Inheritance versus Delegation — which is more energy efficient?

Déaglán Connolly Bree
School of Computer Science
University College Dublin
Ireland

deaglan.connolly-bree@ucdconnect.ie

Mel Ó Cinnéide
School of Computer Science
University College Dublin
Ireland
mel.ocinneide@ucd.ie

ABSTRACT

Energy consumption of software is receiving more attention as concerns regarding climate change increase. One factor that significantly impacts how much energy is expended by a software application is the design of the software itself. Existing studies find few consistent results regarding the impact of common refactorings on energy consumption, nor do they define a concrete set of metrics that measure the energy efficiency of software. In this paper, we present the results of preliminary experiments that explore the Replace Inheritance with Delegation refactoring, and its inverse, to assess the impact these design-level refactorings have on energy consumption in the Java programming language. In the tested programs, inheritance proved to be more energy efficient than delegation, with a reduction in run time of 77% and a reduction in average power consumption of 4%. We subsequently propose a research plan to further explore this problem and observe a number of specific challenges in this area. The primary goals of this research are threefold: (i) to investigate how redundancy in an object-oriented design can contribute to unnecessary energy consumption, (ii) to determine how refactoring of the software can remove this redundancy, and (iii) to develop a general-purpose automated tool to perform this refactoring.

KEYWORDS

Software Refactoring, Energy-Aware Software, Code Smells, Design Patterns, Electricity Consumption

ACM Reference Format:

Déaglán Connolly Bree and Mel Ó Cinnéide. 2020. Inheritance versus Delegation — which is more energy efficient?. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20)*, May 23–29, 2020, Seoul, Republic of Korea. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3387940.3392192>

1 INTRODUCTION

Refactoring is usually undertaken to improve the clarity of software, and to make the software easier to maintain and evolve. Recent interest in reducing the amount of energy consumed by software has brought attention to the idea that refactoring may also serve to

make software more energy efficient. Research into mobile technology remains the focus of refactoring to reduce energy consumption. The reason for this is twofold: (i) battery life remains a precious commodity for the everyday user in the mobile world, and (ii) the growth of Internet of Things (IoT) calls for an increased longevity of small computing devices.

There are other motivations for reducing energy consumption. Cost may be an important factor. For example, it is estimated that data centres consumed up to 4.8% of Ireland's electricity supply in 2016 [9]. Thus, a reduction in energy consumption by single percentage points globally has the potential to create large cost saving benefits.

Furthermore, as the concerns about climate change increase, so too does the demand to reduce energy consumption wherever this is possible. Reports on the proportion of electricity the ICT (Information Communication Technology) sector will consume in the future vary. Andrae and Edler [2] propose a worst-case scenario where the ICT sector will consume 51% of the global electricity supply by 2030, and account for 23% of global greenhouse gas emissions. A study completed by Malmödin and Lundén [11] provided a more positive perspective, finding that the energy consumption of the ICT sector has stopped growing, and may reduce further in the future. They suggest the reduction is partly due to improved energy efficiency in IoT and mobile devices, and to improved energy efficient displays [11]. Nonetheless, the Irish electricity transmission operator EirGrid have estimated that 29% of Ireland's total electricity demand will come from data centres by 2028 [1].

The remainder of this paper is structured as follows. In Section 2 we discuss related work in the area of improving the energy efficiency of software. A proof of concept experiment is carried out in Section 3, examining the effects of two refactorings, Replace Inheritance with Delegation, and Replace Delegation with Inheritance, on the energy consumption of a small program. A research plan for the future is proposed in Section 4. We discuss a number of challenges in this area in Section 5 before providing some overall conclusions in Section 6.

2 RELATED WORK

Some research has been undertaken in the refactoring of software to reduce energy consumption, with methods ranging from altering application UI colours and introducing power saving modes on mobile platforms [5], to introducing approximate programming and Dynamic Voltage and Frequency Scaling (DVFS) [17]. Relatively few studies have investigated the effect simple refactorings have on energy consumption.

