Empirical Software Engineering (2023) 28:145
https://doi.org/10.1007/s10664-023-10387-8

Energy efficiency of the Visitor Pattern: contrasting Java and C++ implementations

Déaglán Connolly Bree[1] · **Mel Ó Cinnéide**[1]

Accepted: 24 August 2023 / Published online: 28 October 2023

Abstract
Design patterns are applied frequently during software evolution in order to make the software more flexible and amenable to extension. One little-studied aspect of design patterns is their propensity to increase run-time energy consumption due to the indirection and additional structure they introduce. In this paper we study the impact of the Visitor pattern on energy efficiency. The Visitor pattern separates an algorithm from the objects it acts upon and improves maintainability by placing each algorithm within a single visitor class. This is at the cost of increased indirection due to the double dispatch required when the algorithm is invoked. We experimentally investigate the energy impact of varying the implementation of this pattern, and of removing the pattern entirely from software written in Java and C++. In our results we observe energy consumption reductions greater than 7% in a Java-based textbook example when the pattern is implemented using reflective dispatch, and reductions of over 10% when experimenting with an open source Java project, JavaParser. The complete removal of the pattern yields more complex results, with little impact in the textbook example but reductions of over 7% in the JavaParser study. To explore the generalisability of our findings, we subsequently apply the same transformations to the C++ based CppParser. Total pattern removal here sees energy consumption reductions of over 66% while the reflective dispatch approach increases energy consumption by up to 2012%. Our results highlight the energy savings that can be achieved when the Visitor pattern is removed both in Java and C++ implementations, and also show that some language specific features can allow for further energy savings when the implementation of the pattern is varied.

Keywords Empirical · Design patterns · Visitor pattern · Green technology · Energy efficiency · Refactoring · Software transformation · Software design · Object-Oriented programming

Communicated by: Paris Avgeriou and Dave Binkley

This article belongs to the Topical Collection: *Software Maintenance and Evolution (ICSME)*

✉ Déaglán Connolly Bree
deaglan.connolly-bree@ucdconnect.ie

Mel Ó Cinnéide
mel.ocinneide@ucd.ie

[1] School of Computer Science, University College Dublin, Dublin, Ireland

1 Introduction

Software consumes energy when it executes. While this has received limited attention from researchers in the past, the severity of the climate crisis and the consequent rise in energy costs (Avgerinou et al. 2017; Flucker and Tozer 2013), coupled with the growth of mobile technology and the burgeoning number of energy-hungry data centres, has led to a heightened awareness of the energy performance of software (Pinto et al. 2015; Malmodin and Lunden 2018; Andrae and Edler 2015).

Source code refactoring is a common practice in software maintenance (Fowler et al. 1999). It is employed to improve software quality, and it is not uncommon that the goal of refactoring encompasses the application of a design pattern (Kerievsky 2005).

Design patterns are solutions that are commonly used to solve recurring software problems in a certain context. While the notion was originally developed by Christopher Alexander (1977) in the context of the architecture of living spaces, design patterns subsequently generated an enormous impact on the software community. The seminal textbook in this area, that of Gamma et al. (1995), describes design patterns in a software context and catalogues a number of commonly used patterns. Although written in 1995, this book remains a bestseller in the software field (Amazon 2022).

Design patterns are also prime examples of good object-oriented programming practice in that they "identify, name, and abstract common themes in object-oriented design" (Gamma et al. 1993). They provide modularity, extensibility, and encapsulation and inheritance are frequent features of their designs. Design patterns are therefore of particular interest in the exploration of the potential impact of object-oriented programming on energy efficiency. While the encapsulation and indirection introduced by patterns may improve maintainability, there may also be a subsequent energy cost at run-time.

The enduring popularity of design patterns in the software community indicates how pervasive pattern thinking is. In spite of this popularity, the energy implication of design patterns has not been heavily investigated. This provides motivation to explore design patterns and their impact on energy efficiency.

Additionally, while the context in which any design pattern is used can vary, the overall structure of the solution tends to involve a core set of pattern features which may allow for a systematic approach to their removal. This provides further motivation for our research — if design patterns are indeed found to be energy sinks, the process of removing them should be at least partially automatable.

In this paper we investigate the energy implications of one rich design pattern, namely Visitor (Gamma et al. 1995). The Visitor pattern is a behavioural pattern that separates an algorithm from the classes of the objects upon which it acts. This is useful in situations where the algorithm acts upon objects in a data structure that are of differing concrete types.

Rather than the algorithm being spread amongst methods in those classes, it is centralised in a single visitor class, an instance of which is passed to each object in the structure, and each object subsequently invokes its relevant method in the visitor.

The Visitor pattern aims to improve software maintainability by facilitating the implementation of new algorithms, at the cost of increased indirection as double dispatch is one of its prominent features. In most programming languages, including Java and C++ which are used in this study, multiple dispatch is not natively supported, requiring emulation through the use of multiple single dispatches. In the case of the Visitor pattern, each time a visitor visits an element, two method invocations occur which may impact energy efficiency, making the pattern a suitable candidate for further investigation.

This paper is an extended version of our work published in the 38th International Conference on Software Maintenance and Evolution (Connolly Bree and Ó Cinnéide 2022). The original study investigated the energy efficiency of two Java implementations of the Visitor pattern, the first being a proof-of-concept experiment with a textbook example, the second involving more complete experiments with a medium-sized open source application. In extending this paper, we present an extensive experimental investigation of the energy efficiency of a C++ implementation of the pattern, mirroring the case study performed in Java with an equivalent open source application written in C++. With the resulting experimental data we have expanded our discussion, alleviated some of the threats to the validity of our findings, and provided further discussion and clarification of our conclusions. We have also updated related work with additional studies published in this research area.

In our empirical investigations, we experiment with Java and C++. Both are high-level languages that support object-oriented programming, but provide different deployment strategies.

The remainder of this paper is as follows. In Section 2 we study the existing literature in this area, and discuss the Visitor pattern in greater detail in Section 3. Our experimental approach is described in Section 4 before examining the energy impact of the Visitor pattern in a Java textbook example in Section 5. Subsequently, two open source examples in Java and C++ respectively are analysed in Sections 6 and 7. We discuss our findings in Section 8 and highlight threats to validity in Section 9. Lastly, future work is discussed in Section 10 before we present our final conclusions in Section 11.

2 Related Work

While there has been a growing focus on the energy efficiency of software, the research area remains small. Given the growth of mobile computing, many studies focus on less generalisable, mobile oriented changes that can be made such as using dark UI colours or avoiding binding resources too early (Couto et al. 2020; Rodriguez et al. 2012; Cruz and Abreu 2019; Li and Halfond 2014; Ayala et al. 2019).

Pinto et al. (2015) investigated the state of the art in terms of refactoring for energy efficiency and identified concurrent/parallel programming, approximate programming, dynamic voltage and frequency scaling, in addition to mobile computing, as key areas of existing research. Georgiou et al. (2019) has categorised some research in this area and provided a high-level overview of the field in terms of software transformations that have been investigated and the tools used to measure energy consumption. However, additional work focusing on the impact of higher-level software changes, such as refactorings, code smells, and design patterns, on energy efficiency has been published which we discuss below.

Sahin et al. (2012) studied the impact of 15 design patterns on energy consumption in C++. They used short samples of code sourced online and found the implementation of the Composite, Abstract Factory, Observer, and Decorator patterns increased energy consumption by 5%, 21%, 62%, and 712%, respectively; they also found the implementation of the Visitor, Mediator, Proxy, and Flyweight patterns resulted in a reduction in consumption by 7%, 9%, 36%, and 58% respectively.

