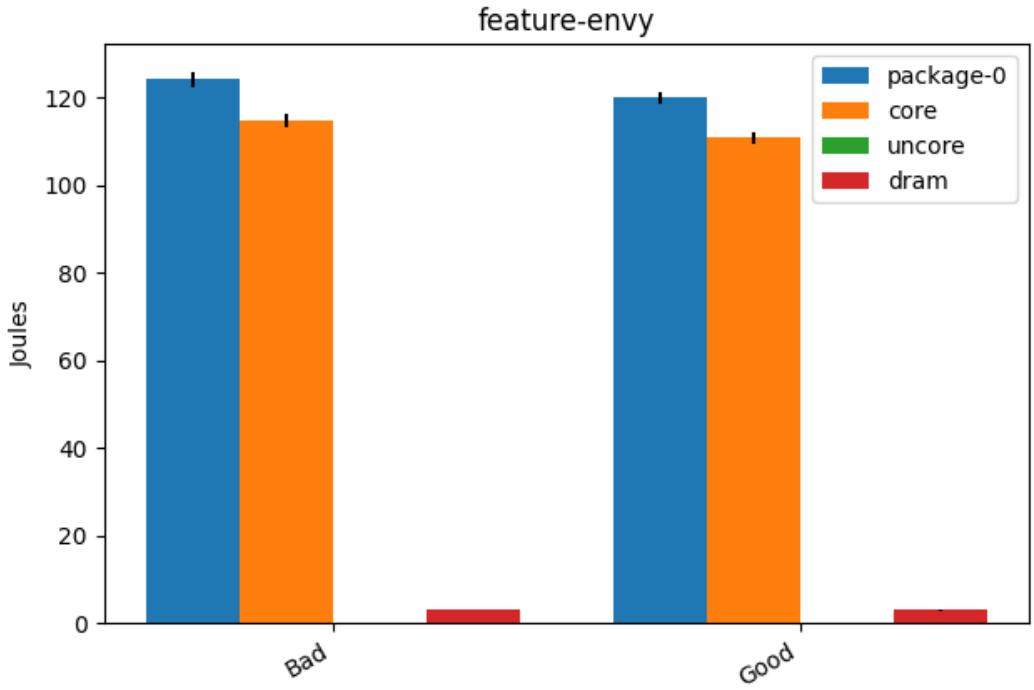

Figure 4.2: Feature envy code smell energy consumption

The bad variant of feature envy has an average energy consumption of 124.12 joules for a run, meanwhile the good variant uses an average of 119.98 joules per run, a reduction of 2.86%. Looking at the run time of the two the bad variant averages out at 6.64 seconds, while the good variant completes in about 6.45 seconds, about 3.34% faster. This is a much smaller energy reduction than the reported 49.9% mentioned in [4], however an energy reduction is nevertheless seen in the "good" variant of the implementation, albeit much smaller than what has been reported for fixing the same smell in Java applications.

Inspecting IL code

In order to assess the reason behind the small increase in energy consumption relative to the literature, the IL code is inspected. Snippets 4.1.5 and 4.1.6 lists the bad and good variants of field access in the code smell, respectively.

```
IL_000b: ldfld        class Smells.CodeSmellExamples.ContactInfo
↪  Smells.CodeSmellExamples.FeatureEnvyBad::_contactInfo
IL_0010: ldfld        string Smells.CodeSmellExamples.ContactInfo::streetName
IL_0015: stelem.ref
IL_0016: dup
```

Snippet 4.1.5: Field access in the bad variant of Feature Envy

```
IL_000b: ldfld        string Smells.CodeSmellExamples.FeatureEnvyGood::streetName
IL_0010: stelem.ref
IL_0011: dup
```

Snippet 4.1.6: Field access in the good variant of Feature Envy

As seen in the snippets the instructions performed are highly similar, in fact there is only one difference: the bad variant performs one additional lookup on the evaluation stack. This small difference between the two implementations would explain the equally small difference in energy consumption.

However, this does not explain the large difference between what was found here, and the 49.9% found by Verdecchia et al.. In their paper they argue that the large difference in energy consumption before and after refactoring is likely due to the application in question being very old, in fact their results on two other, newer applications showed results highly similar to what was found in this report [4]. In addition to this, the examples of feature envy found in their software may be more comprehensive than what was used in this report, perhaps containing more field references to more complex objects, further increasing the impact of the code smell.

4.1.3 Dead Local Store

Dead Local Stores appear when some local variable is assigned to some value, which is then never used. Vetro et al. found that refactoring and fixing this smell in C++, gives an energy consumption reduction of less than 1% [1].

To benchmark a dead local store smell, two very identical looking classes are created with the same effective functionality. In this specific case, each class finds the area of a circle. The code smell variant of the class has 3 dead local variables stored additionally, this just means variables that are declared in the function body, but never used.

The aforementioned paper does not contain code listings save for one example of another smell, making a 1-to-1 replication of the C++ smell impossible, as we

do not know the contents of it. Given this, a new set of code snippets were created to test the dead local store code smell.

Both classes return the resulting area of a circle with the radius given through the parameter. The implementations can be seen in Snippet 4.1.7 and Snippet 4.1.8. [3]

```
1  public override double
   ↪  DeadLocalStore(double radius)
2  {
3      double pi = 3.14;
4      double notPi = 4.13;
5      int cousinsAgeInMonths = 146;
6      string niceGreeting = "Hellow
       ↪  World";
7
8      return (pi * Math.Pow(2,
       ↪  radius));
9  }
```

Snippet 4.1.7: Dead Local Store Code Smell

```
1  public override double
   ↪  DeadLocalStore(double radius)
2  {
3      double pi = 3.14;
4
5      return (pi * Math.Pow(2,
       ↪  radius));
6  }
```

Snippet 4.1.8: Dead Local Store Code Smell, refactored

The total energy consumption of the tested variants of the dead local store code smell can be seen in Figure 4.3.

[3] CodeSmellExamples/DeadLocalStore.cs

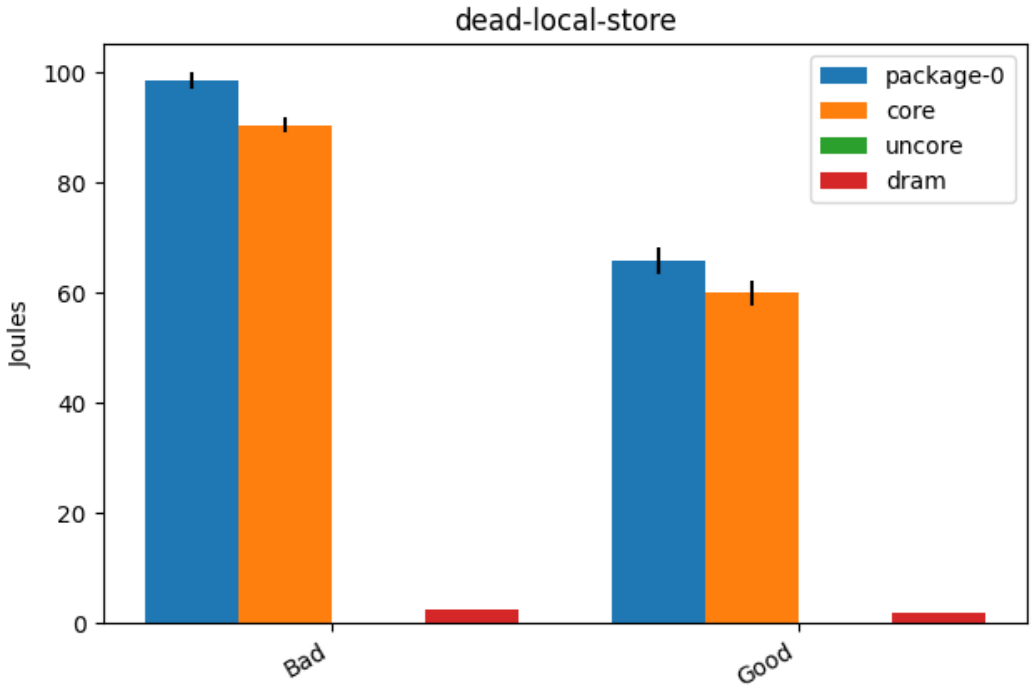

Figure 4.3: Dead local store code smell energy consumption

The bad variant of the dead local store smell uses an average of 98.62 joules per run, while the non smell variant averages out at 65.79 joules per run, which equates to a reduction of the energy consumption of about 33.29% moving to the good variant. In regards to time, the smell variant averages out at a run time of 5.62 seconds, meanwhile the non-smell uses about 3.94 seconds, a reduction of about 29.89%. Compared to the less than 1% performance gain found in C++, this is a surprisingly large energy saving. Given that we do not use a semantically equivalent implementation of the dead local store smell as the one used in the C++ literature, a direct comparison between the two should not be made, however the energy performance gap is large enough that we feel it warrants further analysis.

Inspecting IL code

In order to assess whether the implementation behaves as expected, the IL code has been inspected. Snippet 4.1.9 lists the IL code for the unused instantiations present in the bad varial of Dead Local Store.

```
IL_0001: ldc.r8       3.14
IL_000a: stloc.0      // pi

IL_000b: ldc.r8       4.13
IL_0014: stloc.1      // notPi

IL_0015: ldc.i4       146 // 0x00000092
IL_001a: stloc.2      // cousinsAgeInMonths

IL_001b: ldstr        "Hellow World"
IL_0020: stloc.3      // niceGreeting
```

Snippet 4.1.9: The unused variable instantiations in the bad variant

As seen in the snippet, the variables are correctly allocated. This shows that instantiating variables that are never used does indeed increase the energy consumption, however it is still unknown why the increase is so large.

4.1.4 Long Method

As the name suggests, Long Method is a code smell that occurs when a method grows too large. For a long time long procedures have been known to inhibit the readability of code, and thereby also the maintainability.

Beck & Fowler suggest amending this smell by dividing such a method into smaller methods. [8]

Much like Feature Envy, Verdecchia et al. has shown that Long Method can have a large impact on the energy consumption. They found that for `JTrac` the energy consumption was reduced by 49.9% [4].

To benchmark the long method smell, two similar classes are created, each containing a method `Compute`. In both cases the method will perform various mutations on an array. Firstly an array of 10 random numbers is created, then the array is sorted using selection sort, the sum of the array is then calculated, followed by an inversion of the array, and finally each element in the array is multiplied by some number. For the bad version all of these operations are computed in the same method, while in the good version each of the operations are moved to their own method, which is then called in the `Compute` method. The implementation is included in Appendix A.1 [4]

The comparative energy consumption of the tested variants of the long method code smell can be seen in Figure 4.4.

[4]CodeSmellExamples/LongMethod.cs

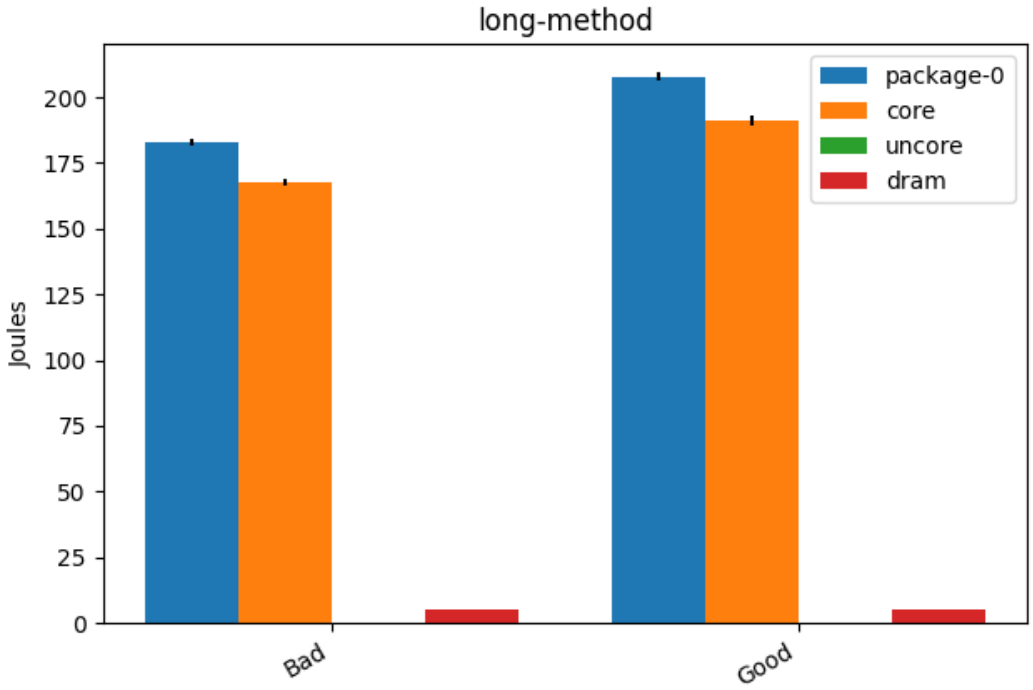

Figure 4.4: Energy consumption of long method code smell

The bad variant of the long method smell consumes on average 183.12 joules, whereas the refactored version consumes 207.98 joules, meaning that refactoring this code smell sees an increase in energy consumption of about 13.58%. This is interestingly in contrast to what was found in the literature, and it may be caused by the increased number of method calls that happen when the snippets are extracted to individual methods. For runtime, the bad variant spent 10.74 seconds executing, whereas the refactored version spent 11.81 seconds, this being an increase of about 9.96%.

Analyzing results

Long Method unexpectedly showed an increase in energy consumption once refactored, contrary to what was found in the literature. Similarly to Feature Envy, this can likely be explained by `JTrac` used by Verdecchia et al. being very old software and thereby having a larger impact on the energy consumption [4].

Another possible factor is the difference in languages. `JTrac` was written in Java, whereas the implementation used in this report was written in C#. This may indicate that the greater amount of method calls performed in the refactored

version has a larger impact on the energy consumption in C# than it does in Java.

4.1.5 Parameter by Value

Passing a parameter to a function by value is generally a heavy operation compared to e.g., passing by reference. The value is copied to the function, thereby requiring more computation time. In Vetro et al.'s exploratory study they found that passing by value may in fact also have an impact on the energy consumption. Keeping in mind that their experiments were microbenchmarks, they did find a statistical significant difference in energy consumption once passing by value was refactored [1].

To benchmark parameter by value, two classes are made which both call a function *Compute*. The first class declares two integer variables *a* and *b*, before passing them to the function by reference. The other, Smell version, passes what would normally be the values of a and b directly into the function by value. The implementations can be seen in Snippet 4.1.10 and Snippet 4.1.11.

```
1 public override void
  ↪ ParameterByValue()
2 {
3     int a = 100;
4     int b = 50;
5
6     Compute(a, b);
7 }
```

Snippet 4.1.10: Parameter by Value Code Smell

```
1 public override void
  ↪ ParameterByValue()
2 {
3     int a = 100;
4     int b = 50;
5
6     Compute(ref a, ref b);
7 }
```

Snippet 4.1.11: Parameter by Value Code Smell, refactored

The `Compute` function, makes a declaration of an integer c, runs an if statement and does some standard arithmetics. The implementation of the `Compute` function can be seen in Snippet 4.1.12. [5]

[5] CodeSmellExamples/ParameterByValue.cs

```
1  public int Compute(int a, int b)
2  {
3      int c = a * b;
4
5      if (a > b) c = c + b;
6
7      b = b * 2;
8
9      return c;
10 }
```

Snippet 4.1.12: The `Compute` function called in the Parameter by Value smell. A second variant is also included with a different signature: the two input parameters are `ref`s.

The comparative energy consumption of the tested variants of the parameter by value code smell can be seen in Figure 4.5.