Verdecchia et al. [21] completed an empirical evaluation on the change in energy consumption following the removal of five code smells, namely Feature Envy, Type Checking, Long Method, God

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7963-2/20/05...\$15.00

<https://doi.org/10.1145/3387940.3392192>

Class, and Duplicated Code. Several software metrics, such as DIT (Depth of Inheritance Tree) and NOC (Number of Classes), were investigated but none were found to be a reliable indicator of the impact refactoring has on energy consumption. The authors noted that the removal of God Classes and Duplicated Code saw a minimal change or even an increase in run time, but a reduction average energy consumption, resulting in an overall reduction in total energy consumed.

Pérez-Castillo and Piattini [16] analysed the effect of removing God Classes on energy consumption. In contrast to the results of Verdecchia et al. [21], Pérez-Castillo and Piattini found that the removal of God classes resulted in a significant increase in run time and average energy consumption.

Sahin et al. [19] investigated the impact of various creational, structural, and behavioural design patterns on energy consumption. The introduction of certain patterns were shown to have a substantial effect on energy consumption; this is evident in the Decorator pattern which was found to increase energy consumption by approximately 700%. However, the design patterns in each of the three categories did not yield consistent impacts on energy consumption.

Bunse and Stiemi [3] also undertook research into the impact of numerous software patterns, finding an increase in energy consumption of 133% after the implementation of the Decorator pattern, in line with the large impact that Sahin et al. [19] found.

Sahin et al. [20] subsequently evaluated how six common refactorings affect energy consumption. Convert Local Method to Field, Extract Local Variable, Extract Method, Introduce Indirection, Inline Method and Introduce Parameter Object were the code smells investigated. Results showed that the refactoring of software can significantly impact its energy usage, though these results were not consistent across both Java Virtual Machine (JVM) 6 and 7. The majority of refactorings also resulted in both positive and negative impacts on energy consumption in different test cases.

da Silva et al. [6] investigated the impact of the Inline Method refactoring on energy consumption. Through several iterations, the methods most frequently invoked were inlined, and Java bytecode instruction calls were recorded. Results varied depending on the complexity of the program being refactored. A balance appeared to be reached between a reduction in messages passed, potentially reducing energy consumption, and the increased number of variables in scope, potentially increasing energy consumption.

Park et al. [15] investigated the effect of Fowler's refactorings [8] on energy consumption. These refactorings include the two we investigate: Replace Delegation with Inheritance and Replace Inheritance with Delegation. Their brief experiments found that replacing delegation with inheritance reduced energy consumption when applied to a simple C++ program. However, a power estimation tool was used to measure energy consumption, and they found the difference between the two implementations to be approximately 0.5%. Furthermore, they did not consider the program's run time when running experiments or listing results—reducing the scope of discussion on the effects of the refactorings themselves. They also found some contradictory results, such as in the case of Pull up Field, and Push Down Field, both reducing energy consumption.

Morales et al. [12] investigated a series of object-oriented anti-patterns and mobile anti-patterns on the Android platform. The

results varied: removing some anti-patterns reduced energy consumption, but removing others increased energy consumption.

Rodriguez et al. [18] also measured the impact of code smells on energy consumption, employing several mobile applications in their experiments. For each mobile application, they created one version where all code smells were removed and another version where smells were left and accessor methods were removed (to reduce energy consumption). The results highlighted a notable impact where the smell-free versions of the applications consumed up to 70% more energy when compared to the energy-optimised versions.

To conclude, we note that a number of papers have investigated the impact software design has on energy consumption by exploring how refactoring and the addition/removal of design patterns have on this property. The results are inconclusive at this stage, and in some cases, contradictory.