Noureddine and Rajan (2015) studied 14 design pattern examples in C++ and 7 in Java sourced online and found the Observer, Decorator, and Mediator patterns to increase the energy consumption by 30.63%, 12.24%, and 26.61% respectively. They found conflicting results to that of Sahin et al. (2012) in the case of the Composite pattern, where Sahin et al.

found its implementation increased energy consumption, and in the cases of the Visitor and Proxy patterns, Sahin et al. found their implementations to reduce energy consumption—the opposite of the findings of Noureddine and Rajan.

Bunse and Stiemer (2013) measured the impact of six design patterns in simple Android applications and found the Abstract Factory, Prototype, and Decorator patterns increased energy consumption by 14%, 33%, and 133%, respectively.

Litke et al. (2005) studied three patterns in short C++ examples but only found a notable difference in energy consumption in the case of the Observer pattern, an increase of 44%.

Feitosa et al. (2017) studied the effect of the Template Method and State/Strategy patterns on two pieces of open source software and saw the removal of the patterns reducing energy consumption of the applications by as much as 17% and 53% respectively.

Maleki et al. (2017) examined five design patterns in C++ code snippets and noted an increase in energy consumption of over 495% when the Decorator pattern was implemented, and a reduction in energy consumption of just over 4% and 49% when the Facade and Flyweight patterns were implemented respectively.

Hurbungs et al. (2022) investigated the impact of the two styles of the Singleton pattern in an IoT computing scenario and found power consumption reductions of approximately 1% when the patterns were implemented.

In previous work we found the removal of the Decorator pattern from a large, open source application reduced energy consumption by up to 5% (Connolly Bree and Ó Cinnéide 2022).

Atomic refactorings, code smells, and static metrics have also been investigated in order to highlight their relation to energy performance. We previously examined Replace Delegation with Inheritance in a preliminary study (Connolly Bree and Ó Cinnéide 2020) and noted substantial improvements in energy performance when inheritance structures are used instead of delegation. Park et al. (2014), Sahin et al. (2014) and da Silva et al. (2010) investigated various refactorings but saw mixed results. da Silva et al. (2010) highlighted the non-triviality of this type of research as they iteratively inlined the most invoked methods in a program; while they initially saw improvements in energy efficiency, they saw energy efficiency decrease again as they continued inlining methods.

The potential for code smells to positively impact energy efficiency has also been suggested. Verdecchia et al. (2018), Rodriguez et al. (2015), Pérez-Castillo and Piattini (2014), and Vetro et al. (2013) saw some code smells reduced energy efficiency, such as Feature Envy and Long Method, but also highlighted others that improved energy efficiency such as God Class and No Data Encapsulation. A study conducted by Morales et al. (2018) found the opposite in the case of God Class, potentially due to the method of smell removal, further highlighting the need for more research in this field.

While static metrics have been examined (Mancebo et al. 2021; Verdecchia et al. 2018), none have been strong indicators of energy efficiency.

There are several other avenues of research also being examined such as the energy cost of particular collection libraries (Hasan et al. 2016; Pereira et al. 2016), frameworks which assist developers improve energy efficiency (Manotas et al. 2014; Hindle 2015; Palomba et al. 2017), and the impact language selection may have on a project's energy efficiency (Pereira et al. 2021).

With the popularity of mobile computing, catalogues of android oriented smells have also been investigated from an energy consumption perspective (Tonini et al. 2013; Gottschalk et al. 2014; Hecht et al. 2016; Palomba et al. 2019; Cruz and Abreu 2017). The extent to which Java based android smells are generalisable, with respect to other languages used for mobile development such as Swift or Kotlin, or in broader contexts remains unclear.

Design patterns in other contexts have also been examined such as cloud specific patterns (Abtahizadeh et al. 2015), UI patterns (Nayak and Chandwadkar 2021), and patterns focusing on embedded systems (Menghin et al. 2015; Schaarschmidt et al. 2020) and the IoT (Crestani et al. 2021).

As aforementioned, the Visitor pattern has been studied by Sahin et al. (2012) and Noureddine and Rajan (2015). Both of these studies undertook only preliminary examinations, testing with short examples of the Visitor pattern in C++ and found contradicting evidence regarding its energy impact.

In this paper, in line with our research plans (Connolly Bree and Ó Cinnéide 2021), we also undertake an exploratory study on a short, textbook code example to clarify existing results. We subsequently expand the scope of existing research by experimentally investigating the impact of the pattern with two open source software applications, one in Java (JavaParser) and another C++ (CppParser). To our knowledge, this paper represents the first study of the energy consumption of the Visitor pattern in the context of large open source applications.

3 The Visitor Pattern

A visitor essentially encapsulates an algorithm that interacts with objects in a structure that are of different types. It contains a method for each concrete type of object it interacts with. Thus the pattern separates the algorithm from the classes it acts upon, making it easier to understand and facilitating the addition of new algorithms.

The non-pattern design approach is to spread the algorithm logic among each of the relevant concrete classes, but this leads to "algorithm sprawl" where the methods comprising the algorithm are spread across the classes of the object structure. If changes are made to the algorithm, it will be necessary to edit every class to make the appropriate alterations.

3.1 Implementation of the Visitor Pattern

The typical implementation of the Visitor pattern is described in Fig. 1. Every class to be visited (`ElementA`, `ElementB`) implements an interface[1] (`Element`) that includes the method `accept`, which takes a visitor as an argument.

Each visitor (`Visitor1`, `Visitor2`) implements an interface (`Visitor`) that includes a `visit` method for every concrete element that can be visited (e.g. `visitElementA`, etc.) that takes an instance of that element type as an argument.

The typical sequence when an element is visited is described in Fig. 2. The `accept` method is invoked on the element and the visitor is passed as an argument; the element subsequently invokes the appropriate `visit` method in the visitor, and passes itself as an argument. The visitor executes its logic which often includes accessing data through accessor methods (i.e. `getData`) in the element, and returns.

For brevity, we will usually omit the arguments of visitor-related methods in the remainder of this paper.

[1] It is not essential that the `Element` classes share a common superclass or interface—they only need to have an implementation of the `accept` method.

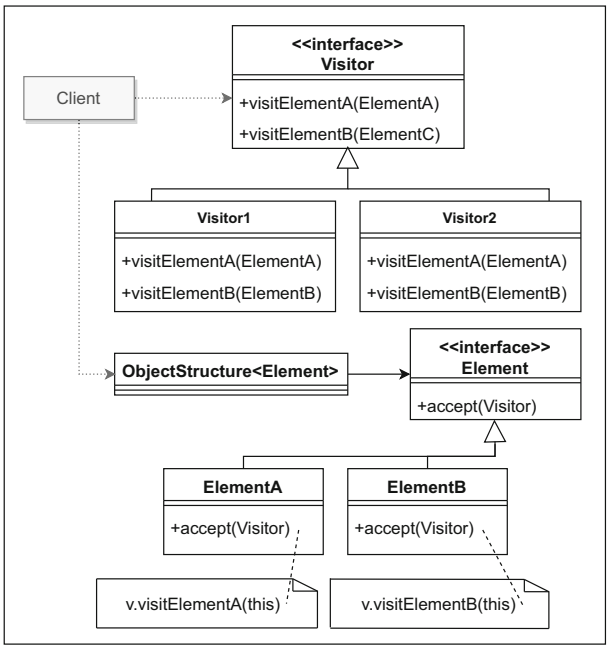

Fig. 1 UML diagram of Visitor pattern. Amended version from Gamma et al. (1995)

3.2 Features of the Visitor Pattern

The operation of the Visitor pattern relies on *double dispatch*, which means the selection of the method to invoke is based on the run-time types of *two* objects, the so-called receiving object, and an argument. Java only supports single dispatch, i.e. the decision of which method to invoke is based on the type of the receiving object only. For example, consider a class `Dog` containing methods `barkAt(Cat)` and `barkAt(Sheep)`, and assume that `Cat` and `Sheep` both implement the interface `Animal`. A method invocation in the form of

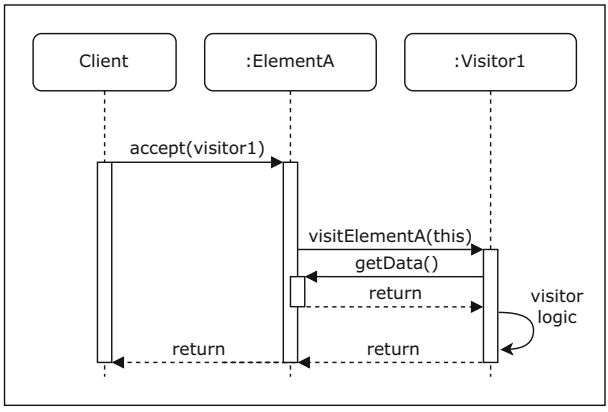

Fig. 2 Sequence diagram of a typical visitor interaction

`dog.barkAt(animal)` will not be able to select the appropriate `barkAt` implementation to execute because the `animal` argument is not used in a single-dispatch scenario.