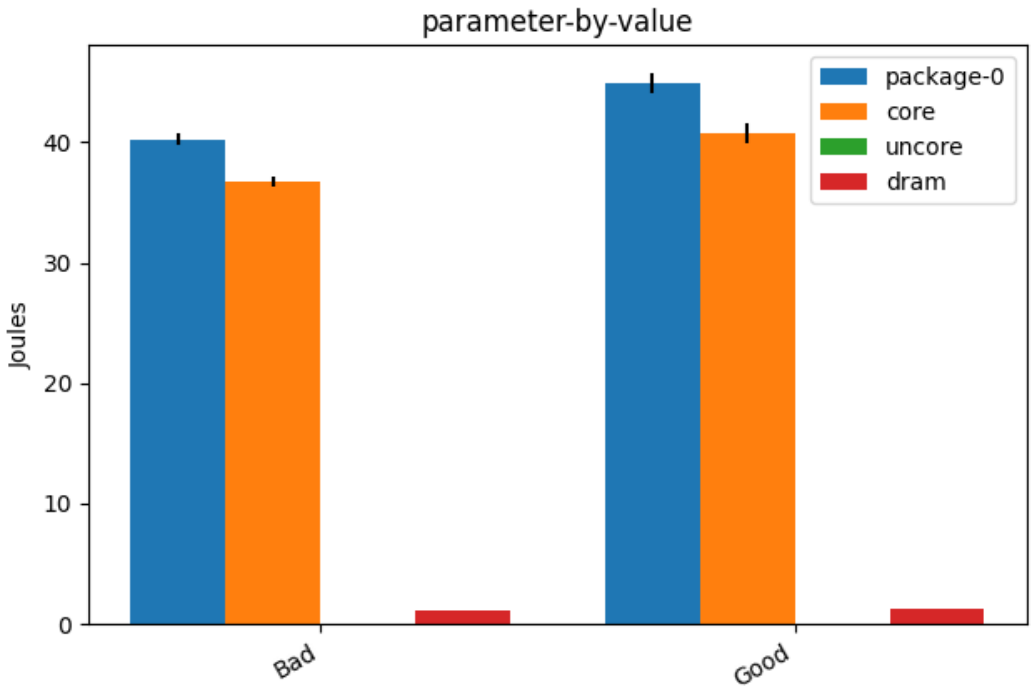

Figure 4.5: Energy consumption of parameter by value code smell

The bad variant of the parameter by value smell consumes on average 39.66

joules, whereas the refactored version consumes 45.42 joules, meaning that contrary to the literature, refactoring this smell sees an increase in energy consumption of 11.66% as opposed to the expected decrease. In addition to this, the bad variant spent 2.35 seconds executing, whereas the refactored version spent 2.65 seconds, a similar increase of 12.71%.

Inspecting assembly code

As this code smell showed different results than that of the current literature, analyzing the behavior of the generated code is particularly interesting. Looking at the assembly code shows that only one of the `ParameterByValue` methods is generated, meaning that the compiler likely reuses the one method that does have generated assembly code. This is quite interesting as, while the two methods and the two `Compute` methods are nearly identical, the `Compute` methods have different signatures, one accepting `int` values and one accepting `int` value references.

However, looking at the code for the two `Compute` methods, we see that these are actually different.

```
Smells.CodeSmellExamples.ParameterByValueBa
↪ Int32)
    L0000: mov eax, [esp+4]
    L0004: mov ecx, edx
    L0006: imul ecx, eax
    L0009: cmp edx, eax
    L000b: jle short L000f
    L000d: add ecx, eax
    L000f: mov eax, ecx
    L0011: ret 4
```

Snippet 4.1.13: Assembly code for the bad variant of Parameter By Value

```
Smells.CodeSmellExamples.ParameterByValueBase.Compute(Int32
↪ ByRef, Int32 ByRef)
    L0000: push esi
    L0001: mov eax, [esp+8]
    L0005: mov edx, [edx]
    L0007: mov ecx, [eax]
    L0009: mov esi, edx
    L000b: imul esi, ecx
    L000e: cmp edx, ecx
    L0010: jle short L0014
    L0012: add esi, ecx
    L0014: add ecx, ecx
    L0016: mov [eax], ecx
    L0018: mov eax, esi
    L001a: pop esi
    L001b: ret 4
```

Snippet 4.1.14: Assembly code for the good variant of Parameter By Value

Snippet 4.1.13 shows the assembly code for the bad variant of the `Compute` method, and Snippet 4.1.14 shows the good variant. It is clear that in the good variant almost twice as many instructions are executed compared to the bad variant, and in addition to this, the good variant performs twice as many moves. This increase in instructions is likely the cause of the unexpected increase in energy

consumption after refactoring the smell.

4.1.6 Self Assignment

Self assignments occurs when a variable is assigned to itself, e.g., `x = x;`. While this may seem redundant, it can occur when some object has two or more available references at the same time. In most programming languages this is handled internally, however in languages such as C++ it is not. Because of this, Vetro et al. examined the impact these self assignments have on energy consumption, and found that refactoring indeed had a statistically significant difference [1].

To benchmark the self assignemnt smell, two classes are created, each contains four strings which are which can be returned concatanated with one another, by calling the method within the class, this happens in both classes. The only difference between the classes is that one has a number of self assignments occur before the return statement while the other simply goes straight to return the string. [6]

```
1  public override string
   ↪  SelfAssignment()
2  {
3      string a = "";
4      string x = "Played College Ball
       ↪  you know";
5      string y = "Could have gone pro
       ↪  if I hadn't joined the
       ↪  navy";
6      string z = "Nanomachines Son";
7
8      a = z;
9      x = x;
10     y = y;
11     z = z;
12
13     return a + x + y + z;
14 }
```

Snippet 4.1.15: Self Assignment Code Smell

```
1  public override string
   ↪  SelfAssignment()
2  {
3      string a = "";
4      string x = "Played College Ball
       ↪  you know ";
5      string y = "Could have gone pro
       ↪  if I hadn't joined the navy
       ↪  ";
6      string z = "Nanomachines Son ";
7
8
9      a = z;
10
11     return a + x + y + z;
12 }
```

Snippet 4.1.16: Self Assignment Code Smell, refactored

The comparative energy consumption of the tested variants of the self assignment code smell can be seen in Figure 4.6.

[6] CodeSmellExamples/SelfAssignment.cs

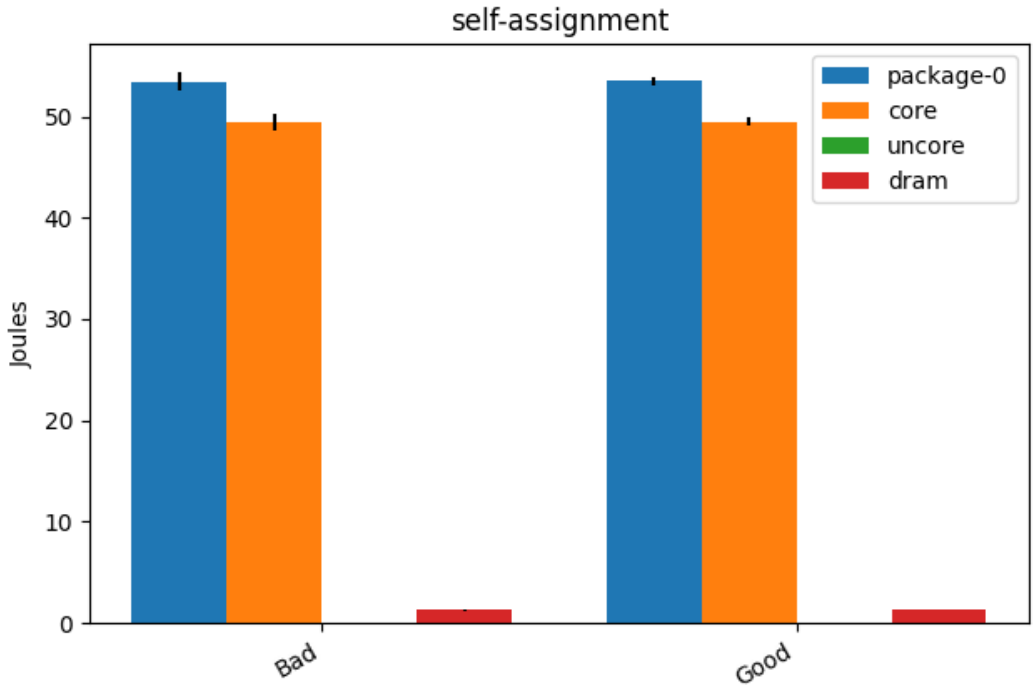

Figure 4.6: Energy consumption of self assignment code smell

The bad variant of the self-assignment implementation uses an average of 53.46 joules per run, compared to the good variants 53.52 this is an increase of about 0.11% in energy consumption, comparing this to the run times of the two variants at 2.79 seconds for the bad variant and 2.77 seconds for the good, we see that this is about a 0.72% decrease between the two variants. This effectively means that removing these redundant assignments decreases the time spent executing, however it does not reduce the energy consumption.

Inspecting assembly code

Because the results of this smell showed a negligible difference between the bad and good variant, it is particularly interesting to inspect the IL and assembly code generated, in order to assess whether the implementation behaves as expected.

Inspecting the IL code shows that the three lines with self assignment in the bad variant are in fact generated, however, inspecting the assembly code shows different results. In the assembly the generated code is identical between the two variants, namely what is shown in Snippet 4.1.17.

```
1 L0000: mov edx, [0x9367b34]
2 L0006: mov ecx, [0x9367b38]
3 L000c: mov eax, [0x9367b44]
4 L0012: push ecx
5 L0013: push eax
6 L0014: mov ecx, eax
7 L0016: call System.String.Concat(System.String, System.String, System.String,
  ↪ System.String)
8 L001b: ret
```

Snippet 4.1.17: Assembly code generated for both the bad and good variants of Self Assignment

Because the code generated is identical, it is safe to assume that redundant self assignment statements are caught and removed by the compiler at compile-time. As such, self assignments do not have a negative impact on the energy consumption in C#.

4.1.7 Repeated Conditionals

As the name suggests, Repeated Conditionals occur when some boolean condition is evaluated more than once within in the same scope. For example, some `if`-clause in a function evaluates the condition `x == 0`, which is then evaluated again later in the same function. Vetro et al. found that refactoring this smell had a statistical significant difference, however being the smallest difference of the examined smells [1].

To benchmark the repeated conditionals smell, two classes are created, each contains a method which declares three integer variables, after this, a conditional check is then made after which two pieces of light arithmetics are performed. In the smell version, a separate conditional check exists for both arithmetic operations. These if statements check for the same thing however, thus simulating a repeated conditional. The implementations can be seen in Snippet 4.1.18 and Snippet 4.1.19. [7]

[7] CodeSmellExamples/RepeatedConditionals.cs

```
public override void
↪ RepeatedConditionals()
{
    int a = 100;
    int b = 200;
    int c = 20;
    int d = 0;

    if (a * c > b) {
        d = b + a;
    }

    if (a * c > b) {
        b = d * c;
    }

    d = a * c + b;
}
```

Snippet 4.1.18: Repeated Conditionals Code Smell

```
public override void
↪ RepeatedConditionals()
{
    int a = 100;
    int b = 200;
    int c = 20;
    int d = 0;

    if (a * c > b) {
        d = b + a;
        b = d * c;
    }

    d = a * c + b;
}
```

Snippet 4.1.19: Repeated Conditionals Code Smell, refactored

The comparative energy consumption of the tested variants of the repeated conditionals code smell can be seen in Figure 4.7.

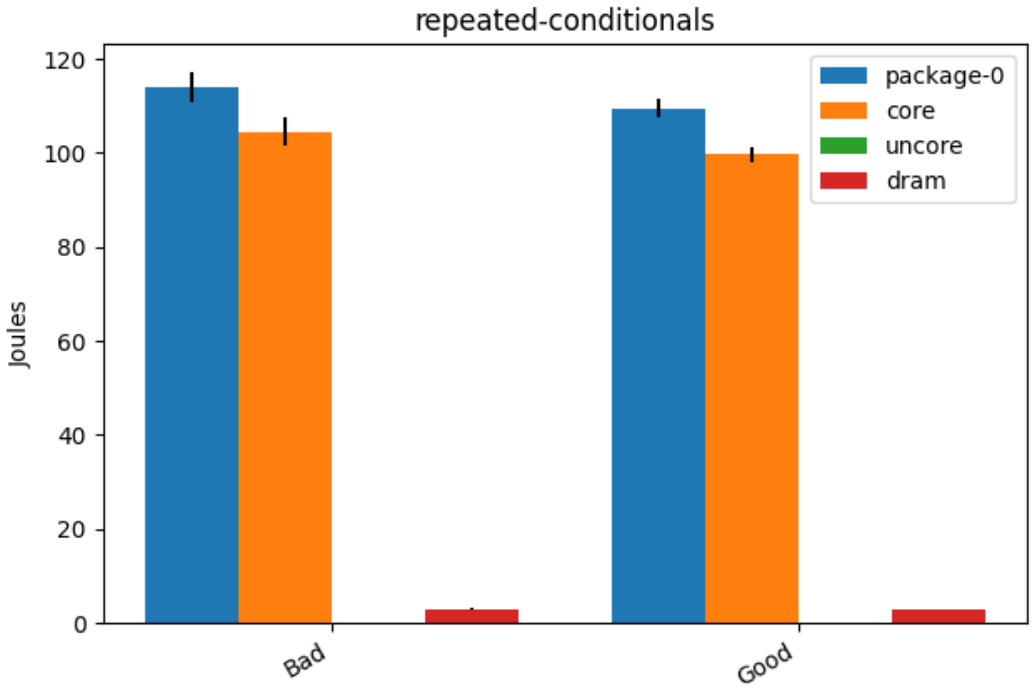

Figure 4.7: Energy consumption of the repeated conditionals code smell

The bad variant of repeated conditionals uses an average of 114.01 joules, meanwhile the good variant uses 109.43 joules. A rather significant difference of about 4.02% on the energy consumption. Looking at time spent, the bad variant spent 6.63 seconds executing, whereas the refactored version spent 6.43 seconds, a similar difference of about 3.02%.

Inspecting IL code

Repeated conditionals shows a small decrease in energy consumption once refactored. Snippet 4.1.20 shows the IL code of a singular `if` statement from the code smell implementation.

```
1 IL_000d: ldloc.0      // a
2 IL_000e: ldloc.2      // c
3 IL_000f: mul
4 IL_0010: ldloc.1      // b
5 IL_0011: cgt
6 IL_0013: stloc.s      V_4
7
8 IL_0015: ldloc.s      V_4
9 IL_0017: brfalse.s    IL_0023
```

Snippet 4.1.20: IL code for the `if` statement in both the bad and good variants of Repeated Conditionals.

The snippet shows the computation done in the evaluation, followed by the comparison on line 5. In the good variant, this code is generated once, whereas in the bad variant it is generated twice, and this is the cause behind the increase in energy consumption in the bad variant.

While the difference is rather small, the code smell implementation is also rather simple. In a larger system with more, or more complex, repeated conditionals, this smell may have a larger effect.

4.1.8 Non Short-Circuit

Using non-short circuiting boolean operators (i.e., & and |) will naturally, in some cases, produce more executed instructions if the left-hand side of the expression does not satisfy the condition, as opposed to short circuiting operators (i.e., && and ||). Vetro et al. found that refactoring non-short circuiting operators to short circuiting created a statistical significant reduction in energy consumption [1].