3 PROOF OF CONCEPT EXPERIMENTS

Many studies have been completed examining the effects several types of refactoring, code smells, and patterns have on the energy consumption of software as highlighted in Section 2. However, the field lacks a clearly defined experimental process by which to run related experiments. In order to begin experimental research in this area, and commence the development of a well-defined process, we used snippets of code based on examples taken from Fowler's refactoring textbook [8]. The refactorings we chose to study were Replace Inheritance with Delegation and Replace Delegation with Inheritance as they effectively produce two opposite implementations of the same program: one with Delegation, the other with Inheritance. Given the complexity a just-in-time (JIT) enabled compiler can introduce, discussed further in Section 5.1, these experiments were run in *interpreted mode*. This means that the bytecode is not compiled to native machine language at run-time, but is interpreted as it is executed, avoiding any optimisations that the JIT enabled compiler might perform. Because of this, the results presented here are not necessarily applicable to real-world scenarios where programs would be run with a JIT enabled compiler. The proof of concept allows for an early exploration of the effects of refactoring on energy consumption and an initial examination of potential static metrics that might correlate with energy consumption.

3.1 Instrumentation

The code snippets were run on a device with an Intel Core 2 Duo T6400 at 2.0GHz and 2.9GiB of DDR2 667MHz ram, running Ubuntu 18.04.3 LTS. A *Watts Up Pro* power meter¹ was used to record power consumption every second. The programs were compiled and run in Java version 1.8.

3.2 Experiment Design and Execution

Having developed the code snippets as described in Listings 1 and 2, two near identical `Main` classes were made. They instantiated the two implementations of the `Employee` class and subsequently invoked the `getName` method 1×10^8 times. The large number of method invocations allowed the program to run for a longer period

¹https://www.powermeterstore.com/p1206/watts_up_pro.php

of time, during which we could collect power readings. The longer run times also reduced the potential impact of noise in the power readings over multiple iterations of the experiment. The time was printed just before the loop began, and again immediately after allowing for power readings to be noted in relation to the time the programs were being run.

All unnecessary programs were closed and power data was recorded for at least 30 minutes before experiments began in order to measure idle power consumption. The programs were compiled normally, and executed with the `-Xint` argument, causing them to be executed in interpreted mode. The programs were run 25 times to reduce any potential noise or outliers in the results.

In gathering results, the power measurements were compared against the timing of the programs to calculate the run time of each program (seconds), average power consumption (watts), and energy consumption (joules). The watt and the joule are both derived units of energy in the International System of Units, commonly known as SI units. Some related studies measure energy consumption in terms of watt-hours, which is not an SI unit. For this reason, it is appropriate to measure energy consumption in terms of joules.

3.3 Results and Discussion

The two versions of the program, inheritance- and delegation-based, were both run 25 times to reduce the potential of outliers in the recorded results. As noted in Table 1, the implementation with Inheritance (Replace Delegation with Inheritance, RDI) resulted in a 77% and 4% reduction in run time and power consumption respectively, when compared with the implementation using Delegation (Replace Inheritance with Delegation, RID). This led to a reduction in overall energy consumption of 81%.

The difference in the run time of the two versions of the program accounts for the majority of the reduction in energy consumption. The difference in average power consumption being relatively minimal also highlights the close relation between run time and total energy consumption. This is to be expected as energy is calculated as: $Energy_{Joules} = Power_{Watts} * Time_{Seconds}$, thus the amount of energy a program consumes is directly proportional to its run time.

While consideration must be made for the variation in power consumption at run-time, it appears to remain relatively stable after an initial start-up period as highlighted in Figure 1 and 2. The change in power consumption is most substantial in the first 4 seconds of the program running. This may be due to the JVM starting up, and also due to a lack of accuracy in time measurement as discussed in Section 3.4.2. As the results in Table 1 are relative, the time taken to reach a consistent power consumption has little impact as both versions of the program experience this lag at start-up.