In the case of the Visitor pattern, the method to be executed depends on the concrete type of the visitor and the concrete type of the element being visited, thus requiring double dispatch. In Java this is achieved through the use of two single dispatches: one method call of `accept` and another of `visit`. The double dispatch the Visitor requires may cause additional energy consumption due to the extra method invocations required at run-time.

The management of state is another aspect of the Visitor. The partial results computed by the `visit` methods must be aggregated in some way. One approach is to maintain explicit state in the visitor that can be accessed by the `visit` methods. Alternatively, a stateless solution is achievable by passing additional arguments through the `accept` and `visit` methods.

Lastly, the traversal of the structure being visited must be considered. The traversal can be handled in three main ways: (i) by the visitor itself, i.e. after visiting an element it computes the next element to visit; (ii) by the object structure, which would require the visitor to be passed to the structure initially; or (iii) by using a separate iterator object, which could either be an internal or an external iterator.

4 Experimental Design

The goal of this study is to examine the impact on energy efficiency of transforming the typical implementation of the Visitor pattern. We consider three treatments: the patterned application, the unpatterned application, and an alternately patterned application. Our study includes three subjects: a textbook style application developed in Java, and two open source applications: JavaParser and CppParser, written in Java and C++ respectively. The dependent variables are run time and power consumption which are used to compute overall energy consumption.

Our study addresses the following research questions:

- RQ1 In the textbook example, which implementation (unpatterned, patterned, alternately patterned) is most energy efficient?
- RQ2 In JavaParser, which implementation (unpatterned, patterned, alternately patterned) is most energy efficient?
- RQ3 In CppParser, which implementation (unpatterned, patterned, alternately patterned) is most energy efficient?
- RQ4 Are the findings of the case studies examining JavaParser and CppParser consistent?

Studying two open source applications, one written in Java and one written in C++, assists in exploring the generality of our findings. Java is compiled into an intermediate language, bytecode, which is interpreted and executed by the Java Virtual Machine (JVM) on a given platform. Described as "write-once-run-everywhere" (Javasoft 1996), Java provides flexibility as a program can be compiled once, and subsequently be executed on any machine upon which a JVM is installed. This flexibility comes with a performance cost due to the need to interpret bytecode at run-time; however, modern JVMs usually include a Just-in-Time compiler which can optimise frequently executed parts of the software, or compile parts directly to machine code at run-time. On the other hand, C++ is a platform-dependent, compiled language which is compiled directly to native machine code. This adds complication

in requiring recompilation to execute the software on different platforms, but it is precisely the direct compilation to native machine code that enables the generation of a highly performant and energy efficient executable.

4.1 Software Versions

To address the research questions outlined above, we develop three versions of each application: (i) one with the Visitor pattern implemented as per the description of Gamma et al. (1995) (patterned), (ii) one without the pattern (unpatterned), and (iii) one with an alternate implementation (alternately patterned) which we describe further below. High level UML diagrams for every version of each application are provided in the appendices.

4.1.1 Patterned Version

The patterned version of the software is implemented to have a design similar to that of Fig. 1, reflecting the original Gamma et al. (1995) implementation of the pattern. A visitor class contains the methods relating to the elements. The client invokes `accept` on each element and passes the visitor to it, and the element subsequently invokes the appropriate `visit` method in the visitor.

4.1.2 Unpatterned Version

The unpatterned version of the software sees the visitor being entirely removed. Each method in the visitor is renamed (all to the same name), the arguments of those methods are removed, references to the argument are updated to reference `this`, and each method is moved to its appropriate element class. A method with the same signature is added to the interface/abstract class that the elements implement/extend, and the code in the client is then updated to invoke that method rather than the `accept` method.

4.1.3 Alternate Patterned Version

The alternate patterned version of the software employs a different implementation of the Visitor pattern. While double dispatch is usually executed through a double method invocation (`accept` and `visit`), the alternate implementation excludes the invocation of the `accept` method. Instead, in an approach inspired by that of Büttner et al. (2004), as the client iterates through the elements, the element type is checked and passed to the appropriate `visit` method in the visitor. In Java, the `instanceof` operator is used to type check; in C++, a `dynamic_cast` is employed and the resulting pointer is checked for nullability. We describe this as *reflective dispatch*. The method to be invoked depends on the type of the visitor object (determined by dynamic binding), and the type of the node being visited (determined using `instanceof` or `dynamic_cast`).

4.2 Experimental Method

Each application is executed, and its run time (Seconds, s) and power consumption (Watts, W) are recorded. Each application is executed many times in a single experiment to extend

run time in order to accurately measure power consumption, and each experiment is executed 200 times to reduce the effects of random noise.

Run time is multiplied by the mean power consumption to calculate energy consumption ($Energy_J = Power_W \times Time_s$). The percentage change in energy consumption was calculated as such: $\frac{V2-V1}{V1} \times 100$, where $V1$ is the mean energy consumption before the transformation, and $V2$ is the mean energy consumption after transformation. Statistical significance testing is calculated using Wilcoxon rank-sum tests with a Bonferroni adjusted α where appropriate. Vargha and Delaney's $\hat{A}_{12}$ statistics are employed in both case studies to highlight effect size.

Our experimental study comprises three parts: (i) we examine a textbook example of the pattern in Java, creating three software implementations all exhibiting the same behaviour, (ii) we explore the pattern further in Java using an open source software application, JavaParser,[2] and (iii) we explore the impact of the pattern in another open source C++ application, CppParser.[3]

4.3 Experimental Setup

The software was executed on a device with an Intel Xeon E-2224G CPU at 3.5GHz and 8 GB of DDR4 2666MT/s RAM, running a fresh, minimal installation of Ubuntu 20.04.2 LTS. The Java based software was compiled and executed with OpenJDK Java 11.0.14. The C++ applications were written in C++17 and compiled with g++ 9.4.0. A Wattsup Pro Power Meter[4] was used to record power consumption. The device executing the software under test was plugged into the Wattsup Pro Power Meter, and a secondary device connected to the power meter logged the power consumption every second.

4.4 Testing Scenarios

A textbook-style example of the Visitor pattern provides the basis for a proof-of-concept experiment. To assist in clarifying the pattern's impact on energy consumption, we use a variety of configurations of the Just-In-Time (JIT) compiler.

We subsequently examine the pattern in the context of two larger open source applications. Bespoke testing scenarios are employed to test the applications and are detailed in each case study section. The testing scenario for the applications mirror each other to the greatest extent possible in terms of the ratio of elements visited and the way in which the elements are handled. While the use of included unit tests was considered, test cases do not necessarily execute code paths typically executed during normal software execution, nor do they necessarily exercise the parts of the application that include the transformed design pattern. The use of unit tests could also further reduce the parity of the testing scenarios of the two case studies. The implications of these testing scenarios are discussed further in Section 9.