A listing in the literature shows the code snippet used to benchmark the non short-circuit smell, which gives a great opportunity for creating a semantically equivalent smell and having a 1-to-1 comparison between the two benchmarks. The C++ implementation is as follows:

```
void NonShortCircuit_With(){
    int count = 0;
    int total = 345;
    if ( count > 0 & total / count > 80 )
        count=0;
}
void NonShortCircuit_Without() {
    int count = 0;
    int total = 345;
    if ( count > 0 && total / count > 80 )
        count=0;
}
```

Snippet 4.1.21: Non-Short Circuit Code Smell

Sadly however, upon rewriting the smell to C we encounter an error, namely that in the version without short circuiting a division is made with 0, causing a run-time error which terminates the program. Given that due to this error we cannot achieve semantic equivalence, it was decided to create a new benchmark instead.

In this benchmark, two classes are created, each class contains a method which first declares 3 integer variables and then makes two conditional checks before performing some arithmetics if the conditionals are fulfilled. The conditionals each have a boolean logical operator in them, and are set up such that they could theoretically be short circuited after the first part of the condition has been read. The smell version uses a non short-circuit check for these, meaning both sides will be evaluated once regardless of an impossible outcome after reading the first side. The non smell version, has the same conditional checks, however allows for short circuiting in the evaluation. The implementations can be seen in Snippet 4.1.22 and Snippet 4.1.23. [8]

[8] CodeSmellExamples/ShortCircuit.cs

```
public override void ShortCircuit()
{
    int a = 100;
    int b = 1000;
    int c = 5;

    if (b > a | a > c) a = a * c;

    if (a > b & b == 1000) a = a *
    ↪  c;
}
```

Snippet 4.1.22: Non-Short Circuit Code Smell

```
public override void ShortCircuit()
{
    int a = 100;
    int b = 1000;
    int c = 5;

    if (b > a || a > c) a = a * c;

    if (a > b && b == 1000) a = a *
    ↪  c;
}
```

Snippet 4.1.23: Non-Short Circuit Code Smell, refactored

The comparative energy consumption of the tested variants of the non short-circuit code smell can be seen in Figure 4.8.

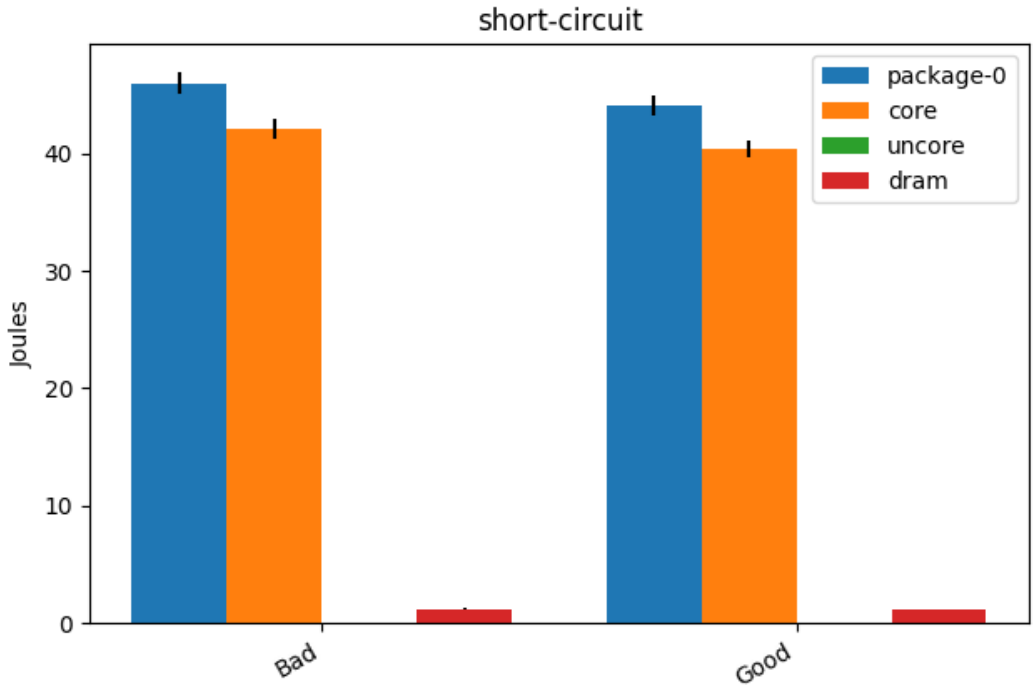

Figure 4.8: Energy consumption of non short-circuit code smell

The smell variant of non short-circuit uses an average of 46.03 joules per run, meanwhile the non-smell variant uses an average of 44.08 joules, a reduction of 4.49%. In regards to execution time the smell variant completes execution in 2.67 seconds while the non smell variant's execution time averages out to 2.55 seconds, a 4.24% decrease in run time between the two. While this is one of the smaller differences, it is a very small and isolated example and it may have a bigger impact on larger systems with more, and more complex, conditional statements where non-short circuiting boolean operators are used. This will be further analyzed in section 4.2

Inspecting IL code

As expected this code smell showed a small decrease in energy consumption after being refactored. As such, it is interesting to inspect the IL code in order to see if the implementation behaves as expected.

```
1 IL_000c: ldloc.1      // b
2 IL_000d: ldloc.0      // a
3 IL_000e: cgt
4 IL_0010: ldloc.0      // a
5 IL_0011: ldloc.2      // c
6 IL_0012: cgt
7 IL_0014: or
8 IL_0015: stloc.3      // V_3
```

Snippet 4.1.24: IL code for the bad variant of Non-Short Circuit

```
1 IL_000c: ldloc.1      // b
2 IL_000d: ldloc.0      // a
3 IL_000e: bgt.s        IL_0016
4 IL_0010: ldloc.0      // a
5 IL_0011: ldloc.2      // c
6 IL_0012: cgt
7 IL_0014: br.s         IL_0017
8 IL_0016: ldc.i4.1
9 IL_0017: stloc.3      // V_3
```

Snippet 4.1.25: IL code for the good variant of Non-Short Circuit

Snippets 4.1.24 and 4.1.25 lists the IL code for the logical or evaluation for the bad and good variants of the code smell, respectively. Looking at the good variant, line 3 performs the instruction `bgt.s` which will perform a jump to the specified instruction if the condition evaluates to true, in this case it will jump to line 8. This means that if the left hand side of the or condition is true, the right hand side will be skipped, correctly performing a short circuit, unlike the bad variant.

4.1.9 Type Checking

N. Tsantalis et al. identify conditional Type Checking as a code smell. An example of this is executing different code statements dependent on conditional statements based on the type of a variable, e.g., through a `switch` statement or an `if-else` clause. They suggest amending this smell through one of two different meth-

ods: if the attribute represents some state e.g., a type field on a class, then use the *State/Strategy* pattern; if the type check is done through **RTTI** (RunTime Type Identification), then utilize *polymorphism*. [3]

Verdecchia et al. has shown that Type Checking can have some impact on the energy consumption. They found that for `JTrac`, refactoring this code smell reduced the energy consumption by 4.9% [4].

In the paper, small examples of the kind of type checking smell mentioned can be found, these will be used as the baseline for the implementations created in this paper to test the type checking smells.

The implementations will and results for type field and RTTI will be explained separately in the following two sections. [9]

Type Checking with Type Fields

Type checking with type fields refer to storing a type identifier on class level and comparing this to the available types. For example, an employee may have a position in a company, for example an engineer or a salesman. In Snippet 4.1.26 this identifier is stored on an employee object as a const integer and is compared against the available identifiers, `ENGINEER`, `SALESMAN`, and `DIRECTOR`. Once the correct position has been found, a string identifier is returned. For the refactored version in Snippet 4.1.27, this string is retrieved from an inherited object denoting one of these positions.

```
1  public override string getType()
2  {
3      if (this.obj.getTypeField() ==
       ↪  ENGINEER) return "Engineer";
4      else if (this.obj.getTypeField()
       ↪  == SALESMAN) return
       ↪  "Salesman";
5      else if (this.obj.getTypeField()
       ↪  == DIRECTOR) return
       ↪  "Director";
6      else return "Error";
7  }
```

Snippet 4.1.26: Type Checking with Type Fields Code Smell

```
1  private Employee state;
2
3  public override string getType()
4  {
5      return state.getTypeString();
6  }
```

Snippet 4.1.27: Type Checking with Type Fields Code Smell, refactored

The comparative energy consumption of the tested variants of type checking with type fields can be seen in Figure 4.9.

[9] CodeSmellExamples/TypeChecking.cs

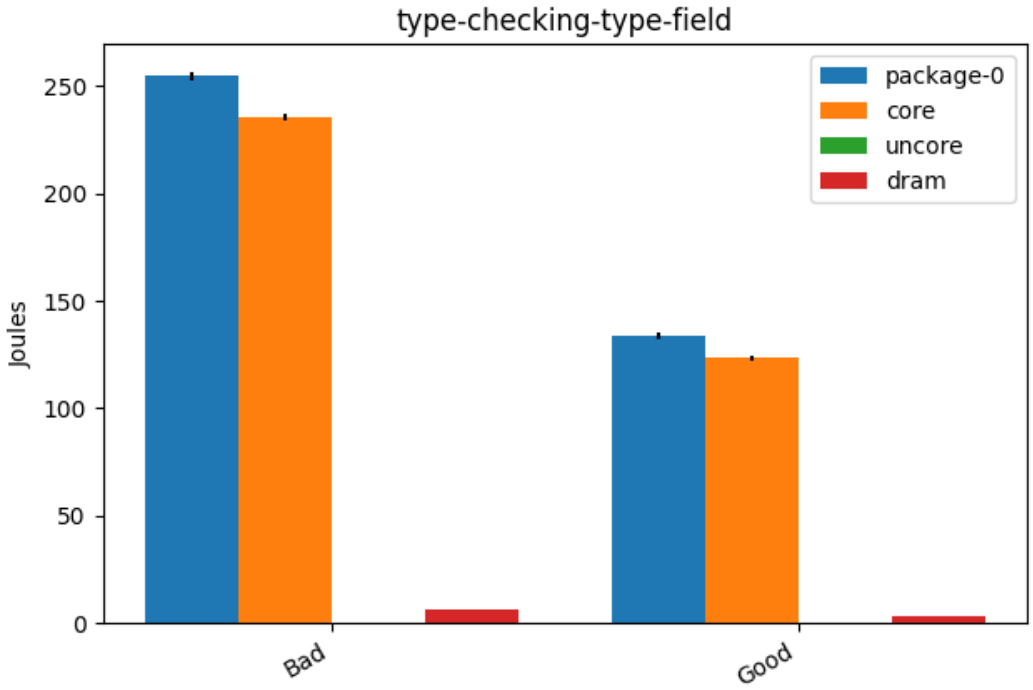

Figure 4.9: Energy consumption of the type checking with type fields code smell

The bad variant of type checking with type fields uses on average 254.79 joules per run, whereas the refactored version uses only 133.73 joules per run. This is an astounding 45.09% reduction in energy consumption on the refactored version. As for runtime, the bad variant spent 13.35 seconds executing, whereas the refactored version spent 7.33 seconds, an equally astounding reduction in runtime of 47.51%. While a majority of the reduction in energy consumption likely stems from the reduced runtime, and that several conditional checks are executed, it is clear that the refactored version is a far better option with regards to energy consumption.

Type Checking with RTTI

Type checking with RunTime Type Identification (RTTI) refers to looking at the built-in type of some object, i.e., through `typeof()` in C# or `instanceof` in Java. In the bad variant, as seen in Snippet 4.1.28, the objects type is accessed through the built-in `GetType()` method on an object, and is compared against the inherited classes denoting the different positions. Much like the type field implementation, this will return a string identifier. The same refactored version as with type fields is used for this implementation.

```
public override string getType()
{
    if (this.obj.GetType() ==
    ↪ typeof(Engineer)) return
    ↪ "Engineer";
    else if (this.obj.GetType() ==
    ↪ typeof(Salesman)) return
    ↪ "Salesman";
    else if (this.obj.GetType() ==
    ↪ typeof(Director)) return
    ↪ "Director";
    else return "Error";
}
```

Snippet 4.1.28: Type Checking with RTTI Code Smell

```
private Employee state;

public override string getType()
{
    return state.getTypeString();
}
```

Snippet 4.1.29: Type Checking with RTTI Code Smell, refactored

The comparative energy consumption of the tested variants of type checking with RTTI can be seen in Figure 4.10.

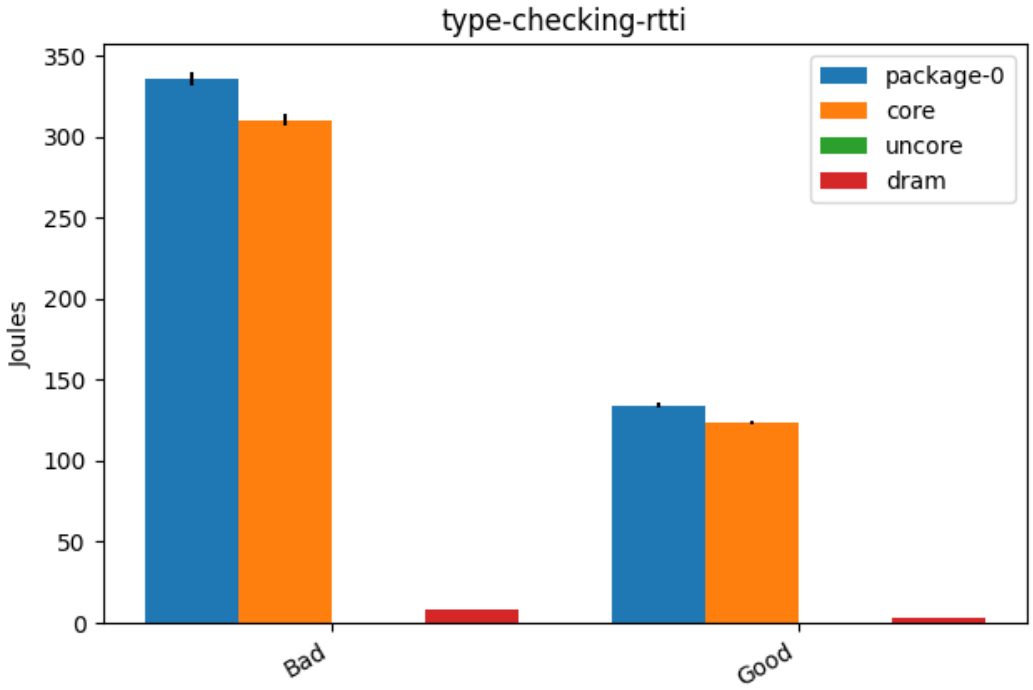

Figure 4.10: Energy consumption of the type checking with RTTI code smell

The bad variant of type checking with RTTI uses on average 335.83 joules per run, whereas the refactored version uses only 134.34 joules per run. Much like type fields this is an even more astounding reduction of 60.00% in energy consumption. As for runtime, the bad variant spent 17.76 seconds executing, whereas the refactored version only spent 7.33 seconds, yielding a reduction in runtime of about 58.73%. Much like type fields, RTTI spends far more energy than when using polymorphism to check the type of an object, however with the same considerations with regards to runtime and conditional checks in mind. However, comparing the bad versions of type field and RTTI it is clear that RTTI is the worse option of the two. In fact, RTTI consumes about 24.13% more energy than using type fields.