The reduction in run time is an expected result. This is due to the increased number of method invocations, and so, an increased number of bytecode instructions in the Delegation version of the program. While the number of method invocations in the Delegation version effectively doubles, the run time does not. This highlights some complexity in the Java Virtual Machine at run-time regarding inherited classes and the invocation of their methods.

```

1 public class Person {
2     public Person(String name) {
3         this.name = name;
4     }
5
6     public String getName() {
7         return name;
8     }
9
10    private String name;
11 }
12
13 public class Employee {
14     public Employee(String name) {
15         person = new Person(name);
16     }
17
18     public String getName() {
19         return person.getName();
20     }
21
22     private Person person;
23 }

```

Listing 1: Java implementation of the Delegation version of the program.

```

1 public class Person {
2     public Person(String name) {
3         this.name = name;
4     }
5
6     public String getName() {
7         return name;
8     }
9
10    private String name;
11 }
12
13 public class Employee extends Person {
14     public Employee(String name){
15         super(name);
16     }
17 }

```

Listing 2: Java implementation of the Inheritance version of the program.

Table 1: Average results of Replace Inheritance with Delegation (RID) and Replace Delegation with Inheritance (RDI) for n=25. Idle power consumption is 18.8W.

	RID	RDI	Difference
Run time (Seconds)	26.76	15.04	77.93%
Power Consumption (Watts)	30.32	28.93	4.8%
Energy Consumption (Joules)	841.66	464.08	81.36%

Power Consumption vs Time with Inheritance

Figure 1: Power consumption over time for a selection of inheritance-based program runs.

While reducing the run time of a program is clearly an effective method to reduce energy consumption, the scale of the impact of replacing delegation with inheritance may be reduced heavily in practice due to the impact of a JIT enabled compiler. Further research should be carried out into the effect of replacing Delegation with Inheritance in open-source software to test the impact in a real-world, JIT compiler enabled scenario.

The reduction in power consumption is a more complex result to describe. The variation may be caused by the limitations of the experimental setup as highlighted in Section 3.4. The lag in power consumption readings at start-up, combined with the substantial difference in run times are the likely causes. However, it does pose a question regarding the potential energy cost difference of individual bytecode instructions. A difference in power consumption here could also arise from the different ways in which the Java Virtual Machine handles the search, and execution of inherited methods, as opposed to the execution of methods in delegated objects. A similar concept was hypothesised by da Silva et al. [6] when accounting for the variation in power consumption for longer methods, which required more complex bytecode instructions. The specifics regarding the invocation of methods from inherited classes can vary between Java Virtual Machine implementations adding further complexity to the problem. The potential for varying energy performances of Java Virtual Machines, and the potential differences in the cost of bytecode operations, highlights another avenue for further research.

With a reduction in run time of over 77%, this proof of concept experiment has highlighted the impact simple refactorings have on the energy consumption of software. While the difference in power consumption of over 4% requires an improved experimental methodology to increase accuracy, and raises further questions regarding individual instruction costs, it is clear that a reduction in method invocations, and subsequently a reduced number of bytecode calls have a major impact on the energy consumption of software.

Power Consumption vs Time with Delegation

Figure 2: Power consumption over time for a selection of delegation-based program runs.

Power Consumption when Idling

Figure 3: Graph highlighting the fluctuation in power consumption over a 10 minute period of idling.

3.4 Threats to Validity

In this section we consider some of the issues that need to be considered in interpreting our results.

3.4.1 Background Electricity Consumption. Background electricity consumption presents a substantial threat to validity. If programs are running in the background, and consume varying amounts of power over time, the results may be inconsistent and inaccurate. To remedy this problem, any unnecessary user programs were closed prior to the experiments being run, and the device was placed in airplane mode to reduce potential interrupts. Some fluctuation in power consumption remains however, with near periodic changes as highlighted in Figure 3. These changes in power consumption appear consistent, though not perfectly so with some power consumption peaks lasting longer than others. The device was left to idle before experiments to calculate idle power consumption, and to ensure no substantial irregularities existed in the background consumption.

The question also arises regarding how long it takes for power consumption to return to idle levels after the program terminates when running experiments consecutively. To maintain continuity,

the programs were re-run 30 seconds after the previous program had terminated. The experiments were run 25 times to reduce the impact of any background noise.