[2] javaparser.org

[3] github.com/satya-das/cppparser

[4] powermeterstore.com/p1206/watts_up_pro.php

5 RQ1: Textbook Example

As a preliminary study, we first examine the impact of the Visitor pattern in a short, textbook-style application derived from an online source written in Java (Visitor in Java 2022). The versions under test range from 176 to 202 source lines of code located in seven Java files. We experiment with three different versions of the same application: (i) patterned, (ii) unpatterned, and (iii) alternate patterned, and subsequently compare energy consumption across a range of Java Virtual Machine set-ups. Lastly, we investigate the impact of stateless vs. stateful visitors on energy consumption. UML diagrams of each version of the textbook application are located in Appendix A.

5.1 Textbook Implementations

5.1.1 Patterned Version

The textbook example used in this set of experiments consists of three classes that implement the interface `Shape`: `Circle`, `Dot`, and `Rectangle`. Each class stores x and y coordinates, and either radius or width and height values. The visitor `XMLExportVisitor` contains an `export` method, which iterates through the list of shapes, invoking `accept` and passing itself as an argument. Each `XMLExportVisitor` visit method takes relevant data from each shape and formats it into XML string which is then returned and ultimately appended to a string in the `export` method of the `XMLExportVisitor`.

In this example, the visitor is stateless as it has no instance variables. In addition, to avoid complications regarding the method of traversal, the shapes are simply passed to the visitor as a list, which then invokes `accept` on each shape.

5.1.2 Unpatterned Version

In removing this pattern, every `visit` method in `XMLExportVisitor` is renamed `exportXML`, before subsequently being moved to its appropriate class. The `Visitor` argument is removed, and references to it are updated to reference `this`. The `accept` invocation in the `export` method is then updated to invoke `shape.exportXML()`. Lastly, the `exportXML` method signature is added to the `Shape` interface.

This set of refactorings leaves behaviour unchanged, and the traversal of the shapes unchanged. However the visitor is removed and the double dispatch has been eliminated.

5.1.3 Alternate Patterned Version

In this version, the only change that is made is the logic handling the shapes during traversal. In implementing reflective dispatch, rather than invoking `accept` on each shape, there is a block of type checks. When the instance of a given shape is found, it is cast and passed as an argument to the appropriate `visit` method in the visitor.

With this alteration, the behaviour remains the same, the traversal of the shapes remains the same, and while the double dispatch effectively remains, the way in which it is implemented is significantly altered; instead of two methods being invoked, a single method is invoked following an `if` statement and `instanceof` operation.

5.1.4 State vs Stateless

An important implementation detail of the Visitor pattern is whether or not the visitor holds explicit state. In order to investigate the impact of state, two additional versions of the patterned experiment were conducted: one with an instance variable in the visitor holding a list of strings to which each formatted XML string for each shape is added, and another version in which the list is passed as an additional argument through each `accept` and `visit` method.

5.2 Textbook Experiments

In a single set of experiments, three different shapes are instantiated and the `export` method is invoked with the three shapes as arguments 1×10^9 times when the JVM is operating normally and inlining is disabled, and 1×10^7 times when the JIT compiler is disabled.

The time before and after the loop is recorded providing overall run time, and power consumption is recorded throughout.

Additionally, the first three versions, patterned, unpatterned, and alternate patterned, are executed with three different Java Virtual Machine (JVM) set-ups: (i) a normal JVM with no custom parameters; (ii) a JVM with the Just-in-Time (JIT) compiler disabled; and (iii) with the maximum method inline size set to 1 (`-XX:MaxInlineSize=1`) effectively disabling method inlining as an optimisation technique. The experiments involving state are executed twice: (i) with a normal JVM, and (ii) with the JIT compiler disabled.

Table 1 Mean run time (s), power consumption (W), and energy consumption (J) for the patterned (double dispatch), unpatterned (single dispatch), and alternate patterned (reflective dispatch) versions of the textbook example across differing settings of the JVM. Reduction in energy consumed is statistically significant in cases marked with * having conducted a Wilcoxon rank-sum test with a Bonferroni adjusted $\alpha = 0.025$ (0.05/2))

Version	Run Time (s)	Power Cons. (W)	Energy Cons. (J)	Change (%)	P-value
Normal JVM					
Patterned	211.71	55.51	11751.4		
Unpatterned	210.79	55.69	11739.09	-0.1	0.44
Patterned	211.71	55.51	11751.4		
Alternately Patterned	195.73	55.37	10838.54	-7.77*	2.43×10^{-67}
No JIT JVM					
Patterned	244.14	43.32	10576.76		
Unpatterned	242.37	43.24	10480.65	-0.91*	6.25×10^{-6}
Patterned	244.14	43.32	10576.76		
Alternately Patterned	228.51	43.03	9832.2	-7.04*	2.25×10^{-34}
No Inlining JVM					
Patterned	215.91	51.76	11175.02		
Unpatterned	211.67	51.86	10976.59	-1.78*	9.49×10^{-18}
Patterned	215.91	51.76	11175.02		
Alternately Patterned	198.5	51.38	10199.21	-8.73*	1.21×10^{-49}

Table 2 Mean run time (s), power consumption (W), and energy consumption (J) for the stateful and stateless versions of the textbook example when the JVM operates normally, and when JIT is disabled. Reduction in energy consumed is statistically significant in cases marked with * having conducted a Wilcoxon rank-sum test ($\alpha = 0.05$)

Version	Run Time (s)	Power Cons. (W)	Energy Cons. (J)	Change (%)	P-value
Normal JVM					
Stateful	137.18	54.11	7422.28		
Stateless	137.15	53.84	7384.56	-0.51*	1.66×10^{-4}
No Normal JVM					
Stateful	205.51	42.31	8695.34		
Stateless	207.01	42.59	8815.97	1.37*	7.69×10^{-4}

5.3 Textbook Results

The mean run times, power consumption, and overall energy consumption of each set of experiments are listed in Tables 1 and 2. Power consumption remained constant throughout each of the first three sets of experiments, however a notable reduction in power consumption can be seen when the JIT compiler is disabled. Given the otherwise stable power consumption, the major factor influencing energy consumption is run time.

6 RQ2: Case Study I— JavaParser

Subsequent to our examination of a textbook example of the Visitor pattern, we consider a larger, open source example. A common use of the Visitor pattern is in the traversal of tree-like structures, and it is often employed in code analysis tools when handling abstract syntax trees (ASTs).

JavaParser is an open source library that employs ASTs to enable the analysis, transformation, and generation of Java source code. Our experiments were conducted with release 3.23.0,[5] consisting of 87,671 source lines of code (SLOC) across 717 files. The application that is parsed is JHotDraw V7.0.6.[6] JavaParser's parsing of JHotDraw, a project consisting of 309 Java files containing 32,122 SLOC, yields 187,288 AST nodes. This provides a realistic use case of the Visitor pattern.

In this section we use the JavaParser term *Node* for the objects being visited, rather than the generic term *Element*.

6.1 JavaParser Implementations

High level UML diagrams describing transformed parts of each version of JavaParser are located in Appendix B.

[5] github.com/javaparser/javaparser/releases/tag/javaparser-parent-3.23.0

[6] sourceforge.net/projects/jhotdraw

6.1.1 Patterned Version

The patterned visitor experimented with is called `CustomVisitor`, and implements JavaParser's `VoidVisitor` interface. Each `visit` method is included in this class, the implementation of which varies depending on the set of experiments being conducted, as described below. Once JHotDraw is parsed, a `BreadthFirstIterator` (an iterator implemented in JavaParser) is used to traverse the AST; this avoids confounding variables regarding traversal across implementations.

6.1.2 Unpatterned Version

To remove the Visitor pattern, a new method called `addToList` is added to the `Node` class, which each concrete node extends. Where necessary, this method is also overridden in relevant nodes' concrete classes. Where the `accept` method was invoked in the patterned version, the `addToList` method is now invoked instead, thus removing the need for the `CustomVisitor` class and the double dispatch, eliminating the pattern itself.