Inspecting IL code

In order to analyze not only the large difference between the bad variants and the refactored version, but also the relatively large difference between the bad variants, the IL code of the smell is examined.

```
1 IL_0001: ldarg.0      // this
2 IL_0002: call         instance class Smells.CodeSmellExamples.Employee
  ↪  Smells.CodeSmellExamples.TypeCheckingTypeFieldBad::get_obj()
3 IL_0007: callvirt     instance int32
  ↪  Smells.CodeSmellExamples.Employee::getTypeField()
4 IL_000c: ldarg.0      // this
5 IL_000d: ldfld        int32
  ↪  Smells.CodeSmellExamples.TypeCheckingTypeFieldBad::ENGINEER
6 IL_0012: ceq
7 IL_0014: stloc.0      // V_0
```

Snippet 4.1.30: IL code for the bad variant of Type Checking with Type Field

Snippet 4.1.30 shows the IL code for the bad variant of Type Checking with Type Field. Line 2 is a call to the internal getter for the object whose type field is being evaluated, and line 3 is a call to the inherited getter for the type field. Line 5 is a field lookup on the aforementioned object while it is on the evaluation stack. Line 6 is an equality check between the type field and one of the available types, in this case `ENGINEER`. This code is generated three times, one for each `if-else` clause, and on an arbitrary run at least one of these are evaluated.

```
1 IL_0001: ldarg.0      // this
2 IL_0002: call         instance class Smells.CodeSmellExamples.Employee
  ↪  Smells.CodeSmellExamples.TypeCheckingRTTIBad::get_obj()
3 IL_0007: callvirt     instance class [System.Runtime]System.Type
  ↪  [System.Runtime]System.Object::GetType()
4 IL_000c: ldtoken      Smells.CodeSmellExamples.Engineer
5 IL_0011: call         class [System.Runtime]System.Type
  ↪  [System.Runtime]System.Type::GetTypeFromHandle(valuetype
  ↪  [System.Runtime]System.RuntimeTypeHandle)
6 IL_0016: call         bool [System.Runtime]System.Type::op_Equality(class
  ↪  [System.Runtime]System.Type, class [System.Runtime]System.Type)
7 IL_001b: stloc.0      // V_0
```

Snippet 4.1.31: IL code for the bad variant of Type Checking with RTTI

Snippet 4.1.31 shows the IL code for the bad variant of Type Checking with RTTI. Line 2 and 3 perform the same actions as with type fields, except line 3 retrieving the object type instead. On line 4 a metadata token for the class `Engineer` is converted to its runtime representation. On line 5 the builtin function `typeof()` is called on the `Engineer`, and on line 6 the two retrieved types are compared for equality through an additional function call. Like for type fields, this code is

generated three times.

```
1 IL_0001: ldarg.0      // this
2 IL_0002: call         instance class Smells.CodeSmellExamples.Employee
  ↪ Smells.CodeSmellExamples.TypeCheckingGood::get_obj()
3 IL_0007: callvirt     instance string
  ↪ Smells.CodeSmellExamples.Employee::getTypeString()
4 IL_000c: stloc.0      // V_0
5 IL_000d: br.s         IL_000f
```

Snippet 4.1.32: IL code for the refactored variant of Type Checking

Snippet 4.1.32 shows the IL code for the refactored version of Type Checking. Like the other two snippets, line two is a call to the internal getter for the object. Line 3 is a call to an inherited method retrieving the same information as the `if-else` clauses in the bad variants. As there is no `if-else` construct in this variant, the code is generated and executed only once.

Comparing the bad variants to the refactored variant, it is clear that far more code is generated in the bad variants. This is likely the reason behind a large portion of the difference in energy consumption. However, it is also clear that more operations are performed through method calls and boolean operations.

Comparing the two bad variants, the snippets show that Type Checking with RTTI perform two additional method calls in order to evaluate a single conditional statement. In fact using RTTI, in this case, performs twice as many method calls than Type Field, indicating that these method calls are the primary cause of the increase in energy consumption between the two bad variants.

4.1.10 Dead Code

Dead code occurs when some code is implemented and never used. This causes the code to be purposelessly loaded into memory and thereby may increase the energy consumption of software. Most often this smell is caught by compiler optimizations, however cases where the code is executed and the results are not used can also occur with the same effect. [2]

In order to benchmark this smell two classes are created, one including the smell, and one where the smell has been fixed. Both classes include a `Run` method that performs the Pythagoras theorem. The smelly class also includes a number of additional methods that perform various mathematical formulas that are never used. In the class where the smell is fixed, these methods are removed. The implementations can be seen in Snippet 4.1.33 and Snippet 4.1.34. [10]

[10] CodeSmellExamples/DeadCode.cs

```
public override void Run()
{
    int a = 5;
    int b = 10;

    double c = Math.Sqrt(Math.Pow(2,
    ↪  a) + Math.Pow(2, b));

    if (a > b)
    {
        int fib = Fibonacci(10);
        int fac = Factorial(5);
        bool prime = isPrime(400);
        double sqArea =
        ↪  SquareArea(40);
        double cArea =
        ↪  CircleArea(10);
        double cylVol =
        ↪  VolumeCylinder(10, 40);
    }
}

public int Fibonacci(int n){ ... }
public int Factorial(int n){ ... }
public bool isPrime(int n){ ... }
public double SquareArea(int side){
↪  ... }
public double CircleArea(int r){ ...
↪  }
public double VolumeCylinder(int r,
↪  int h){ ... }
```

Snippet 4.1.33: Dead Code Code Smell

```
public override void Run()
{
    int a = 5;
    int b = 10;

    double c = Math.Sqrt(Math.Pow(2,
    ↪  a) + Math.Pow(2, b));

    // ensure the evaluation is done
    ↪  here as well
    if (a > b) c = c * 2;
}
```

Snippet 4.1.34: Dead Code Code Smell, refactored

The comparative energy consumption of the tested variants of dead code can be seen in Figure 4.11.

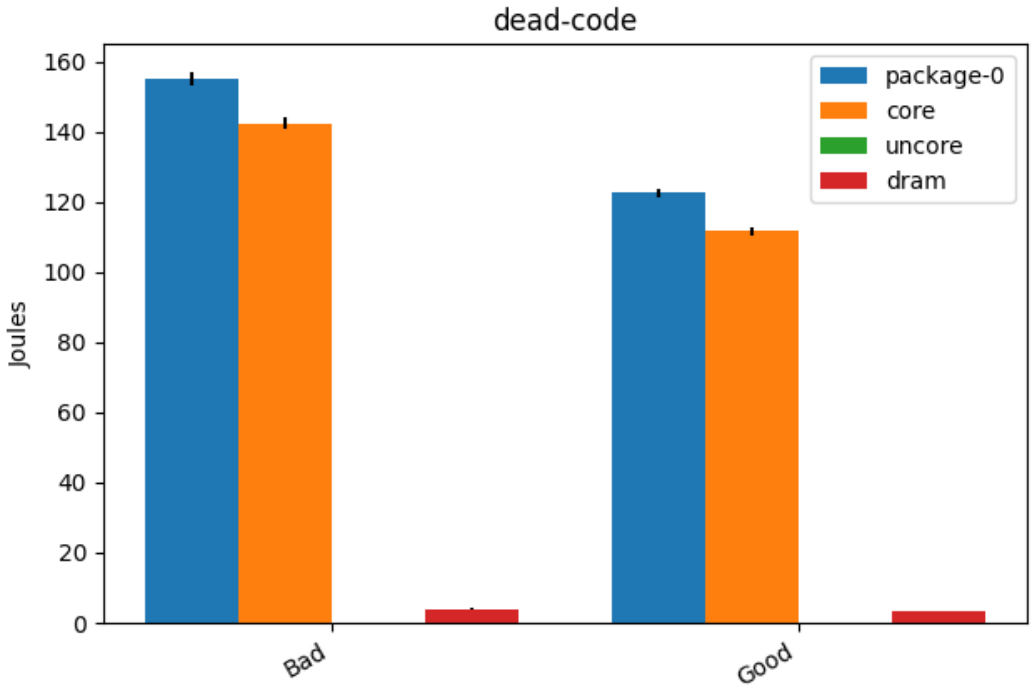

Figure 4.11: Energy consumption of the dead code code smell

The bad variant of dead code uses an average of 155.25 joules per run, whereas the good variant uses 122.64. This is a decrease in energy consumption of about 21.00% in the refactored version. As for time spent, the bad variant spent 9.01 seconds executing, whereas the good variant spent 7.57 seconds, a decrease in time spent of about 15.98%. Interestingly the energy consumption and time spent are not as closely tied as many of the other code smells, indicating that loading unused code into memory may have a heavy impact on the energy consumption.

Inspecting IL code

In order to ascertain whether or not the unused code is actually loaded into memory, the IL code is inspected. Prior to the implementation listed in Snippets 4.1.33 and 4.1.34 a more simplified version was tested. This version did not include the `if` statement, that is line 8-16 in Snippet 4.1.33 and line 9 in Snippet 4.1.34. In the bad variant, the additional methods were still present, however not referenced. The results of this test showed a negligible difference of $< 1\%$ on both energy consumption and execution time. Inspecting the IL code also showed that code for the additional methods were in fact generated even though they were not refer-

enced. Interestingly this did not increase the energy consumption of the `dram` zone, despite the larger code base. The results of this test can be seen in Figure 4.12

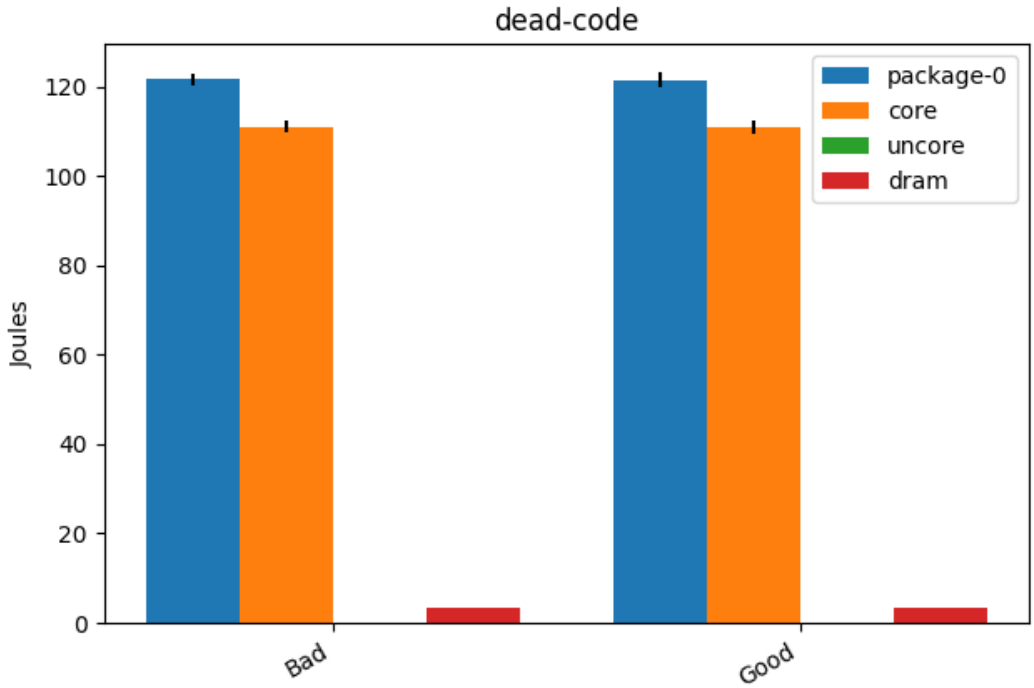

Figure 4.12: Results of the test on Dead Code without the `if` statements in Snippets 4.1.33 and 4.1.34.

Once the `if` statements were included however, the energy consumption of the bad variant increased, despite being guarded behind a conditional statement that will never evaluate to `true`, making it unreachable. In addition to this the `dram` zone shows a small spike in the bad variant in Figure 4.11, indicating that the additional methods were actually loaded into memory in this case.

The results of this experiment shows that the Dead Code code smell does have an impact on the energy consumption of a program. However, this is only the case if the smelly code is referenced in some unreachable location in the code base. While having unused, unreferenced code introduces bloat in a code base, it does not have a noticeable impact on the energy consumption.

4.1.11 In-line Method

In-lining a method is done by the exact opposite principles of Long Method, by moving some lines of code from one method to the method that would other-

wise call it. While this may seem counter intuitive with regards to Long Method, this may reduce the energy consumption as the overhead of calling a method is avoided. [2]

In order to benchmark this smell two classes are created, one for the smell and one where the smell is avoided. Both classes include a method that performs arithmetic calculations on some variables. In the case of the smell, the variables are instantiated and the calculations are performed using another method, whereas for the class where the smell is fixed, the calculations are performed immediately after within the same method. The implementations can be seen in Snippet 4.1.35 and Snippet 4.1.36. [11]

```
1  private int WackyComputations(int
   ↪  age)
2  {
3      age += 2;
4      age = age * 7;
5      age -= 4;
6      age = age / 7;
7      age = age + 1;
8
9      return age;
10 }
11
12 public override void InLine()
13 {
14     int thomasAge = 15;
15     int sophieAge = 32;
16
17     thomasAge =
       ↪  WackyComputations(thomasAge);
18     sophieAge =
       ↪  WackyComputations(sophieAge);
19 }
```

Snippet 4.1.35: Inline Method Code Smell

```
1  public override void InLine()
2  {
3      int thomasAge = 15;
4      int sophieAge = 32;
5
6      thomasAge += 2;
7      thomasAge = thomasAge * 7;
8      thomasAge -= 4;
9      thomasAge = thomasAge / 7;
10     thomasAge = thomasAge + 1;
11
12     sophieAge += 2;
13     sophieAge = sophieAge * 7;
14     sophieAge -= 4;
15     sophieAge = sophieAge / 7;
16     sophieAge = sophieAge + 1;
17 }
```

Snippet 4.1.36: Inline Method Code Smell, refactored

The comparative energy consumption of the tested variants of inline method can be seen in Figure 4.13.

[11] CodeSmellExamples/InLineMethod.cs

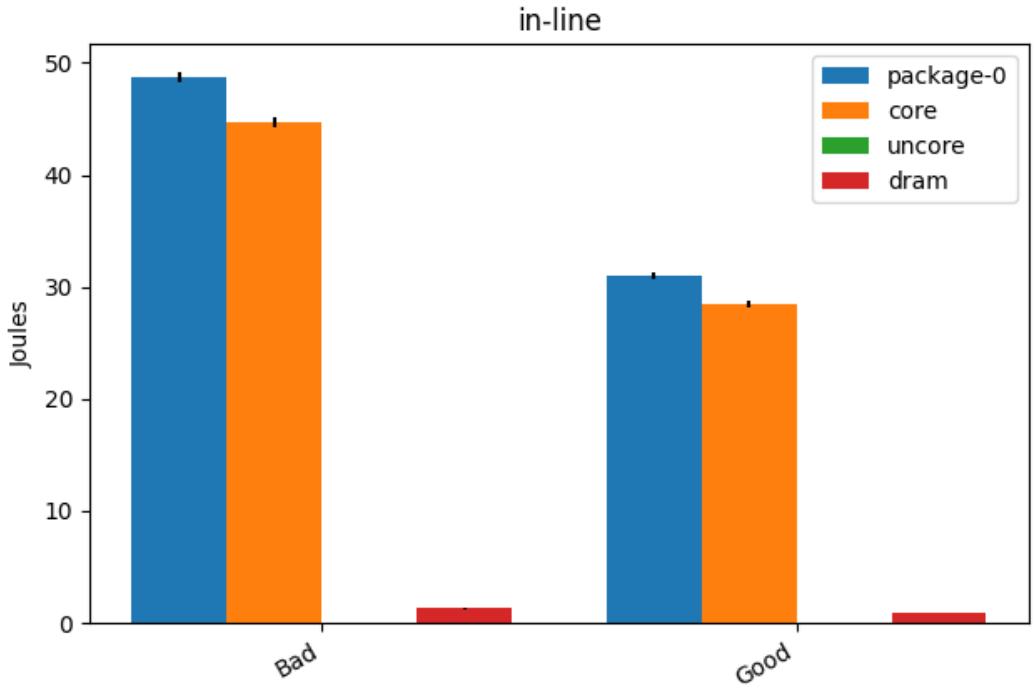

Figure 4.13: Energy consumption of the inline method code smell

The bad variant of inline method consumes on average 48.76 joules per run, whereas the refactored version consumes 31.04 joules per run. This is a significant decrease on the energy consumption of about 36.34%. As for runtime, the bad variant spent 2.84 seconds executing, while the refactored version spent 1.84 seconds, this being a decrease in runtime of about 35.21%.