3.4.2 Experimental Timing. The potential exists for inaccuracies to be introduced when recording the results of completed experiments. The time at which a program runs may not be perfectly synchronised with the timing of the power readings on the *Watts Up Pro* power monitor. While all clocks were synchronised during experiments, their accuracy was approximately to the second.

The resolution of the *Watts Up Pro* power meter is only one measurement per second. This, combined with the resolution of timing, allows for some amount of latitude in the start and end times of experiments. While this would be an issue if we wish to measure the absolute energy consumption of a program, the results are relative in these experiments. Thus, given that the potential inaccuracies apply to all experiments run, they are unlikely to play a major factor in the relative differences described in Table 1.

3.4.3 Power Measurement Accuracy. Measuring the amount of power an individual application consumes is a complex task, and so a threat to the validity to any experiments carried out, as has been noted in related research [14–16, 20].

The tools used to measure power consumption vary widely in related work. Applications like *PowerTutor* [3] are frequently used on mobile platforms, with virtualisation and simulation (e.g. CPU Cycle Measurements [10], *XEMU* [15]) sometimes used for embedded and mobile systems respectively. Physical tools are frequently deployed, with various types of low level measurement tools such as the *TipPie Handyscope HS5* [12], *MSoon* power monitor [18], and *National Instruments USB-6215 DAQ* card [20] being used, while higher level tools such as the *Voltcraft Energy Logger 4000* [16] and *Watts Up Pro* power meter [21] are also employed.

The use of software solutions to measure the energy consumption of applications is a simple and appealing approach but raises the potential for inaccurate results. Fahad et al. [7] compared the accuracy of on-chip power sensors and energy predictive models against physical measurements using a *Watts Up Pro* power meter calibrated with a Yokogawa WT310, and saw varying error rates between the reported on-chip sensors and the ground truth ranging from 8% to 73%.

A *Watts Up Pro* was used in these experiments, with a reported $\pm 1.5\%$ accuracy, and a sampling rate of one reading per second. However, the power meter used had not been calibrated with, nor tested against a certified ANSI C12.20 standard power meter as was carried out by Fahad et al. [7] in their research.

While inaccurate measurements are undesirable, these results are relative, thus any inaccuracies should have little to no impact as long as the experiments are performed a sufficient number of times. Nonetheless, a certified or calibrated power meter could be used in further experiments to reduce any potential inaccuracies in measuring power consumption and the use of a power monitor with a greater resolution would also assist in improving the accuracy of timing.

4 RESEARCH PLANS

Based on the related work described in Section 2, we can conclude that there is minimal consensus on the specific positive or negative impact that code smells, software patterns, or software refactorings have on the energy consumption of software, beyond the fact that they do have some impact on consumption. In order to understand this better, we propose a research plan to extend the research in this field, and to clarify how refactorings can affect energy consumption. Our proposed research plan is as follows:

- A. Apply individual refactorings to code snippets to gauge how energy consumption is affected and formalise an experimental method. While some research makes use of well-known refactorings, such as Park et al. [15] employing Fowler’s Refactorings [8], there may be other, lesser known types of refactorings that have an impact on energy consumption.
- B. Attempt to develop a number of metrics that can be used to measure the energy consumption of a program in a dynamic and/or static manner.
- C. Using these metrics, investigate the possibility of using automated search-based refactoring to explore the range of possible designs for a given program, in order to find one with a better energy consumption profile. Experiments will be carried out with open source software to investigate the impact on energy consumption in a real-world scenario.
- D. Develop a tool to automatically refactor software to reduce energy consumption. This is envisaged to fit in the release toolchain just prior to the deployment of the software.

While our early proof of concept experiments described in Section 3 are run in Java, any techniques that are found to reduce energy consumption should be investigated in other object-oriented programming languages, such as C++, Python, and Ruby, to expose any generalisations that can be made.