6.1.3 Alternate Patterned Version

The alternate implementation of this pattern leaves the `CustomVisitor` as is. However, with reflective dispatch, as the AST is traversed the type of each node is checked with `instanceof`, and if it is of the type the visitor is seeking, it is cast and passed as an argument to the appropriate `visit` method in the `CustomVisitor` class.

6.2 JavaParser Experiments

In exploring the application of the Visitor pattern further, we consider three usage scenarios: (i) the visitor not interacting with any nodes, (ii) the visitor interacting with some nodes, and (iii) the visitor interacting with every node in the AST. While the visitor de facto interacts with every node given it visits each one, we use the term "interact" to reference further logic in the visitor that processes the node.

6.2.1 Never Interacts

In the first scenario, while the visitor visits every node in the AST, the node type it wants to handle (`WildCardType`) is never found (i.e. there are no nodes of type `WildCardType` in JHotDraw), thus it does not interact with any. We describe this as "never interacting," as the visitor never finds a node it has `visit` logic implemented for. The tree is traversed by the visitor 10,000 times in each experiment.

The patterned version in this experiment invokes `accept` on each node while passing the visitor to it. Only the `visit` method for `WildCardType` is implemented, adding its `toString` return value to a list in the visitor.

In the unpatterned version of this experiment, the method `addToList` is added to the `Node` class. In this scenario, only the class `WildCardType` overrides the method, adding its `toString` return value to a list provided as an argument.

In the alternate patterned version, the AST is iterated through and if the instance of a given node is of type `WildCardType`, the node is cast appropriately and passed as an argument to the appropriate `visit` method in the visitor.

6.2.2 Sometimes Interacts

In the second scenario, the visitor collects the name of every `ClassOrInterfaceDeclaration` node. There are 323 of these nodes making up less than 1% of the total number of nodes in the AST, providing another use case of a visitor. We describe this scenario as "sometimes interacting." The tree is traversed by the visitor 10,000 times in each experiment.

In the patterned version of this experiment, the `visit` method for `ClassOrInterfaceDeclaration` in `CustomVisitor` adds the declaration's name to a list.

In the unpatterned version of this experiment, the method `addToList` is added to the `Node` class, and a method of the same name is overridden in the class `ClassOrInterfaceDeclaration`, which adds the declaration's name to a list passed as an argument.

In the alternate patterned version of this experiment, a single `instanceof` operation checks for the type `ClassOrInterfaceDeclaration` as the nodes are iterated through. If found, the node is cast appropriately and passed as an argument to the appropriate `visit` method in the visitor.

6.2.3 Always Interacts

In the final scenario, the visitor interacts with every node of the AST, storing the string returned by the nodes' `toString` method. This scenario is described as "always interacting," and can provide insights into the overhead introduced when every node is handled by the visitor logic. The tree is traversed by the visitor 300 times in each experiment.

In the patterned version of this experiment, every `visit` method is implemented in the `CustomVisitor` class. Each method adds the return value of that node's `toString` method to a list in the visitor.

In the unpatterned version of this experiment, every `visit` method in the `CustomVisitor` is renamed to `addToList`. The arguments referencing each concrete type of `Node` is replaced with an argument of type `List<String>`, to which the returned value from `toString` is added. These methods are subsequently moved to every concrete class extending `Node`.

In the alternate patterned version of this experiment, the `instanceof` operator is used to type check every node. When a node of the desired type is reached, it is cast and passed as an argument to the appropriate `visit` method in the visitor.

6.3 JavaParser Results

The mean run times, power consumption, and overall energy consumption of this set of experiments are listed in Table 3. The mean power consumption remains stable throughout each set of experiments, though there is a notable increase in the case where the visitor interacts with every node, and a very slight increase in power consumption is also seen when some nodes are interacted with when compared to no nodes being interacted with. The energy consumption of each version of the program, relative to the patterned version is presented in Fig. 3.

Table 3 Mean run time (s), power consumption (W), and energy consumption (J) for the patterned (double dispatch), unpatterned (single dispatch), and alternate patterned (reflective dispatch) versions of the open source JavaParser example in cases where the visitor interacts with no nodes, interacts with some nodes, and interacts with all nodes. Reduction in energy consumed is statistically significant in cases marked with * having conducted a Wilcoxon rank-sum test and a Bonferroni adjusted $\alpha = 0.025$ (0.05/2)). Vargha and Delaney's $\hat{A}_{12}$ statistic was computed comparing the original, patterned version to each transformed version of the application

Version	Run Time (s)	Power Cons. (W)	Energy Cons. (J)	Change (%)	P-value	$\hat{A}_{12}$
Never Interacts						
Patterned	114.81	43.15	4954.13			
Unpatterned	105.54	42.69	4506.05	-9.04*	3.13×10^{-36}	0.862
Patterned	114.81	43.15	4954.13			
Alternately Patterned	102.86	42.28	4349.00	-12.21*	1.75×10^{-36}	0.863
Sometimes Interacts						
Patterned	117.26	44.17	5179.18			
Unpatterned	107.61	43.88	4722.32	-8.82*	5.11×10^{-38}	0.871
Patterned	117.26	44.17	5179.18			
Alternately Patterned	106.54	43.37	4620.69	-10.78*	1.07×10^{-38}	0.875
Always Interacts						
Patterned	232.99	55.47	12923.48			
Unpatterned	232.14	55.31	12839.92	-0.65*	3.68×10^{-5}	0.615
Patterned	232.99	55.47	12923.48			
Alternately Patterned	232.40	55.38	12870.20	-0.41*	0.003	0.580

7 RQ3: Case Study II — CppParser

Given the impact of the pattern in JavaParser, we expand the scope of this study to include investigation into the pattern with an additional programming language. As described in Section 4, C++ is a popular language that enables object-oriented development but compiles to machine code rather than using an interpreter like Java. If the theory regarding the impact of design patterns on energy consumption is correct, given the indirection and additional structures they introduce, we expect to see results in line with those found in Section 6 regardless of an application's implementation language. Thus, C++ provides an excellent language to experiment with given its contrasts when compared to Java. CppParser is an open source library that creates ASTs while parsing code written in C++. Our experiments were conducted with the version committed on October 3 2022,[7] consisting of over 4907 source lines of code (SLOC) across 32 source files. The contents of *Constants.cpp*, a part of the source code for LLVM, is parsed which contains 2648 SLOC yielding 1326 elements. Similarly to JavaParser, CppParser provides a realistic use case of the Visitor pattern in practice. While CppParser does not natively include an instance of the Visitor pattern, its use of an AST provides an excellent use-case, so the pattern was introduced in line with

[7] github.com/satya-das/cppparser/commit/f9a4cfac1a3af7286332056d7c661d86b6c35eb3