Analyzing results

As this smell is the inverse of the long method code smell, this is in contrast to the literature, where it was found that refactoring long method, and effectively introducing inline method, had a decrease in energy consumption. As argued by Gottschalk et al., this is likely due to the overhead induced by additional method calls[2], however, as the literature found that long method may decrease energy consumption as well, the reduction may occur on a case-by-case basis.

4.1.12 Redundant Storage of Data

Redundant Storage of Data can occur when two or more methods store identical data in memory rather than it being shared between the methods. Avoiding this smell will effectively reduce the number of memory reads and writes and thereby may reduce the energy consumption of the software. [2]

In order to benchmark this smell two classes are created. Both classes perform three operations on an array: reverse the array, sort the array, and sum the array. The smelly class will instantiate an identical array in each of these methods prior to performing the operations, whereas the class where the smell is fixed, the array is instantiated on class-level, effectively sharing the data between each of the methods. The implementation can be seen in Appendix A.2. [12]

The comparative energy consumption of the tested variants of redundant storage of data can be seen in Figure 4.14.

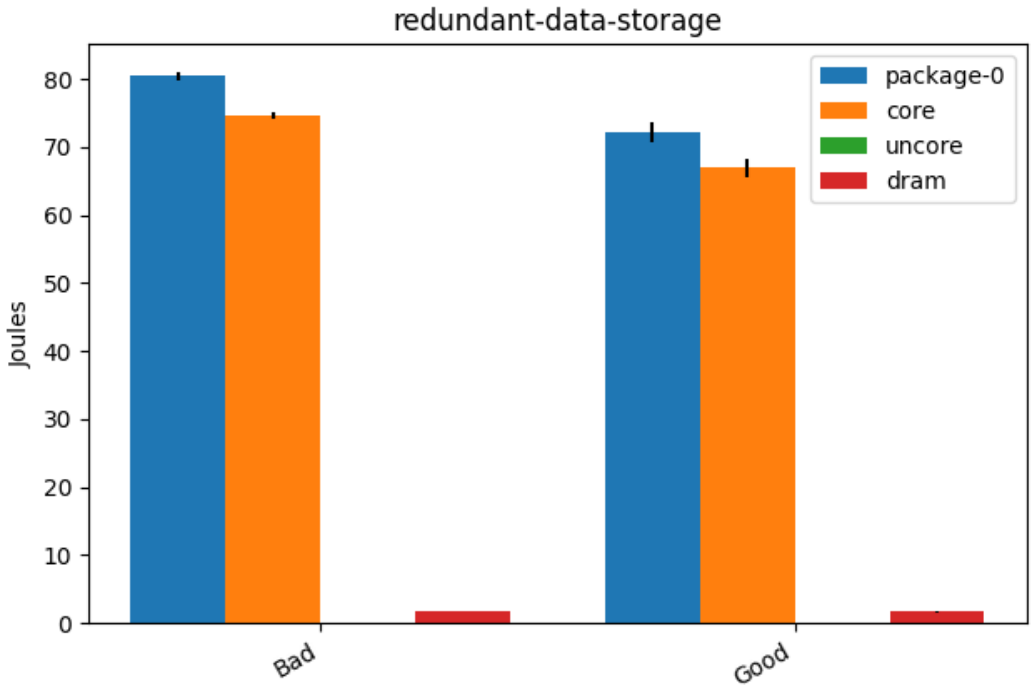

Figure 4.14: Energy consumption of the redundant storage of data code smell

The bad variant of redundant storage of data consumes on average 80.45 joules per run, while the refactored version consumes about 72.14 joules, yielding a de-

[12] CodeSmellExamples/RedundantDataStorage.cs

crease of about 10.33% on the energy consumption. As for runtime, the bad variant spent 4.08 seconds executing, whereas the refactored version spent 3.64 seconds, this being a reduction of about 10.78%.

Inspecting IL code

In order to assess whether the implementation behaves as expected, the IL code is inspected. Snippets 4.1.37 and 4.1.38 lists the IL code for the array usage of the bad and good variants of the code smell, respectively.

```
IL_0001: ldc.i4.s     31 // 0x1f
IL_0003: newarr       [System.Runtime]System.Int32
IL_0008: dup
IL_0009: ldtoken      field valuetype
  ↪  '<PrivateImplementationDetails>'/'__StaticArrayInitTypeSize=124'
  ↪  '<PrivateImplementationDetails>'
  ↪  ::FD69FED59B02FCA8008FCDDD9F46A3CA62228D325620B6259242371B75738D62
IL_000e: call         void
  ↪  [System.Runtime]System.Runtime.CompilerServices.RuntimeHelpers
  ↪  ::InitializeArray(class [System.Runtime]System.Array, valuetype
  ↪  [System.Runtime]System.RuntimeFieldHandle)
IL_0013: stloc.0      // arr
```

Snippet 4.1.37: Array usage in the bad variant of Redundant Data Storage

```
IL_0007: ldfld        int32[]
  ↪  Smells.CodeSmellExamples.RedundantDataStorageGood::arr
```

Snippet 4.1.38: Array usage in the good variant of Redundant Data Storage

The usage listed in the snippets is performed in each of the three methods that are called in the implementation. As seen in the bad variant, a new array is allocated and filled in each method call, whereas for the good version the class-level array is simply referenced whenever it is needed.

It is clear that far more instructions are required for the bad variant, showing a clear cause behind the increased energy consumption. However, unexpectedly this increased is considerably less than that of Dead Local Store, as shown in Section 4.1.3.

Summary

	Bad		Good		Diff	
Smell	*Time*	*Power*	*Time*	*Power*	*Time*	*Power*
Dead local store	5.62	98.62	3.94	65.79	−29.89%	−33.29%
Duplicate code	4.14	74.75	3.72	68.41	−10.14%	−8.48%
Feature envy	6.64	124.12	6.45	119.98	−2.86%	−3.34%
Long method	10.74	183.12	11.81	207.98	+9.96%	+13.58%
Parameter by value	2.36	40.23	2.66	44.92	+12.77%	+14.52%
Repeated conditionals	6.63	114.01	6.43	109.43	−3.02%	−4.02%
Self assignment	2.79	53.46	2.77	53.52	−0.72%	+0.11%
Short circuit	2.67	46.03	2.55	44.08	−4.49%	−4.24%
Type checking (type field)	13.35	254.79	7.33	133.73	−45.09%	−47.51%
Type checking (RTTI)	17.76	335.83	7.33	134.34	−58.73%	−60.00%
Dead code	9.01	155.25	7.57	122.64	−15.98%	−21.00%
Redundant data storage	4.08	80.45	3.64	72.14	−10.78%	−10.33%
In-line method	2.84	48.76	1.84	31.04	−35.21%	−36.34%

Table 4.1: Measured durations and power consumption of the different smells. **Bad** refers to the code smell, and **Good** refers to the implementation where the smell has been refactored. Time is measured in seconds, and power is measured in joules. All power measurements are collected from the RAPL package as a whole.

4.2 Super Smell

To further test the impact of individual smells a so called "Super Smell" is created. The purpose of the Super Smell is to allow us to test numerous smells in the same environment and match various smells with ease. This is partly due to the findings of the paper by Roberto Verdecchia et al. [4] which showed that correcting for specific smells together, might provide an energy saving less than what each smell gave on its own. In order to test this, the Super Smell is able to individually include each smell through its given input parameters. This way, smells can be tested together with relative ease and the relative performance and energy consumption of the system when different smells are included can be analyzed.

The Super Smell also serves to give a more general overview of how the smells affect performance in a larger system. Currently the smells are only measured against their own relative performance, with a varying amount of iterations depending on the relevant smell. This means that despite having found a large relative energy saving between a smell and it's good variant, when put into an actual system it might prove that the overall energy consumption is rather negligent for a smell that otherwise has a large relative gain for being resolved. A smell might see

a 50% energy reduction by having it resolved, however if the corresponding code containing the smell only consumes 50 micro joules in a system that otherwise uses upwards of 200 joules, the reduction, while still present, will not visibly affect overall energy consumption as one might otherwise believe from seeing the initial benchmarks.

4.2.1 Super Smell Testing

Different variations of the Super Smell are tested. One where all smells are included to create a "worst case", one where no smells are included to create a baseline or "best case", and a collection of runs where each individual smell is included and the rest are excluded, so as to give an indication of the individual smell's performance impact on the overall system. Once these tests have been run and the smells with the highest performance impact have been identified, the smells will then be run in pairs so as to see if combining smells might cause a greater influence on overall energy consumption when compared to the smell's individual consumption. This idea comes from the paper by Roberto Verdecchia et al[4]. Of the smells tested, the 5 found to have the individually highest performance impact will be tested this way, so as to limit the scope of the test. Smells will be tested in pairs of 2, 3 and 4 such that each of the 5 smells is tested in every possible combination with the other 4.

The implementation of the super smell can be found linked through the footnote [13].

The results from the test can be seen below in Table 4.2, note that the "good" variant listed for each smell, displays the results for the version run without any of the smells included.

[13]SuperSmell/SuperSmell.cs

	Bad		Good		Diff	
Smell	*Time*	*Power*	*Time*	*Power*	*Time*	*Power*
Dead local store	4.37	85.27	4.37	85.06	0%	−0.25%
Duplicate code	4.38	85.79	4.37	85.06	−0.23%	−0.85%
Feature envy	4.38	86.33	4.37	85.06	−0.23%	−1.47%
Parameter by value	4.38	86.28	4.37	85.06	−0.23%	−1.41%
Repeated conditionals	4.36	85.65	4.37	85.06	+0.23%	−0.69%
Self assignment	4.38	85.58	4.37	85.06	−0.23%	−0.61%
Short circuit	4.38	85.60	4.37	85.06	−0.23%	−0.63%
Type Checking	4.38	85.23	4.37	85.06	−0.23%	−0.20%
Redundant data storage	5.3	110.43	4.37	85.06	−17.55%	−22.97%
Dead Code	4.38	85.58	4.37	85.06	−0.23%	−0.61%
In line method	4.36	85.15	4.37	85.06	+0.23%	−0.11%
All Smells	5.35	111.25	4.37	85.06	−18.32%	−23.54%

Table 4.2: Measured durations and power consumption of the different smells included in the Super Smell. Time is measured in seconds, power is measured in joules. All power measurements are collected from the RAPL package as a whole.

Looking at the test results, we see that every smell, aside from type checking, has a greater power than time saving. Even in the cases of the repeated conditionals and in-line method smells, which both saw an increase in run-time. The overall energy consumption of the good, non-smelly variant, is still lower than that of the version with the smell incorporated.

Redundant data storage is a clear outlier in the data set, as it has an energy saving of almost 24%, whereas the second most notable saving is that of the feature envy smell, which sees a reduction of 1.47%. Compared to the individual test runs, the redundant data storage smell sees a massive reduction in energy in the super smell environment. This is likely due to a larger data set being used in these tests, meaning that copying the data therein becomes a more resource consuming process.

Because redundant data storage has such a large disparity in energy consumption compared to the other smells, it was decided to exclude it from the mixed smell testing. This is due to the fact that any energy reduction or increase in this environment, caused by a smell paired with redundant data storage, would likely be overshadowed by the massive time and power saving from redundant data storage. This would then in turn dilute the test pool, and make it harder to see whether the two smells combined have an actual positive impact on the performance when combined.

As for the total energy consumption, combined all the smells saw an energy decrease of 23.54%. If the energy reduction from the individually tested smells were to be added together however, a total reduction of 29.8% is found. This

further lends credence to the idea that some smells, provide less of an energy saving when paired together, as opposed to when handled individually.

4.3 Mixed Super Smell Testing

In this section, the 5 smells with the highest performance impact found in section 4.2 (discarding redundant data storage) will be further tested in pairs of 2, 3, 4, and 5, so as to test every possible combination of the 5. The tested smells are *Duplicate Code*, *Feature Envy*, *Parameter by Value*, *Repeated Conditionals*, and *Short Circuit*.

To represent the name of each test and make the listing more readable, one letter short hands are used for each of the smells. The short hands are based on the first letter of each smell and are denoted as such:

- **D** - Duplicate Code
- **F** - Feature Envy
- **P** - Parameter by Value
- **R** - Repeated Conditionals
- **S** - Short Circuit

The short hands are put together as the description for the test to describe which of the smells have been run. As an example, the test named D-F-P is the test of the combined smells of Duplicate Code, Feature Envy and Parameter by Value. The percentage difference listed for each smell indicates the potential performance gain should each of the listed smells be fixed. The full list of mixed tests can be seen in Table 4.3.

	Bad		Good		Diff	
Smell	*Time*	*Power*	*Time*	*Power*	*Time*	*Power*
D-F	4.4	85.74	4.37	85.06	−0.68%	−0.79%
D-P	4.42	86.04	4.37	85.06	−1.13%	−1.14%
D-R	4.42	85.86	4.37	85.06	−1.13%	−0.93%
D-S	4.42	85.74	4.37	85.06	−1.13%	−0.79%
F-P	4.4	85.1	4.37	85.06	−0.68%	−0.05%
F-R	4.41	86.22	4.37	85.06	−0.91%	−1.35%
F-S	4.41	85.9	4.37	85.06	−0.91%	−0.98%
P-R	4.42	86.44	4.37	85.06	−1.13%	−1.60%
P-S	4.42	86.24	4.37	85.06	−1.13%	−1.37%
R-S	4.42	85.87	4.37	85.06	−1.13%	−0.94%
D-F-P	4.41	85.89	4.37	85.06	−0.91%	−0.97%
D-F-R	4.44	87.69	4.37	85.06	−1.58%	−3.00%
D-F-S	4.41	86.7	4.37	85.06	−0.91%	−1.89%
D-P-R	4.43	86.65	4.37	85.06	−1.35%	−1.83%
D-P-S	4.43	86.02	4.37	85.06	−1.35%	−1.12%
D-R-S	4.43	86.19	4.37	85.06	−1.35%	−1.31%
F-P-R	4.38	85.79	4.37	85.06	−1.13%	−1.39%
F-P-S	4.42	86.48	4.37	85.06	−1.13%	−1.64%
F-R-S	4.43	86.39	4.37	85.06	−1.35%	−1.54%
P-R-S	4.43	86.57	4.37	85.06	−0.23%	−0.85%
D-F-R-P	4.42	86.2	4.37	85.06	−1.13%	−1.32%
D-F-R-S	4.42	86.19	4.37	85.06	−1.13%	−1.31%
D-F-P-S	4.42	86.03	4.37	85.06	−1.13%	−1.13%
D-P-R-S	4.43	86.57	4.37	85.06	−1.35%	−1.74%
F-P-R-S	4.42	86.06	4.37	85.06	−1.13%	−1.16%
D-F-P-R-S	4.43	86.55	4.37	85.06	−1.35%	−1.72%

Table 4.3: All smell pairings compared to the test not including any smells. Time is measured in seconds, power is measured in joules. All power measurements are collected from the RAPL package as a whole.

To further analyze the results of the tests, additional tables are made comparing the performance of each smell pairing containing a specific smell to the original run of said smell. As an example, any mixed tests containing the Duplicate Code smell will be listed in the Duplicate Code smell table. Here we can then see the performance of the mixed smells containing the Duplicate Code smell as compared to the individually run smell, giving a better overview of the actual performance impact of putting various smells together.