5 CHALLENGES

While code smells may be an indicator of poor programming practice in object-oriented programming, there is no evidence to suggest that the removal of all traditional code smells should result in a reduction in energy consumption. The opposite is sometimes true [16, 18]. We hypothesise that the practices followed in typical object-oriented design, where clear structure, reusability of code, and long-term maintenance are key, results in greater redundancy and unnecessary energy consumption. Thus, object-oriented design may be the antithesis of energy-aware programming.

Given this hypothesis, it is worthwhile to continue investigation into commonly employed software patterns. Those patterns which result in a substantial increase in energy consumption could be removed from existing software retroactively, or avoided in future. However, this would have a negative impact in the future, as software engineers would no longer have the flexibility and reusability of those frequently used patterns.

This highlights the issue whereby in order to be energy-aware, software engineers need to reduce the quality of their object-oriented programming. Alternatively, having written software, software engineers could refactor it manually to remove known energy-specific

code smells. This is unlikely to be feasible in practice where software engineers in industry already struggle to write adequately maintainable, and tested code, all the while under time pressure.

Instead, we have proposed the development of a general-purpose automated refactoring tool to edit software in order to make it more energy efficient. When included in the toolchain, it would allow for software engineers to develop software as normal, following traditional object-oriented programming techniques. Once their completed code is pushed to production, it would be refactored with this tool to reduce energy consumption in deployment.

The threats to validity highlighted in Section 3.4 are not the only complications this research faces in the future. While many more are likely to appear, we discuss three challenges facing the field in the near future, and how they can be approached.

5.1 Programming Language Intricacies

Many previous studies make use of Java in their experiments [3, 6, 12, 16, 18, 20, 21]. One feature of Java is that it uses the Java Virtual Machine (JVM) for portability between varying hardware configurations and operating systems. The inclusion of a just-in-time (JIT) compiler, as is the case with the HotSpot™ JVM, improves performance through run-time compilation of bytecode into native machine code and by profiling the code at run-time to optimise it on the fly in various ways including inlining methods and eliminating dead code [13].

The introduction of JIT compilation has hugely beneficial impacts on run-time performance, although it poses a potential challenge when performing experiments. When running short snippets of sample code, which may not use the data returned from methods, or may repeat the same task many times to run for a longer period of time, smart profiling can result in the removal of unused variables and whole operations themselves. While applications could be run in interpreted mode only, as was undertaken in Section 3, these experiments may have substantially different results if the same refactorings were applied in a real-world scenario with a JIT compiler enabled.

The refactoring of some open-source software with clearly defined inputs and outputs provides the clearest solution to this program. The JIT compiler would act as it would in production, however the specific impacts of the refactoring would remain difficult to calculate as more run-time complexities come into play such as the number of times the refactored code is actually run, and whether the bytecode is altered by the compiler.

5.2 Application of Refactorings

The application of some refactorings can be clear cut, however in practice, a line must be drawn by software engineers when developing, or maintaining software, as to where to stop refactoring. Attempting to automate the refactoring of software to reduce energy consumption poses similar questions: How many refactorings should be made? And where should they be applied?

The answers are unlikely to be trivial. While some previous studies take a code smell and refactor until that code smell is no longer present in the code such as the removal of all God Classes [16], this may not work in a real-world scenario. As discussed previously, there is little reason to suggest that object-oriented code

smells are equally “negative” from an energy-aware perspective. In the case of God Classes, Pérez and Castillo report the opposite [16].

The introduction or removal of design patterns may provide a simple opportunity to reduce energy consumption. Sahin et al. [20] observed that the introduction of the Decorator and Observer patterns increased consumption by 99% and 38% respectively while the introduction of the Flyweight and Proxy patterns resulted in a reduction in energy consumption by 53% and 38% respectively. Those design patterns that increase energy consumption have the potential to be named energy-oriented code smells, and their complete removal may positively impact energy consumption. Further research must be carried out on real software to clarify this hypothesis.

da Silva et al. [6] iteratively inlined the most frequently invoked methods of an application and saw energy consumption drop before increasing again for later iterations. This was potentially explained by a change in method size and an increased number of complex bytecode instructions. This highlights the subtlety when refactoring and calls for some set of dynamic, and optimally, static metrics.