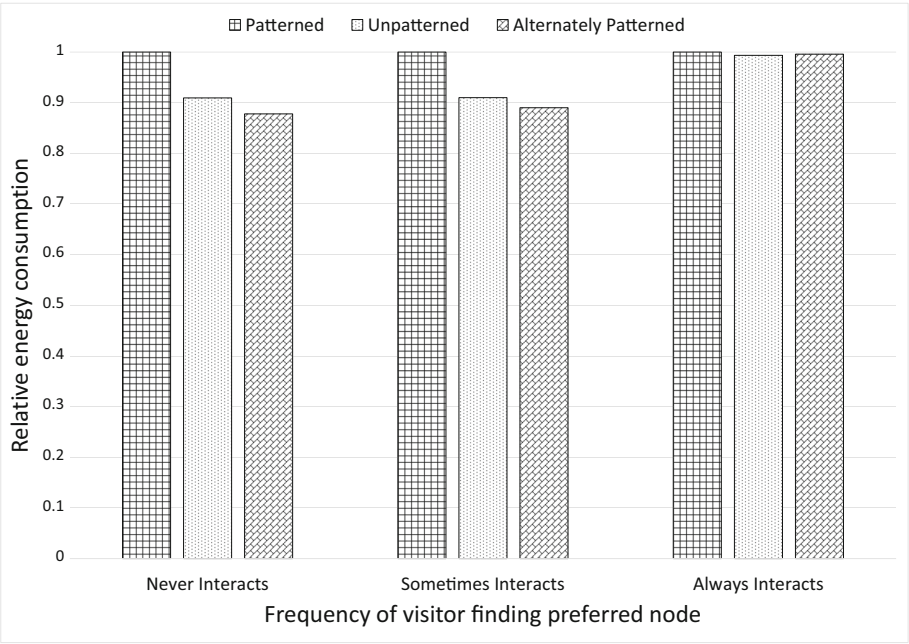

Fig. 3 Energy consumption of each version of JavaParser as a proportion of that set's patterned version in JavaParser

the pattern implementation described by Gamma et al. (1995). Every version of CppParser created for these experiments mirror the experiments conducted with JavaParser in terms of pattern implementation, or lack thereof, and in terms of the specific experiment scenarios described in Section 7.2.

7.1 CppParser Implementations

High-level UML diagrams of the transformed parts of each version of CppParser are located in Appendix C.

7.1.1 Patterned Version

In order to implement the pattern, an abstract class `Visitable` was included in `cppast.h` that contains an `accept` method. `CppObj`, which is the commonly used base class for all AST elements, extends `Visitable` and an `accept` method is added to every concrete AST element that invokes their respective `visit` methods in `Visitor`. An additional abstract class `Visitor` is implemented containing pure virtual visit functions for each AST type, and a `CustomVisitor` class is included which implements each of those abstract methods; `CustomVisitor` is not required for a simple implementation of the pattern, but provides utility in being a base from which future visitors can be easily extended, overriding only the `visit` methods relevant to that visitor. This reflects the approach to the Visitor pattern that is implemented in JavaParser. Once the code is parsed, the AST is flattened, providing a vector of elements which is iterated through, and `accept` is invoked on each element.

7.1.2 Unpatterned Version

In removing the pattern, a new method `addToList` is added to `CppObj`. Where required, this method is overridden in the relevant element's concrete class. While `accept` was invoked in the patterned version as the elements are iterated through, `addToList` is now invoked instead, resulting in an unused visitor, no double dispatch, and an effective removal of the pattern.

7.1.3 Alternate Patterned Version

The alternate implementation of the pattern leaves the visitor as is, however instead of invoking `accept`, we type check for relevant element types. This is completed using `dynamic_cast`; if the returned pointer is not null, i.e. the element is of a type we are interested in visiting, that element's `visit` method in the visitor is invoked, and the cast pointer is passed as an argument.

7.2 CppParser Experiments

Reflecting our previous experiments with JavaParser, we investigate the same three usage scenarios: (i) the visitor not interacting with any elements, (ii) the visitor interacting with some elements, and (iii) the visitor interacting with every element in the AST. To reiterate, while the visitor de facto visits every element, we use the term "interact" to reference the execution of further logic in the visitor processing the element.

7.2.1 Never Interacts

In this scenario, the element type the visitor is interested in is `CppPragma`, of which there are none in the AST. While every element is visited, none are interacted with further by the visitor. The tree is traversed 3×10^6 times in each experiment.

The patterned version in this experiment sees the visitor implementing a visit method only for `CppPragma`; if a `CppPragma` is found, it is added to a vector.

The unpatterned solution invokes `addToList` on every element passing a pointer to a vector as an argument. `addToList` is only implemented in `CppObj` (an empty method) and also in `CppPragma`, in which `this` is added to the vector.

The alternate patterned version invokes `dynamic_cast` on every element, casting to `CppPragma`, and if the resulting pointer is not null, passes it as an argument to `visitCppPragma` in the visitor where it is added to a vector.

7.2.2 Sometimes Interacts

In the second scenario, the visitor collects every `CppForBlock` element. There are 15 elements of this type making up just over 1% of total number of elements in the AST, mirroring the 1% of nodes interacted with in the JavaParser case study. The AST is traversed by the visitor 3×10^6 times in each experiment.

In the patterned version of this experiment, only `visitCppForBlock` is implemented, adding the `CppForBlock` to a vector when found.

The unpatterned version invokes `addToList` on every element, passing a pointer to a vector as an argument; however, it is only implemented in `CppObj` and in `CppForBlock`, in which `this` is added to the vector.

The alternate patterned version invokes `dynamic_cast` on every element, casting to a `CppForBlock`, and if the resulting pointer is not null, it is passed as an argument to `visitCppForBlock` in the visitor where it is added to a vector.

7.2.3 Always Interacts

In the final scenario, the visitor collects every element of the AST in a vector. The AST is traversed by the visitor 5×10^6 times in the patterned and unpatterned versions, and 5×10^5 times in the alternate patterned version. Following early exploration of the alternate implementation of the pattern in CppParser, substantially longer run times were noted when compared to other versions, thus the number of times the always interacting, alternate patterned set of experiments was executed was reduced by a factor of 10 to reduce unnecessary energy usage. We clarify this further in Section 7.3. In the patterned version of this experiment, every `visit` method is implemented, adding every element to a vector.

In the unpatterned version, `addToList` is implemented in every concrete AST type, in every case adding `this` to a vector passed as an argument.

Lastly, in the alternate patterned version, every element is cast to every concrete AST type with `dynamic_cast`, and when resulting pointer is not `null`, its relevant `visit` method in the visitor is invoked, passing the pointer as an argument where it is added to a vector.

7.3 CppParser Results

The mean run times, power consumption, and overall energy consumption of this set of experiments are listed in Table 4. The mean wattage remains stable throughout experiments except for the alternate patterned solution which sees an approximate five Watt increase in power consumption in all scenarios. The energy consumption of each version of the program, relative to the alternate patterned version, is presented in Fig. 4. In executing the alternate implementation, the number of loops executed was reduced by a factor of 10 to eliminate energy usage as preliminary executions highlighted substantially greater run times when compared to the other versions. Multiplying the resulting energy consumption data by 10 to reach an approximation of real energy consumption, we see an increase in energy consumption of 2012.64%.

8 Discussion

The data described in Sections 5.3, 6.3, and 7.3 highlight a negative relationship between the typical implementation of the visitor pattern and energy efficiency. However, they also emphasise the consideration that must be paid to how we design software, and also the potential for short, textbook style software examples to lack adequate context to provide relevant results in this style of empirical experimentation. We explore this further in the remainder of this section.