In total, 5 tables will be made to illustrate the performance of the smells, one

for each of the smells. The base case, or "good" performing metric in each table will be the individually run test made for that table's specific smell as found in subsection 4.2.1. This is done with the hope of creating clarity in the relative performance of each smell combination.

The percentage difference listed in each table displays the potential gain should each listed smell besides the one originating the list be fixed. So for example for the listing D-F in the Duplicate Code table, the percentage gain for both power and speed listed, are as when compared to the performance of the system when run only with the Duplicate Code smell. Should this value be positive, it indicates that fixing the listed smells with the exception of duplicate code would heighten the execution time or power consumption of the program.

The test results of each smell pairing are listed in the tables below, with respect to Duplicate Code on Table 4.4, Feature Envy on Table 4.5, Parameter by Value onTable 4.6, Repeated Conditionals on Table 4.7, and Short Circuit on Table 4.8.

	Bad		**Good**		**Diff**	
Smell	*Time*	*Power*	*Time*	*Power*	*Time*	*Power*
D-F	4.4	85.74	4.38	85.79	−0.45%	−0.06%
D-P	4.42	86.04	4.38	85.79	−0.90%	−0.29%
D-R	4.42	85.86	4.38	85.79	−0.90%	−0.08%
D-S	4.42	85.74	4.38	85.79	−0.90%	+0.06%
D-F-P	4.41	85.89	4.38	85.79	−0.68%	−0.12%
D-F-R	4.44	87.69	4.38	85.79	−1.35%	−2.17%
D-F-S	4.41	86.7	4.38	85.79	−0.68%	−1.05%
D-P-R	4.43	86.65	4.38	85.79	−1.13%	−0.99%
D-P-S	4.43	86.02	4.38	85.79	−1.13%	−0.27%
D-R-S	4.43	86.19	4.38	85.79	−1.13%	−0.46%
D-F-R-P	4.42	86.2	4.38	85.79	−0.90%	−0.48%
D-F-R-S	4.42	86.19	4.38	85.79	−0.90%	−0.46%
D-F-P-S	4.42	86.03	4.38	85.79	−0.90%	−0.27%
D-P-R-S	4.43	86.57	4.38	85.79	−1.13%	−0.90%
D-F-P-R-S	4.43	86.55	4.38	85.79	−1.13%	−0.88%

Table 4.4: Smell pairings containing the Duplicate Code smell compared in performance to the initial run of the Duplicate Code smell in the Super Smell. Time is measured in seconds, power is measured in joules. All power measurements are collected from the RAPL package as a whole.

	Bad		Good		Diff	
Smell	*Time*	*Power*	*Time*	*Power*	*Time*	*Power*
F-D	4.4	85.74	4.38	86.33	−0.45%	+0.69%
F-P	4.4	85.1	4.38	86.33	−0.45%	+1.45%
F-R	4.41	86.22	4.38	86.33	−0.68%	+0.13%
F-S	4.41	85.9	4.38	86.33	−0.68%	+0.50%
F-D-P	4.41	85.89	4.38	86.33	−0.68%	+0.51%
F-D-R	4.44	87.69	4.38	86.33	−1.35%	−1.55%
F-D-S	4.41	86.7	4.38	86.33	−0.68%	−0.43%
F-P-R	4.38	85.79	4.38	86.33	−0.90%	+0.08%
F-P-S	4.42	86.48	4.38	86.33	−0.90%	−0.17%
F-R-S	4.43	86.39	4.38	86.33	−0.90%	−0.88%
F-D-R-P	4.42	86.2	4.38	86.33	−0.90%	+0.15%
F-D-R-S	4.42	86.19	4.38	86.33	−1.90%	+0.16%
F-D-P-S	4.42	86.03	4.38	86.33	−0.90%	+0.35%
F-P-R-S	4.42	86.06	4.38	86.33	−0.90%	+0.31%
F-D-P-R-S	4.43	86.55	4.38	86.33	−1.13%	+0.16%

Table 4.5: Smell pairings containing the Feature Envy smell. Time is measured in seconds, power is measured in joules. All power measurements are collected from the RAPL package as a whole.

	Bad		Good		Diff	
Smell	*Time*	*Power*	*Time*	*Power*	*Time*	*Power*
P-D	4.42	86.04	4.38	86.28	−0.90%	+0.28%
P-F	4.4	85.1	4.38	86.28	−0.90%	+1.39%
P-R	4.42	86.44	4.38	86.28	−1.13%	−0.19%
P-S	4.42	86.24	4.38	86.28	−0.90%	+0.05%
P-D-F	4.41	85.89	4.38	86.28	−0.68%	+0.45%
P-D-R	4.43	86.65	4.38	86.28	−1.13%	−0.43%
P-D-S	4.43	86.02	4.38	86.28	−1.13%	+0.30%
P-F-R	4.38	85.79	4.38	86.28	−0.90%	+0.02%
P-F-S	4.42	86.48	4.38	86.28	−0.90%	−0.23%
P-R-S	4.43	86.39	4.38	86.28	−1.13%	−0.13%
P-D-F-R	4.42	86.2	4.38	86.28	−0.90%	+0.09%
P-D-F-S	4.42	86.03	4.38	86.28	−0.90%	+0.29%
P-D-R-S	4.43	86.57	4.38	86.28	−1.13%	−0.33%
P-F-R-S	4.42	86.06	4.38	86.28	−0.90%	+0.26%
P-D-F-R-S	4.43	86.55	4.38	86.28	−1.13%	−0.31%

Table 4.6: Smell pairings containing the Parameter by Value smell. Time is measured in seconds, power is measured in joules. All power measurements are collected from the RAPL package as a whole.

	Bad		Good		Diff	
Smell	*Time*	*Power*	*Time*	*Power*	*Time*	*Power*
R-D	4.42	85.86	4.36	85.65	−1.36%	−0.24%
R-F	4.41	86.22	4.36	85.65	−1.13%	−0.66%
R-P	4.42	86.44	4.36	85.65	−1.36%	−0.91%
R-S	4.42	85.87	4.36	85.65	−1.36%	−0.26%
R-D-F	4.44	87.69	4.36	85.65	−1.80%	−2.33%
R-D-P	4.43	86.65	4.36	85.65	−1.58%	−1.15%
R-D-S	4.43	86.19	4.36	85.65	−1.58%	−0.63%
R-F-P	4.38	85.79	4.36	85.65	−0.46%	−0.27%
R-F-S	4.43	86.39	4.36	85.65	−1.36%	−1.66%
R-P-S	4.43	86.57	4.36	85.65	−1.58%	−0.86%
R-D-F-P	4.42	86.2	4.36	85.65	−1.36%	−0.64%
R-D-F-S	4.42	86.19	4.36	85.65	−1.36%	−0.63%
R-D-P-S	4.43	86.57	4.36	85.65	−1.58%	−1.06%
R-F-P-S	4.42	86.06	4.36	85.65	−1.36%	−0.48%
R-D-F-P-S	4.43	86.55	4.36	85.65	−1.58%	−1.04%

Table 4.7: Smell pairings containing the Repeated Conditionals smell. Time is measured in seconds, power is measured in joules. All power measurements are collected from the RAPL package as a whole.

	Bad		Good		Diff	
Smell	*Time*	*Power*	*Time*	*Power*	*Time*	*Power*
S-D	4.42	85.74	4.37	85.06	−0.90%	−0.16%
S-F	4.41	85.9	4.37	85.06	−0.68%	−0.35%
S-P	4.42	86.24	4.37	85.06	−0.90%	−0.74%
S-R	4.42	85.87	4.37	85.06	−0.90%	−0.31%
S-D-F	4.41	86.7	4.37	85.06	−0.68%	−1.27%
S-D-P	4.43	86.02	4.37	85.06	−1.13%	−0.49%
S-D-R	4.43	86.19	4.37	85.06	−1.13%	−0.68%
S-F-P	4.42	86.48	4.37	85.06	−0.90%	−1.02%
S-F-R	4.43	86.39	4.37	85.06	−0.90%	−1.72%
S-P-R	4.43	86.57	4.37	85.06	−1.13%	−0.91%
S-D-F-R	4.42	86.19	4.37	85.06	−0.90%	−0.68%
S-D-F-P	4.42	86.03	4.37	85.06	−0.90%	−0.50%
S-D-P-R	4.43	86.57	4.37	85.06	−1.13%	−1.12%
S-F-P-R	4.42	86.06	4.37	85.06	−0.90%	−0.53%
S-D-F-P-R	4.43	86.55	4.37	85.06	−1.13%	−1.10%

Table 4.8: Smell pairings containing the Short Circuit smell. Time is measured in seconds, power is measured in joules. All power measurements are collected from the RAPL package as a whole.

Interestingly, every single Super Smell measurement, when compared to the single run smell test, has a positive time impact when fixed. In regards to power however, the Parameter by Value and Feature Envy smells largely show results where the power consumption increases as more smells are fixed. This indicates that some smells may be better left unfixed when seen in combination with one another. The most obvious outlier is the case of the F-P smell, where a more than 1% increase in power consumption is seen if either of the two smells are fixed exclusively, however either leaving both smells as is or removing them both gives a similar performance in regards energy consumption. This is a rather surprising result, as we had initially thought the decline in energy would be a more gradual process. Where since every smell on its own has a positive impact on overall energy consumption as seen in Table 4.2, taken in tuple pairs, while not necessarily scaling 1-1 in performance, we expected to see a steady decline in overall energy consumption as more smells are fixed.

However, through our tests we instead found that a lot of smells, typically when paired with either the Feature Envy or Parameter by Value smell, can have a directly negative impact on performance if fixed while leaving the remaining smell still in the program. Interestingly however despite the potential loss of removing only certain smells, the overall best performing version of the program is nevertheless still the smell free version. The closest in regards to energy performance is

that of the F-P mixed smell, which performs 0.05% worse than the smell free version. This means that these two smells when put together, creates a system which performs better than any system where only a single smell is present. While in our case we found that the best case for improving performance is simply fixing all present smells. The fact that performance between a smell free version and versions where some smells are present are so similar, raises the question of whether that would also have been the case given a larger subset of smells. Only 5 smells were tested during our super smell testing, however had we added more to the list one could speculate that a combination of smells could be found which, when joined together, outperforms the smell free version in regards to power consumption.

It should be noted that while energy impact can be both positive and negative, the execution time of the various smells stays roughly the same, compared to the individually run smells an increase in execution time is always seen, some combinations of smells have a slightly faster execution speed, but overall the results are typically in the range of +/- 1 millisecond, which lends credence to execution time not being majorly affected by the additional smells.

Chapter 5

Linter

This chapter will explain the implementation of the linter. Firstly the selected framework will be explained, and then the specific code smells that are implemented will be listed and explained.

5.1 Selected Framework

In Section 2.3 several existing frameworks were explored and explained briefly. In order to avoid re-inventing the wheel one of these have been selected to be extended to include energy heavy patterns and will thereby serve as the linter.

Before selecting the framework, a few requirements need to be established. The framework should allow for:

- easily implementing custom rules that detects code smells
- flexible rules that allow for catching varied cases of the same smell

Roslyn Analyzers, as described in section 2.3.1, are a number of static code analyzers included in the .NET compiler platform Roslyn. Being open source it is possible to extend the analyzers with custom rules, however being embedded in the code it is not as flexible and easy to use as some of the other options, and as such Roslyn Analyzers will not be used.

SonarQube, described in section 2.3.2, is one of the more used linters of the examined framworks and works across many different languages. However, due to limitations within the program, it is not possible to create custom rules for C#, and as such SonarQube will not be used.

Semgrep, described in section 2.3.3, allows for defining rules using regex with a few additional keywords that detect, for example, a function name. While the extra keywords add some flexibility, defining rules with regex may quickly become very complex and in some cases impossible, and as such Semgrep will not be used.

Much like Semgrep, InsiderSec, described in section 2.3.4, allows for defining rules using regex. However, in the case of InsiderSec there are no extra keywords, making the task of writing rules more difficult, and as such InsiderSec will not be used either.

CodeQL, described in section 2.3.5, generates a database from both the source and assembly code of a program, and allows for creating `.QL` query files for detecting a code smell. Within a query file it is also possible to define predicates and helper functions that can be used in the query itself. Given that a rule in CodeQL essentially exists as a script rather than e.g., a line of regex, it is deemed the most flexible of the options, and as such CodeQL is selected as the framework to be used for the linter.

5.2 Using CodeQL

This section will give a deeper explanation of CodeQL, including the initial setup required in order to use it, as well as how to use it.

5.2.1 Initial setup

The initial setup of CodeQL has a few mandatory steps. The CodeQL team recommend containing everything needed in a single folder, e.g., `codeql-home`. Once this folder is created the official CodeQL repository[1] should be cloned. This repository includes predefined rules for every supported language, and is as such necessary in order to analyze a program. They recommend renaming this folder to `codeql-repo`.

As the official repository contains only the source code, it is also necessary retrieve the binary. For each release the various binaries are bundled and can be downloaded from the repository[2]. Once downloaded, the binary should either be added to the users' `PATH` environment variable, or be symlinked to `/usr/bin`.

Once this is done, run `codeql resolve qlpacks` in a terminal in order to ensure that all the QL packs are detected. A QL pack is a collection of rules for a given category or language. For example, the QL pack `csharp` includes all rules implemented for C#.

5.2.2 Generating the database

As mentioned in Section 2.3.5, CodeQL generates a database containing information about the code base, being a necessary step to execute before being able to

[1] `https://github.com/github/codeql`

[2] `https://github.com/github/codeql-action/releases`

analyze a program. To generate the database, open a terminal and change directory to the root of the code base, and run the following command:

```
$ codeql database create DATABASE_NAME --language=LANGUAGE_ID
```

where `DATABASE_NAME` is an arbitrary name of the database to generate which will be used as a target for the analysis. `LANGUAGE_ID` is the identifier for the language to generate for, e.g., `csharp` for C# or `java` for Java. If `LANGUAGE_ID` is not specified, CodeQL will try to infer it from the code base.

5.2.3 Analyzing the database

Once the database is generated, the code base can be analyzed. While still in the root directory of the code base, analyze the program by running the following command:

```
$ codeql database analyze DATABASE_NAME --format=OUTPUT_FILE_FORMAT
                --output=OUTPUT_FILE_NAME QUERIES
```

where `DATABASE_NAME` is the name of the database generated previously. `OUTPUT_FILE_FORMAT` is the format of the output, and CodeQL currently supports either csv or sarif. `OUTPUT_FILE_NAME` is the name of the output file, with or without file extension. `QUERIES` is a number of QL packs to analyze the code base with, e.g., `csharp`. If `QUERIES` are not specified, CodeQL will use the default QL packs for the language of the code base.

5.2.4 Output

As mentioned in the previous section, the results of the analysis will be printed to a file of the specified type. If csv is chosen as file type, each line with a result will have the following structure:

`@name`, `@description`, `@problem.severity`, message, location of file, location in file

where the fields prefixed by @ are defined by the query metadata. The message is defined by the `select` clause, and the location is the file where the error is contained, as well as line row and column.