5.3 Development of Metrics

The development of a set of metrics is a fundamental step to systematically, and later automatically, refactor software in order to reduce energy consumption. While some attempts have been made to clarify which common software metrics are related to power consumption, the results vary. Verdecchia et al. [21] did not find any consistent and significant correlation between software metrics described by Chidamber and Kemerer [4] and energy consumption. Pérez-Castillo and Piattini [16] and Sahin et al. [19] measured some static metrics such as the Number of classes and methods, and dynamic metrics such as the Number of Messages Sent and Number of Failures, but found no clear relation between those metrics and energy consumption either. Finally, da Silva et al. [6] tracked the number of times major bytecode instructions were called. da Silva et al. suggested a relation between more complex bytecode instructions and increased energy consumption, highlighting a potential dynamic metric, in terms of bytecode, and a potential static metric, in terms of method size.

It is clear that there is substantial complexity when attempting to describe the relation between most metrics and energy consumption. Further research will need to be undertaken, if not to define metrics that are clearly correlated with energy consumption, to begin describing the relationship between groups of metrics and energy consumption.

6 CONCLUSION

In this paper we have investigated the ability of refactoring to reduce energy consumption. While research in this area is growing, the methods used vary greatly as highlighted in Section 2, and the field in general remains in its infancy.

We have undertaken a proof of concept experiment to compare the impact of two refactorings on the energy consumption of a program. Replace Delegation with Inheritance and its inverse, Replace Inheritance with Delegation, were chosen as they are design-level

refactorings that represent a common design decision that a developer makes: to use either inheritance or delegation to share functionality between two classes. Two versions of the same program were written, one implemented with inheritance, the other with delegation. The version of the program making use of inheritance saw a reduction in run time by over 77%, and its power consumption reduced by over 4% when compared to its delegation-based counterpart. These experiments were run in interpreted mode, which lessens the generalisation of the results to real systems in production. It does however highlight the potential impact this kind of refactoring could have in practice and poses more questions regarding the improvement of the run time and power consumption of software.

In terms of future research plans, we proposed four major waypoints: the application of simple refactorings to investigate their impacts on software, the development of a set of metrics to assess the code in terms of energy consumption, the application of refactorings to working software to test the applicability of the metrics in production, and lastly, the development of a general-purpose, search-based automated refactoring tool to perform those refactorings.

Refactoring to improve energy consumption is a challenging problem, as while the refactorings take place in the design space of the program, the energy is consumed when the compiled version of program executes in a particular hardware configuration. In spite of the uncertainty in the relationship between the design and execution spaces, our experiments showed that the Replace Delegation with Inheritance reliably yielded shorter run times and reduced energy consumption. Future work will determine if this can be developed further into a practical approach to using automated refactoring to reduce energy consumption.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of the Irish Research Council through a Government of Ireland Postgraduate Scholarship GOIPG/2019/3779, and the support of Lero - the Irish Software Research Centre.