Table 4 Mean run time (s), power consumption (W), and energy consumption (J) for the patterned (double dispatch), unpatterned (single dispatch), and alternate patterned (reflective dispatch) versions of open source CppParser example in cases where the visitor interacts with no elements, interacts with some elements, and interacts with all elements. Reduction in energy consumed is statistically significant in cases marked with * having conducted a Wilcoxon rank-sum test with a Bonferroni adjusted $\alpha = 0.025(0.05/2)$. Vargha and Delaney's $\hat{A}_{12}$ statistic was computed comparing the original, patterned version to each transformed version of the application. † Number of loops executed in the always interacting alternate patterned scenario is reduced by a factor of 10 to reduce unnecessary energy consumption (described further in Section 7.3). Run time and energy consumption for this scenario is multiplied by 10 to provide an estimate in overall energy consumption against which we do not perform statistical analysis

Version	Run Time (s)	Power Cons. (W)	Energy Cons. (J)	Change (%)	P-value	$\hat{A}_{12}$
Never Interacts						
Patterned	86.26	44.71	3856.74			
Unpatterned	28.84	44.05	1270.58	-67.06*	2.4×10^{-67}	1.000
Patterned	86.26	44.71	3856.74			
Alternately Patterned	179.81	50.01	8992.73	+133.17	1	0.000
Sometimes Interacts						
Patterned	87.30	44.64	3896.93			
Unpatterned	28.78	43.98	1265.80	-67.52*	2.4×10^{-67}	1.000
Patterned	87.30	44.64	3896.93			
Alternately Patterned	181.12	49.95	9046.30	+132.14	1	0.000
Always Interacts						
Patterned	307.58	44.83	13788.65			
Unpatterned	221.94	45.05	9999.57	-27.48*	2.4×10^{-67}	1.000
Patterned	307.58	44.83	13788.65			
Alternately Patterned	5683.00	51.26	291304.58	+2012.64	n/a†	0.000

8.1 RQ1: Textbook Example Discussion

The data from the experiments involving the textbook example suggests that little can be gained from the removal of the Visitor pattern from software through traditional means, i.e. moving relevant logic from the visitor to the appropriate visitable class, and eliminating the double dispatch. This is not unexpected behaviour given the optimisation the JIT compiler can provide (Paleczny et al. 2001). Method inlining is one aspect of code optimisation used and given double dispatch requires two method invocations, the JIT compiler can be expected to heavily optimise the visitor pattern implementation in a trivial piece of software such as the textbook example. While disabling the JIT compiler presents a less realistic usage scenario, it can provide some insight into the potential impact of transforming software design. It can provide a best-case scenario in an environment where JIT optimisations are impossible e.g. due to long methods or methods not reaching the threshold to be considered "hot," etc. In this case we see a reduction in energy consumption when the pattern is removed of almost 1%. To explore this further, we disable inlining by setting the maximum method inline size to one

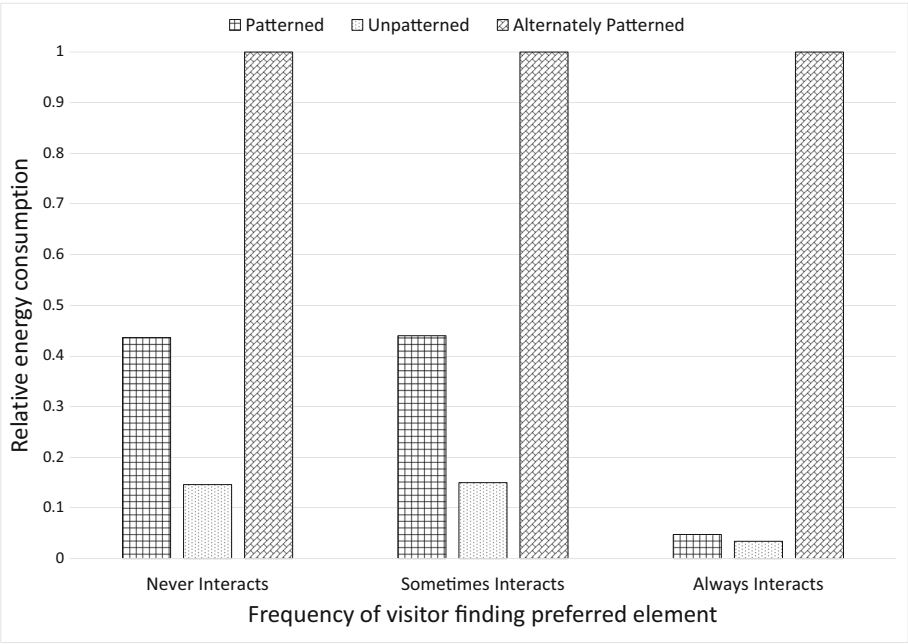

Fig. 4 Energy consumption of each version of the CppParser as a proportion of that set's alternate patterned, reflective dispatch version, in JavaParser

byte. In this scenario, we see a reduction in energy consumption of almost 2%, suggesting that method inlining is a key optimisation technique when the Visitor is operating in a normal JVM.

The results when comparing the traditional pattern implementation to the one using reflective dispatch provides an interesting and unexpected result. Counterintuitively, checking the type of an object and passing a cast version of it to a visitor is more energy efficient than simply invoking a dynamically-bound method on that object. It may be the case that invoking the `instanceof` operator is a more efficient JVM operation, or that the JIT compiler is more effective in optimising the type check and casting code. A difference in energy consumption when comparing the patterned vs unpatterned, and patterned vs alternate patterned version of approximately 6% remains even when the JIT compiler is disabled, providing further evidence that explicitly type checking, casting an object, and invoking a method within the same object, has a distinctly lesser energy cost than what we may intuitively believe is cheaper: invoking one dynamically-bound method on another object. There may also be costs due to polymorphism being employed—the method in the single dispatch version is invoked on an object of type `Shape`, rather than a concrete type which may introduce additional overhead.

The 0.51% difference between a stateful and stateless visitor, while a small change, is statistically significant and could have an impact in situations where the application is constantly running and makes heavy use of an instance of the pattern. The change here is largely a difference in power consumption and suggests certain internal JVM operations may be more costly than others. The shift in result when the JIT compiler is disabled presents more com-

plex results. The difference is largely seen in the run times, suggesting the stateless version can be better optimised at run-time by the JIT compiler, when it is operating as normal.

8.2 RQ2: JavaParser Discussion

The experiments with JavaParser (see Table 3) present improved savings over those found with the textbook example, even when the JIT compiler is disabled in those experiments, with percentage reductions in energy consumption of approximately 7% and 10% when comparing the patterned version to the unpatterned, and alternate patterned version respectively. It is noteworthy (and heartening for this research) that the real-world example yielded more promising results than the textbook example.

In exploring the pattern in the case study we opted not to experiment with different JVM settings. However, to further examine the counterintuitive results regarding the alternate patterned version we examined it in three scenarios covering interaction with no nodes, interactions with some nodes, and interaction with all nodes. We undertake this effort to potentially highlight the impact of the two differing code paths: (i) the path taken when the visitor visits a node it does not need to interact with, and (ii) the path taken when the visitor does interact with a given node.

When there are no interactions, the data again suggests that the alternate pattern implementation shows a better improvement in energy consumption than the unpatterned solution. When a node is visited in the patterned version, but is not interacted with further, there are two method invocations involved (`accept` and `visit`), both methods being invoked on another object. In the case of the unpatterned version, a single method is invoked (`addToList`), and is invoked on another object. In the alternate patterned case, a single `if` statement and an `instanceof` operator are executed. The results seen here present an expected reduction in energy consumption between the patterned and unpatterned versions of the application, but also provides an interesting insight into the efficiency of a binary operator given the impact on energy efficiency of the alternate patterned implementation.

While we still see significant reductions in energy consumption when the visitor interacts with every node, these reductions are substantially less than in other experiments. In the patterned version, every visit to a node involves four method invocations (`accept`, `visit`, `values.add(n.toString)`), all of which are invoked on other objects. In the unpatterned version there are three method invocations (`addToList`, `values.add(this.toString)`), two of which are invoked on other objects. In the alternate patterned version three method invocations (`visit`, `values.add(n.toString)`) are involved, all of which are invoked on other objects, one `if` statement, one `instanceof` operator, and one cast. There is one fewer method invocation in the unpatterned and alternate patterned versions when compared to the patterned version, however, the alternate patterned version also includes an additional `if` statement, an `instanceof` operator, and a cast presenting a logical explanation of the shift in results with the unpatterned version now yielding greatest efficiency.

The "sometimes interacts" case yields results in line with the above explanation: the performance gap between the single dispatch and alternate patterned solution decreases slightly which is as expected given a non-zero number of interactions; while the change in percentage difference is very small, it is to be expected as the number of interactions is small (fewer than 1% of nodes were interacted with).