5.2.5 Automating the process

One caveat of using CodeQL is that for every change in the code base that needs to be analyzed, the database needs to be re-generated. Because of this, running CodeQL as a CI task on e.g., the GitHub repository whenever a commit is pushed

could be favorable. If running CodeQL locally is preferred, a good idea is to write a script that performs the necessary tasks, instead of having to run several lengthy commands every time the code base needs to be analyzed. Such a script can be seen in Snippet 5.2.1.

```
#!/bin/bash

DB_NAME="smelldb"
LANG="csharp"

echo "################## cleaning up"
rm -r $DB_NAME
echo "################## creating database for project"
codeql database create $DB_NAME --language=$LANG
echo "################## analysing database"
codeql database analyze $DB_NAME --format=csv --output=res.csv
↪  csharp-energy-aware-queries
echo "Done. Results written to res.csv"
```

Snippet 5.2.1: Script for performing CodeQL analysis tasks

5.3 CodeQL queries

This section will list and explain the CodeQL queries that make up the energy aware linter, as well as how they are defined as a collection. The queries can be found in the following repository:

`https://github.com/Jones-og-Hartvig-Master/energy-queries`

It should be noted that the results generated from these queries are merely suggestions that can reduce the energy consumption. There may be cases where the smells are intentional behavior. For example, a conditional may be repeated in a method because some other behavior needs to occur before the body of the second condition is executed.

In addition to this, in some cases there may be a trade off between the energy consumption and other software metrics, such as maintainability or readability. For example, a class that introduces the Feature Envy smell may have been created in order to increase one of these other metrics at the cost of a slight increase in energy consumption.

5.3.1 QL pack

As touched upon in the previous section, a QL pack is a collection of rules for a given category or language in CodeQL. The queries created for the energy aware linter is as such defined as a QL pack. A QL pack is identified by a YAML file located in the root directory of the collection of queries. The YAML file defining this QL pack can be seen in Snippet 5.3.1.

```
1 name: csharp-energy-aware-queries
2 version: 0.0.1
3 libraryPathDependencies: codeql/csharp-all
```

Snippet 5.3.1: QL pack specification file

Line 1 specifies the name of the QL pack, meaning the name `csharp-energy-aware-queries` should be included in the `QUERIES` field explained in Section 5.2.3. In addition to this, the YAML file includes a version number and the dependencies required for this QL pack, namely the collection of queries for `csharp`.

5.3.2 Retrieved from official repository

The collection of QL queries already present in the CodeQL repository already includes some of the code smells included in this report, and as such these are simply retrieved from there. The queries that have been retrieved from the repository are Feature Envy[3] and Dead Local Store[4].

5.3.3 Parameter by Value

Snippet 5.3.2 shows the QL query for Parameter by Value. The results of the tests found that for C#, `ref` type parameters actually consume more energy than a regular type, and as such this query has been inverted to reflect this.

The query includes one variable declaration, namely a method `m`. The `where` clause requires that this method is a source definition and not an invocation, and that at least one parameter in its' list of parameters is a `ref` type. For each method where this clause holds, it will be reported with the location of the method and the message specified on lines 18 and 19.

[3] FeatureEnvy.ql
[4] DeadStoreOfLocal.ql

```
/**
 * @name Parameter by value
 * @description Using parameters by reference instead of value
 * @kind problem
 * @precision low
 * @problem.severity recommendation
 * @id cs/energy-aware/parameter-by-value
 * @tags energy-aware
 */

import csharp

from Method m
where
    m.isSourceDeclaration() and
    exists(Parameter p | p = m.getAParameter() and p.isRef())
select m,
    "Method " + m.getName() + " has a parameter of ref type which can consume more
    ↪  " +
    "energy than a non-ref type"
```

Snippet 5.3.2: .QL query for Parameter by Value

5.3.4 Non-Short Circuit

Snippet 5.3.3 shows the QL query for Non-Short Circuit. The query includes one variable declaration, a binary bitwise operation `bbo`. The `where` clause requires that this operation is either a bitwise `or`, or `and` expression. For each operation where this clause holds, it will be reported with the location of the method and the message specified on line 16.

```
/**
 * @name Non-short circuit
 * @description Not using short circuiting boolean operators can increase energy
 ↪  consumption.
 * @kind problem
 * @precision
 * @problem.severity warning
 * @id cs/energy-aware/non-short-circuit
 * @tags energy-aware
 */

import csharp

from BinaryBitwiseOperation bbo
where
    bbo instanceof BitwiseAndExpr or bbo instanceof BitwiseOrExpr
select bbo, "Operator " + bbo.toString() + " should be used with short circuiting"
```

Snippet 5.3.3: .QL query for Non-Short Circuit

5.3.5 Repeated Conditionals

Snippet 5.3.4 shows the QL query for Repeated Conditionals. This query includes two predicate functions. The predicate function `hasMultipleIfstmt` will check if a block statement has multiple if statements by looking through the statements present in this block. The other predicate function, `sameConditionals`, will check the condition of every if statement present in a block statement against each other, and will hold if any of them are the same.

The query itself includes two variable declarations, namely a method `m`, and an integer denoting the number of if statements present in the method. The `where` clause requires that both of the aforementioned predicate functions holds. For each method where this clause holds, it will be reported with the location of the method and the message specified on line 33.

```
/**
 * @name Repeated conditionals
 * @description A conditional statement that occurs more than once within the same
 ↪  scope.
 * @kind problem
 * @precision low
 * @problem.severity recommendation
 * @id cs/energy-aware/repeated-conditionals
 * @tags energy-aware
 *       maintainability
 */

import csharp

predicate hasMultipleIfstmt(BlockStmt bs, int nIfstmt) {
    nIfstmt = count(Stmt s | s = bs.getAChildStmt() and s instanceof IfStmt) and
    nIfstmt > 1
}

predicate sameConditionals(BlockStmt bs) {
    exists(
        Stmt s, Stmt ss |
        s = bs.getAChildStmt() and s instanceof IfStmt and
        ss = bs.getAChildStmt() and ss instanceof IfStmt and
        s.getLocation().toString() != ss.getLocation().toString() and
        s.(IfStmt).getCondition().toString() =
        ↪  ss.(IfStmt).getCondition().toString())
}

from BlockStmt bs, int nIfstmt
where
    hasMultipleIfstmt(bs, nIfstmt) and
    sameConditionals(bs)
select bs,
        "Method " + bs.getEnclosingCallable().getName() + " implements the same
        ↪  conditional statement in multiple locations within the same scope."
```

Snippet 5.3.4: .QL query for Repeated Conditionals

5.4 Testing the linter

In order to validate whether the developed CodeQL queries can correctly catch the intended code smells, they should be tested against a code base. Ideally this code

base should include at least all the code smells implemented as queries. As such, the collection of code smells used for experiments throughout this report will be used.

To perform the tests CodeQL is setup as described in section 5.2.1 with the addition of adding the energy aware queries to the `codeql-home` directory. After this, the code smell collection is analyzed from the root directory of the project following the steps described in sections 5.2.2 and 5.2.3, specifically this is done using the script listed in snippet 5.2.1. The result of this process is a number of detections in an output file. Table 5.1 shows the summarized results from the output file.

Smell	**Expected count**	**Actual count**
Feature Envy	2	2
Dead Local Store	6	46
Parameter by Value	2	2
Non-Short Circuit	4	4
Repeated Conditionals	2	2

Table 5.1: The results from running the energy aware queries against the code smell collection.

For Feature Envy, the smell was present once in its designated file, and once in the Super Smell, and the query successfully caught both.

For Dead Local Store six results were expected, three from its' designated file and three from the Super Smell. However, 40 additional cases were caught and this is because there are a large number of these cases in the other smell implementations. For example, in the Dead Code implementation there are six dead local stores in order to make sure that the dead methods are referenced.

The remaining three, Parameter by Value, Non-Short Circuit, and Repeated Conditionals all had their expected amount of occurrences detected. For all of them, half the occurrences were in their designated file, and the other half was in the Super Smell.

Chapter 6

Discussion

6.1 Applying the linter in the wild

The smells have been tested in microbenchmarks which on the previous semester were found not to entirely reflect a real world situation [12]. Because of this, the smells were also tested in a larger macrobenchmarking environment, which showed entirely different results. As such it is deemed necessary to test whether the smells have a similar impact in an even larger, real-world software scenario. Because such a test was not done during the project, this section will explain a methodology for doing so.

6.1.1 Identifying test programs

A first step to performing these tests is to identify a number of programs to perform the tests on. Thankfully the open source community is very large, and as such it should be possible to find a number of programs suitable for testing the linter on. As an aid to this goal, there exists a number of collections of open source programs for different fields and language, e.g., for C#[1][2][3]. As such, the test programs will be identified from these sources.

In order to narrow down the search for test programs, they should each have a number of key characteristics. Namely, each test program should

- be implemented in C#
- have a number of code smells present in the original code
  - To be tested using the energy aware QL pack for CodeQL

[1] `https://github.com/uhub/awesome-c-sharp`
[2] `https://github.com/quozd/awesome-dotnet`
[3] `https://awesomeopensource.com/projects/c-sharp`

- be easy to set up and build
- preferably be a one-shot program that terminates on its' own, or be able to emulate that behavior
  - This would make it possible to automate the testing process using `rapl.rs`

6.1.2 Performing the tests

Once the test programs have been verified with regards to the list of key characteristics they should be tested. Each test program is likely to be different, and a test procedure should be defined for each of them. This procedure should include execution of the code smells, and should ideally comprise most of the functionality of the system, if possible.

Firstly they should be tested before refactoring any smells in order to have a baseline of the energy consumption. This test will then serve as the "bad" variant of a particular program. Once this is done, the code smells found in the program should be refactored and the test should be performed once again. This will then serve as the "good" variant of a particular program. When both tests are performed the results should be analyzed to ascertain what impact the code smells have on the test programs.

Each test will be done using `rapl.rs` as a benchmarking tool, providing various information about the energy consumption during each run.

6.2 Differing Results

Throughout the report, two "main" sets of tests are run in total, the Super Smell tests and the initial solo run smells. Looking at the results of either however, it quickly becomes apparent that the smells which cause the biggest increase in power consumption are very different from one test set to the other. The potential savings for each smell are also generally much higher during the initial solo tests than in the Super Smell. We put most of this down to the effective size of the smells.

In the solo tests, the program is run potentially millions of times with the sole focus of testing a singular smell. This then in turn means that any potential reduction in power consumption is not affected by any outside factors, which makes it a rather poor benchmark for seeing the actual performance of the various smells. For example, the Dead Local Store smell shows a 33% reduction in power consumption when fixing the smell, however when tested in the Super Smell the reduction drops to 0.25%. The reduced reduction is very likely caused by a clearer view as to how much of an impact the smell would have in a larger system. This also makes the

initial tests rather misleading, as despite entries like Type Checking resulting in an energy saving of over 50%.

When taken to a real system, due to how little of total time is spent executing that particular smell, the overall energy impact also becomes lower. This means that the results from the initial solo run tests are somewhat misleading in regards to how big of an impact some of the smells have. This is somewhat covered once the Super Smell is run after, and more practical results are found. But knowing the direct performance impact of a singular smell in comparison to only itself with no connection to any larger program, is not necessarily useful for any actual implementation which seeks to tackle energy consumption in programs that size beyond micro benchmarks.

6.3 Setup of the Super Smell

To make it easier to facilitate testing, each smell within the Super Smell is compartmentalized into two semantically equivalent chunks. Upon execution of the super smell, given input parameters determine whether the chunk with or without the smell is run in the actual test. This makes testing easier, as smells can easily be enabled or disabled through the input parameters.

However it also has the trade off of making the smells themselves completely compartmentalized and have no direct interaction with one another within the program. This means that tested smells do no necessarily have as much of an impact on each other as they would have in a more standard program, where there is no guarantee that smells are not more gathered or potentially even present within the same line of code. This is however, a conscious trade off which was made to allow for easier testing and mixing of the various smells. As such, while the mixed test results might not be directly applicable for all programs, it nevertheless shows how smells can affect one another, even when not present directly next to one another in the program itself.

6.4 Varying iteration counts

In the initial solo run smell tests, iteration count was chosen at random to be a number that would simply allow the program to run for a "decent" amount of time, which usually meant a couple seconds. This was done as we are not interested in the relative performance of the programs when compared to each other, only how big the difference is from the smell to the non-smell version. Since the iteration count of both versions of the program are scaled linearly alongside one another, increasing it does not impact relative power and time usage between a test's two versions.

Having varying iteration counts for each smell does however mean that the initially run tests cannot be compared with one another beyond the percentage increase for including a smell. The actual concrete energy measurements should only be viewed in relation to the other version of that specific smell, as comparing it to other smells portrays a false image of the performance. During the Super Smell tests, each smell is run for the same amount of iterations within a larger system, which means the results from these tests can be seen as a more accurate representation of performance when compared to one another, at least in regards to the actual power consumption/execution time of the individual smells.

6.5 Threats to Validity

This section will go over the identified threats to validity in this project.

6.5.1 Temperature

As mentioned in section 3.1 the temperature of the group room is a factor out of our control. The test tool `rapl.rs` measures the temperature of the CPU during the run, and this feature has been tested on multiple machines and works as intended. However, on the test machine the CPU temperature sits at a static 28C. This likely means that the cooling of the CPU is adequate and that the tests should not be affected by temperature fluctuations on the CPU.

On the other hand, the temperature of the room the test machine is situated in is likely to fluctuate. This room is a shared group room and throughout most days people will be sitting in that room while working. Because of this and that the windows may be opened during the day, the temperature will likely change and may have an impact on the results of the tests. As such, the temperature of the group room is deemed to be a threat to validity.

6.5.2 Real-world software tests

Throughout this report 13 different code smells have been tested in both a micro- and macrobenchmarking environment. The microbenchmarks showed that in an isolated situation the code smells increase the energy consumption as opposed to their refactored counterparts. The macrobenchmarks showed that the smells do increase the energy consumption in a larger context, however to a lesser extent. However, both these cases were written by the participants of this project, and as such do not reflect software found in the "real world".

As such, the lack of tests on real-world software is seen as a threat to validity. In order to eliminate this threat, the linter should be tested on a number of real-world programs with regards to the procedure explained in section 6.1.

6.5.3 Database size influence on smells

Despite it better representing a larger program, the results from the Super Smell can also not always be taken at face value. A decently sized .csv file was accessed and modified by all of the smells in the Super Smell. While all programs are different and no one benchmark can fully encompass all use cases, it is worth noting that due to the large database used, the found results from the super smell may not be applicable for all cases. This becomes especially apparent with a smell like Redundant Data Storage, where the data from our csv file is redundantly copied for some access point. In our case this causes a massive increase in power consumption, however had the database been smaller, it would in all likelihood also cause the smell to cause less of an energy reduction within the program.

To prove this this theory, a small test was setup for the Super Smell for the redundant data storage smell. The main difference from the previous runs being that the amount of entries from the .csv file was reduced from about 10.000 to 2. To have a fair comparison a base case is also run with the reduced .csv file, where no smells are present. This gives a clear indication as to how much the size of the database affects this specific smell, as no other factors are changed within the Super Smell. The results can be seen in Figure 6.1.