REFERENCES

- [1] 2019. Eirgrid All-Island Generation Capacity Statement 2019-2028. (2019).
- [2] Anders Andrae and Tomas Edler. 2015. On Global Electricity Usage of Communication Technology: Trends to 2030. *Challenges* 6, 1 (April 2015), 117–157. <https://doi.org/10.3390/challe6010117>
- [3] Christian Bunse and Sebastian Stiemer. 2013. On the Energy Consumption of Design Patterns. *Softwaretechnik-Trends* 33, 2 (May 2013), 4–5.
- [4] S. R. Chidamber and C. F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20, 6 (June 1994), 476–493. <https://doi.org/10.1109/32.295895>
- [5] Luis Cruz and Rui Abreu. 2019. Catalog of Energy Patterns for Mobile Applications. *Empirical Software Engineering* 24, 4 (Aug. 2019), 2209–2235. <https://doi.org/10.1007/s10664-019-09682-0>
- [6] Wellisson GP da Silva, Lisane Brisolara, Ulisses B Corrêa, and Luigi Carro. 2010. Evaluation of the Impact of Code Refactoring on Embedded software Efficiency. In *Proceedings of the 1st Workshop de Sistemas Embarcados*. 145–150.
- [7] Muhammad Fahad, Arsalan Shahid, Ravi Reddy Manumachu, and Alexey Lastovetsky. 2019. A Comparative Study of Methods for Measurement of Energy of Computing. *Energies* 12, 11 (June 2019), 2204. <https://doi.org/10.3390/en12112204>
- [8] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.
- [9] Host in Ireland. 2017. Ireland’s Data Hosting Industry 2017. (2017).
- [10] Andreas Litke, Kostas Zotos, Alexander Chatzigeorgiou, and George Stephanides. 2005. Energy Consumption Analysis of Design Patterns. In *Proceedings of the International Conference on Machine Learning and Software Engineering*. Citeseer, 86–90.
- [11] Jens Malmodin and Dag Lundén. 2018. The Energy and Carbon Footprint of the Global ICT and E&M Sectors 2010-2015. *Sustainability* 10, 9 (Aug. 2018), 3027. <https://doi.org/10.3390/su10093027>
- [12] Rodrigo Morales, Ruben Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. 2016. Anti-patterns and the Energy Efficiency of Android applications. *arXiv:1610.05711 [cs]* (Oct. 2016). <http://arxiv.org/abs/1610.05711> arXiv: 1610.05711.
- [13] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot™ Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*. USENIX, 1–12.
- [14] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2019. On the Impact of Code Smells on the Energy Consumption of Mobile Applications. *Information and Software Technology* 105 (Jan. 2019), 43–55. <https://doi.org/10.1016/j.infsof.2018.08.004>
- [15] Jae Jin Park, Jang-Eui Hong, and Sang-Ho Lee. 2014. Investigation for Software Power Consumption of Code Refactoring Techniques. In *SEKE*. 717–722.
- [16] R. Pérez-Castillo and M. Piattini. 2014. Analyzing the Harmful Effect of God Class Refactoring on Power Consumption. *IEEE Software* 31, 3 (May 2014), 48–54. <https://doi.org/10.1109/MS.2014.23>
- [17] G. Pinto, F. Soares-Neto, and F. Castor. 2015. Refactoring for Energy Efficiency: A Reflection on the State of the Art. In *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*. 29–35. <https://doi.org/10.1109/GREENS.2015.12>
- [18] Ana Rodriguez, Matias Longo, and Alejandro Zunino. 2015. Using Bad Smell-driven Code Refactorings in Mobile Applications to Reduce Battery Usage. In *Simpósio Argentino de Ingeniería de Software (ASSE 2015)-JAIIO 44 (Rosario, 2015)*.
- [19] Cagri Sahin, Furkan Cayci, Irene Lizeth Manotas Gutierrez, James Clause, Fouad Kiamilev, Lori Pollock, and Kristina Winbladh. 2012. Initial Explorations on Design Pattern Energy Usage. In *2012 First International Workshop on Green and Sustainable Software (GREENS)*. IEEE, Zurich, Switzerland, 55–61. <https://doi.org/10.1109/GREENS.2012.6224257>
- [20] Cagri Sahin, Lori Pollock, and James Clause. 2014. How do Code Refactorings Affect Energy Usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '14*. ACM Press, Torino, Italy, 1–10. <https://doi.org/10.1145/2652524.2652538>
- [21] Roberto Verdecchia, René Aparicio Saez, Giuseppe Procaccianti, and Patricia Lago. 2018. Empirical Evaluation of the Energy Impact of Refactoring Code Smells. In *ICT4S*. 365–383.