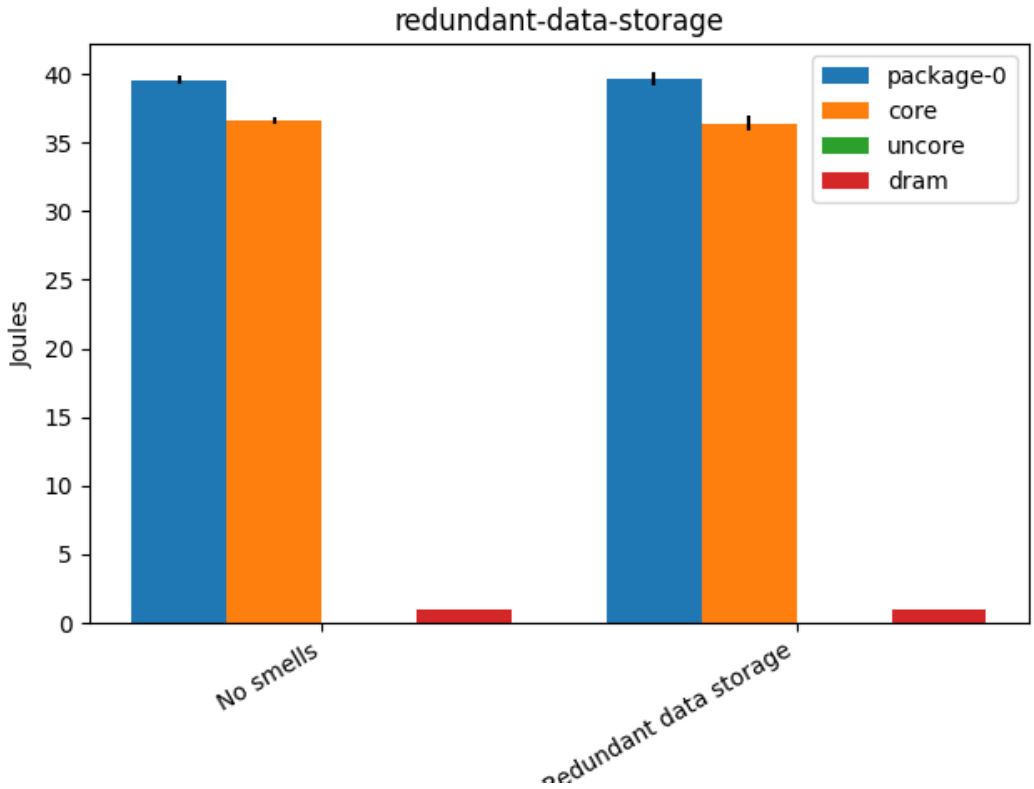

Figure 6.1: Reduced database redundant data storage smell test

As seen, the relative cost of the Redundant Data Storage smell is reduced to a negligible level as the size of the database is decreased. The overall energy consumption of the no smells version of the super smell is also reduced from just over 85 joules to 39.64, meanwhile the Redundant Data Storage smell's cost is reduced from 110.43 joules to 39.56 joules. This illustrates that the performance impact of some smells is highly dependent on the setup of the specific system.

Chapter 7

Conclusion and Future Work

This chapter will conclude on the project and present a number of points to a list of future work that can be done to enhance the results.

7.1 Conclusion

A large number of code smells have been shown to have an impact on not only the execution time and other software metrics such as maintainability, but also the energy consumption in more recent years. A lot of energy aware research has been done with regards to mobile devices, experimenting with patterns and states that consume a large amount of energy. While this is an important topic due to mobile devices' necessity for being charged once in a while, and embedded systems requiring a low energy consumption, this has only been examined on desktop machines to a smaller extent, with regards to code smells. For desktop machines, related literature mainly deal with code smells in a microbenchmarking environment. In addition to this, we found during our 9th semester project that microbenchmarking results do not necessarily reflect the impact that a given refactor will have on a larger, real-world system [12].

In this report we set out to examine a number of code smells implemented in C#. We selected a number of code smells from related literature which examined code smells in Java[4] for desktop systems in order to see if these results carried over to C#. In addition, we also selected a number of code smells from related literature which examined code smells in C++ for embedded systems [1] in order to see if code smells that have an impact on these systems carry over to a desktop environment.

For the code smells we implemented a modular program that allows for easily adding new code smells through its' dispatch component. Each code smell has its' own file in which classes for both a bad and good variant are present. This program can be run from the terminal with arguments such as `feature-envy good`, i.e., to

run the good variant of Feature Envy. Each energy measurement has been done using the benchmarking tool `rapl.rs`.

In total 13 code smells were selected and tested. Of these 13, two were found to have a positive impact on the energy consumption, one had a negligible difference, and the remaining ten had a small to large negative impact. In a microbenchmarking environment, refactoring yielded an energy decrease of up to 60%. However, the smells were also tested in a macrobenchmarking environment, the "Super Smell", where the results were drastically different. Here, all of the smells induced an increase in energy consumption, however for the majority only increased it by up to 1.5%. One particular smell increased the energy consumption by nearly 23%, however this is explained by the nature of the Super Smell program.

Furthermore we performed mixed smell testing with the top five most consuming smells, by testing all unique combinations of the five smells. The singular run of each of the five smells were then compared to each combination where that particular smell was included. From this analysis it was found that while each smell has a negative impact on the energy consumption on its own, this is not necessarily the case when combined with at least one other smell. Particularly for Feature Envy the energy consumption decreased in most cases when combined with other smells.

The contribution of this report is a QL pack of energy aware queries for the semantic analysis engine CodeQL. A number of the tested smells have been implemented as queries and have been included in this QL pack. The queries have initially only been tested on the collection of code smells where the smelly locations in the code were correctly detected. However, a plan for testing a number of open source repositories has been made.

7.2 Future work

In addition to what was done during this report, there are a number of things that could be done to explore the area. This section will go over some potential future work that could be done.

7.2.1 Apply the linter in the wild

As described in section 5.4 the linter was only tested on our collection of code smells in order to validate whether it worked as intended. Section 6.1 put forth a procedure for applying the linter in the wild, i.e., finding a number of open source C# repositories with a number of smells present in the code base, running the energy aware queries against the code base, and fixing said code smells in order to measure the difference in energy consumption.

Given that we have shown the large difference between code smell impact in micro- and macrobenchmarking environments, it would be very interesting to see what kind of impact it can have on real-world programs.

7.2.2 Further examine the low level behavior of smells

During this report we found that some code smells have an negative impact on the energy consumption of software. For most of the smells either the IL or assembly code was examined to ensure that they exhibited the expected behavior, but also to get a view of the reason behind the increase in energy consumption. As a point to future work it could be interesting to take a deeper dive into the IL and assembly code in order to further analyze the cause behind the increase, e.g., to identify patterns that are generally heavy on the energy consumption.

7.2.3 Set up for continuous integration

As mentioned in section 5.2.5 the nature of the workflow of CodeQL makes it very suitable for automation processes. For this reason another point to future work would be to create a GitHub workflow that will run the CodeQL process explained in section 5.2 as an action, e.g., whenever a commit is pushed to the repository.

Additionally, this could be created as a GitHub Action that could easily be integrated into any GitHub repository[1].

7.2.4 Examine different languages

The smells analyzed in this report have only been tested with C# implementations, and the CodeQL queries have only been implemented for C#. The code smell Parameter by Value was shown to have a different behavior than what was found in the related works with regards to energy consumption. The specific paper used C++ for their tests, and we used C#, meaning that energy consumption of different instructions varies from language to language. As such an interesting point to future work is to perform the tests on other languages, such as Python or Go. In addition, CodeQL supports in total nine different languages that the energy aware queries could be implemented for.

7.2.5 Finish the energy aware queries

Thorughout this paper, 13 smells are tested individually in order to examine their impact on the energy consumption before and after refactoring. Only a subset of these are implemented into the QL pack for CodeQL, and as such a potential

[1] `https://github.com/features/actions`

point for future work would be to implement queries for the remaining smells and include them in the energy aware QL pack.

7.2.6 Include additional smells

Another point to future work would be to include more smells. This would require testing of these new smells in both a micro- and macrobenchmarking environment so as to ascertain their impact. Any code smell shown to increase the energy consumption would then be included in the energy aware QL pack.

Bibliography

[1] A. Vetro et al. *Definition, Implementation and Validation of Energy Code Smells: an Exploratory Study on an Embedded System*. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.701.5849&rep=rep1&type=pdf. Last retrieved 17-02-2022.

[2] M. Gottschalk et al. *Removing Energy Code Smells with Reengineering Services*. URL: https://dl.gi.de/bitstream/handle/20.500.12116/17849/441.pdf?sequence=1&isAllowed=y. Last retrieved 12-04-2022.

[3] N. Tsantalis et al. *JDeodorant: Identification and Removal of Type-Checking Bad Smells*. URL: https://www.researchgate.net/publication/27378108_JDeodorant_Identification_and_Removal_of_Type-Checking_bad_Smells. Last retrieved 15-02-2022.

[4] R. Verdecchia et al. *Empirical Evaluation of the Energy Impact of Refactoring Code Smells*. URL: https://www.researchgate.net/publication/323345580_Empirical_Evaluation_of_the_Energy_Impact_of_Refactoring_Code_Smells. Last retrieved 15-02-2022.

[5] Anders SG Andrae and Tomas Edler. "On global electricity usage of communication technology: trends to 2030". In: *Challenges* 6.1 (2015), pp. 117–157.

[6] William G. Cochran. *Sampling Techniques, 3rd Edition*. John Wiley, 1977. ISBN: 0-471-16240-X.

[7] Howard David et al. "RAPL: Memory power estimation and capping". In: *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*. 2010, pp. 189–194. DOI: 10.1145/1840845.1840883.

[8] K. Beck & M. Fowler. *Bad Smells in Code*. URL: http://www-public.tem-tsp.eu/~gibson/Teaching/Teaching-ReadingMaterial/BeckFowler99.pdf. Last retrieved 15-02-2022.

[9] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. 2nd. ed. Addison-Wesley, 2018. ISBN: 9780134757681.

[10] github. *codeql*. URL: https://github.com/github/codeql/blob/main/csharp/ql/src/Bad\%20Practices/EmptyCatchBlock.ql. git hash: daa45322b1cc2fee7623537a376d64

[11] Microsoft. *Overview of .NET source code analysis*. URL: https://docs.microsoft.com/da-dk/dotnet/fundamentals/code-analysis/overview. Last retrieved 05-06-2022.

[12] Daniél Garrido-Y Martinez Nielsen et al. "Energy Benchmarking With Doom". In: (2021). URL: https://projekter.aau.dk/projekter/da/studentthesis/energy-benchmarking-with-doom(54556b3b-5728-493c-8b11-6ac67a4da5fb).html.

[13] Wellington Oliveira et al. "Recommending Energy-Efficient Java Collections". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 160–170. DOI: 10.1109/MSR.2019.00033.

[14] *Sample size in statistics*. Accessed: 2022-03-29. URL: https://www.statisticshowto.com/probability-and-statistics/find-sample-size/.

[15] Arman Shehabi et al. *United States Data Center Energy Usage Report*. Tech. rep. June 2016.

[16] SonarQube. *SonarQube Custom rule documentation*. https://docs.sonarqube.org/latest/extend/adding-coding-rules/. 2021.

[17] Testim. *What Is a Linter? Here's a Definition and Quick-Start Guide*. https://www.testim.io/blog/what-is-a-linter-heres-a-definition-and-quick-start-guide/. 2021.

Appendix A

Code smell snippets

A.1 Long Method

```
public override void Compute()
{
    Random rand = new Random();

    arr = new int[ARR_SIZE];

    for (int i = 0; i < ARR_SIZE; i++) arr[i] = rand.Next(50);

    for (int i = 0; i < ARR_SIZE - 1; i++) {
        int min = i;
        for (int j = i + 1; j < ARR_SIZE; j++)
            if (arr[j] < arr[min])
                min = j;

        int temp = arr[min];
        arr[min] = arr[i];
        arr[i] = temp;
    }

    int total = 0;
    for (int i = 0; i < ARR_SIZE; i++) {
        total += arr[i];
    }

    int upper;
    int lower;
    for (int i = 0; i < ARR_SIZE / 2; i++) {
        lower = arr[i];
        upper = arr[ARR_SIZE - i - 1];
        arr[i] = upper;
        arr[ARR_SIZE - i - 1] = lower;
    }

    int x = 5;
    for (int i = 0; i < ARR_SIZE; i++) {
        arr[i] = arr[i] * x;
    }

```

```
public override void Compute() {
    CreateArray();
    BubbleSort();
    SumArray();
    InvertArray();
    MultiplyArray();
    arr = null;
}

public void CreateArray()
{
    Random rand = new Random();

    arr = new int[ARR_SIZE];

    for (int i = 0; i < ARR_SIZE; i++) arr[i] = rand.Next(50);
}

public void BubbleSort()
{
    for (int i = 0; i < ARR_SIZE - 1; i++) {
        int min = i;
        for (int j = i + 1; j < ARR_SIZE; j++)
            if (arr[j] < arr[min])
                min = j;

        int temp = arr[min];
        arr[min] = arr[i];
        arr[i] = temp;
    }
}

public void SumArray()
{
    int total = 0;
    for (int i = 0; i < ARR_SIZE; i++)
        total += arr[i];
}

public void InvertArray()
{
    int upper;
    int lower;
    for (int i = 0; i < ARR_SIZE / 2; i++) {
        lower = arr[i];
        upper = arr[ARR_SIZE - i - 1];
        arr[i] = upper;
        arr[ARR_SIZE - i - 1] = lower;
    }
}

public void MultiplyArray()
{
    int x = 5;
    for (int i = 0; i < ARR_SIZE; i++) {
        arr[i] = arr[i] * x;
```

A.2 Redundant Storage of Data

```
public override int[] ReverseList()
{
    int[] arr = {12, 42, 54, 1, 29, 390, 2, 39, 5849, 30, 1034, 439, 3228, 20, 392,
    ↪  4832, 203, 39, 3489, 498, 304, 32, 4930, 849, 182, 3892, 483, 37, 19, 93,
    ↪  83};

    int upper;
    int lower;
    for (int i = 0; i < arr.Length / 2; i++) {
        lower = arr[i];
        upper = arr[arr.Length - i - 1];
        arr[i] = upper;
        arr[arr.Length - i - 1] = lower;
    }

    return arr;
}

public override int[] SortList()
{
    int[] arr = {12, 42, 54, 1, 29, 390, 2, 39, 5849, 30, 1034, 439, 3228, 20, 392,
    ↪  4832, 203, 39, 3489, 498, 304, 32, 4930, 849, 182, 3892, 483, 37, 19, 93,
    ↪  83};

    for (int i = 0; i < arr.Length - 1; i++) {
        int min = i;
        for (int j = i + 1; j < arr.Length; j++)
            if (arr[j] < arr[min])
                min = j;

        int temp = arr[min];
        arr[min] = arr[i];
        arr[i] = temp;
    }

    return arr;
}

public override int SumList()
{
    int[] arr = {12, 42, 54, 1, 29, 390, 2, 39, 5849, 30, 1034, 439, 3228, 20, 392,
    ↪  4832, 203, 39, 3489, 498, 304, 32, 4930, 849, 182, 3892, 483, 37, 19, 93,
    ↪  83};

    int total = 0;
    for (int i = 0; i < arr.Length; i++)
        total += arr[i];

    return total;
}
```

Snippet A.2.1: Redundant Storage of Data Code Smell

```
private int[] arr = {12, 42, 54, 1, 29, 390, 2, 39, 5849, 30, 1034, 439, 3228, 20,
↪  392, 4832, 203, 39, 3489, 498, 304, 32, 4930, 849, 182, 3892, 483, 37, 19, 93,
↪  83};
public override int[] ReverseList()
{
    int upper;
    int lower;
    for (int i = 0; i < arr.Length / 2; i++) {
        lower = arr[i];
        upper = arr[arr.Length - i - 1];
        arr[i] = upper;
        arr[arr.Length - i - 1] = lower;
    }

    return arr;
}

public override int[] SortList()
{
    for (int i = 0; i < arr.Length - 1; i++) {
        int min = i;
        for (int j = i + 1; j < arr.Length; j++)
            if (arr[j] < arr[min])
                min = j;

        int temp = arr[min];
        arr[min] = arr[i];
        arr[i] = temp;
    }

    return arr;
}

public override int SumList()
{
    int total = 0;
    for (int i = 0; i < arr.Length; i++)
        total += arr[i];

    return total;
}
```

Snippet A.2.2: Redundant Storage of Data, refactored