To summarise these results, while the alternate pattern sees substantial improvements in terms of energy consumption in most cases, we note that the optimal solution in scenarios

where every element in the tree is interacted with is complete pattern removal. This finding highlights the importance of context (in this case the nature of the tree being processed) in optimising the energy efficiency of software.

8.3 Comparing Textbook and JavaParser Results

While caution must be taken when experimenting with textbook examples as done in Section 5, the results gathered in experiments with the more realistic open source example demonstrate the very notable impact the Visitor pattern can have on energy consumption, with some unexpected results.

The findings when comparing the patterned and alternate patterned versions of the textbook example are congruent with those of the larger case study; however, the reduction in energy consumption when JavaParser was transformed was greater than the reduction seen in the textbook example.

Additionally, no significant change in energy consumption is seen when comparing the patterned and unpatterned versions of the textbook example in the normal JVM scenario, but we see a reduction in energy consumption of over 8% in the JavaParser case study. This substantial difference in energy reduction was noted even when we consider a "best-case scenario" of the textbook example, in which the JIT compiler is disabled, with a reduction of less than 1%. We can suggest two possible reasons for the differences observed: (i) the textbook code example is small in size and less complex than the open source example allowing for greater JIT optimisations; (ii) the code used for the textbook example may simply include too much overhead in terms of string concatenation and formatting (given the example is intended to be an XML style exporter), potentially limiting the impact of the design change following the removal of the pattern. Given the pronounced energy performance differences observed between the case study examining JavaParser and the simpler textbook example, the importance of exploring larger, more realistic examples of pattern instances in future work, as we have done with our case study, is clear.

8.4 RQ3: CppParser Discussion

To mirror the approach taken in examining the impact of the Visitor pattern in JavaParser, the same three scenarios were tested with CppParser: one in which no elements are interacted with, one in which some elements are interacted with, and one where the visitor interacts with every node in the AST. Overall, the resulting data highlights substantial energy savings when the pattern is completely removed from CppParser (see Table 4).

In the first two scenarios, complete removal of the pattern saw energy consumption reduce by a significant 66%. There was a smaller, but still notable, energy consumption reduction of just over 28% in the scenario in which the visitor interacts with every element. The numbers of method invocations involved here mirror those of the JavaParser experiment, an `accept` (`->`), `visit` (`->`), and `push_back` (`.`) invocation in every patterned version compared to the `addToList` (`->`) and `push_back` (`->`) invocations of the unpatterned solution. Further detailed exploration of the C++ implementation may highlight more acute causes of energy consumption shifts such as the energy cost of dereferencing with class member access operators (i.e. `->`), however that lies beyond the scope of this higher level study. The patterned implementation here sees two methods invoked that dereference pointers, one that does not, while the unpatterned solution has two method invocations dereferencing pointers.

The alternate pattern's implementation in C++ presents substantial increases in energy consumption in all cases, however it is most notable in the scenario where every element is interacted with. Type checking in C++ requires an initial cast and subsequent pointer check which greatly hampers energy performance. For each element type the visitor is interested in visiting, the alternate implementation sees a `dynamic_cast`, the result being assigned to a variable, and a subsequent `if` statement that checks if it is null before invoking the relevant `visit` method in the visitor. In the never interacting and sometimes interacting scenarios, this cast and pointer check is only executed once per element (in the search for `CppPragma` and `CppForBlock` elements respectively). In the scenario where every element is interacted with, the cast and pointer check is executed 37 times for each element (there are 37 concrete AST types in CppParser). The impact here could be reduced with nested `if` statements, however the solution employed in our experiments provides a "worst-case scenario" (in a situation where the elements being visited are handled in the innermost `if` statement) and is implemented with the same approach taken in the JavaParser experiments.

The mean power consumption of the patterned and unpatterned solutions remain similar throughout experiments with CppParser, however, the alternate implementations see a five Watt increase (approximately 11%) in all scenarios. This is likely due in part to a greater number of operations being executed; with the greater number of operations, and many of them being located around if/else statements, the greater power draw may suggest that pipeline optimisations and branch predictions are maximising CPU usage. This should improve run time (and intuitively as a side-effect, energy consumption) from a performance perspective, however, it results in this slight increase in power consumption throughout run-time.

8.5 RQ4: Comparing JavaParser and CppParser Results

The experiments investigating the Visitor pattern's impact in JavaParser provided interesting results, not only highlighting the importance of open source testing, but also highlighting the pattern's significant energy consumption. While JavaParser provides a good example of the pattern in use, testing only one application in a single language raises a threat to the validity of our findings (discussed further in Section 9). As discussed above, CppParser provides a similar application of the pattern in a different language which can provide insight into the generalisability of the JavaParser findings.

In terms of complete removal of the pattern, the results from CppParser are even more promising than those of JavaParser. The pattern appears to have a greater impact on energy performance in C++ than in Java with energy consumption reductions in CppParser of 28% to 66% compared to JavaParser's 0.65% to 9%. This can be explained in part by Java's interpreted nature; the JIT compiler optimising frequently executed code can provide substantial performance gains while applications written in C++, once compiled, are not optimised further at run-time. With this consideration, research in this area, which remains in its infancy, may be better applied to non-interpreted languages given the problems that can arise attempting to benchmark software that is being optimised during run-time.

The impact of the alternate implementation of the pattern in C++ contrasts greatly with that of Java, with CppParser's energy consumption increasing by 132% to 2012% compared to a reduction in energy consumption of between 0.41% and 12% when applied to JavaParser. The aforementioned implementation of reflective dispatch in C++ introducing a substantial amount of overhead provides a clear explanation for the large increases in energy consumption. The contrasting results however, highlight the importance of consid-

ering language-centric feature implementations and their quirks, such as how to type check in a given language. While Java's `instanceof` is similar to Ruby's `is_a?`, Python's `isInstance`, and C#'s `is`, it varies greatly from how a developer can type check in C++, and in any case, the implementation of the type checking system in any language, or even different implementations of the same language standard could vary. While worthy of investigation from the perspective of highlighting language specific energy saving approaches, the alternate pattern arguably strays from the underlying thesis regarding the impact of object-oriented programming on energy consumption. However, continuing with an object-oriented perspective on software design, the type checking employed in reflective dispatch can be considered a code smell, a hint that there is something wrong with the design of software. In this case, we have highlighted that type checking, as a code smell, is unlikely to have generalisable impacts on energy consumption when all languages are considered given it improves energy performance in Java, but hampers it in C++.

In comparing the impact on energy consumption of these transformations in applications written in Java and C++, we note that the results differ greatly when using the alternate implementation of the pattern. This difference in results suggests that findings for applications written in one language cannot be assumed to generalise across other languages.

8.6 Comparison to Previous Work

Only two previous studies have examined the Visitor pattern and its relation to energy performance. Sahin et al. (2012) and Noureddine and Rajan (2015) studied the pattern in C++ and found its implementation reduced energy consumption by almost 8% and increased consumption by approximately 3 to 4% respectively.

While our findings are congruent with those of Noureddine and Rajan, they are contrary to the findings of Sahin et al. The reason for the difference between our findings and those of Sahin et al. is difficult to pinpoint as it is not clear from their work what example they used in their Visitor pattern experiments. Nevertheless, several possible reasons can be put forward. It may simply be the case that textbook code examples do not provide a complete insight in what we might assume is a best-case scenario, or may be due to some other confounding effect. An example of a potential confounding effect is the method of traversal employed; while we maintained simple lists of elements which are iterated through in experiments, it is possible their example used a more common approach in which the tree structure handles the visitor's traversal. This would change when the pattern is removed which is likely to have an impact on the results. The method of traversal employed in the Visitor falls into the ambit of the Iterator pattern which is out of the scope of this study and will therefore be considered in future work.

8.7 Implications

For a developer looking to improve the energy efficiency of their applications, we have highlighted considerations that must be taken into account when handling the Visitor pattern. Total pattern removal consistently improves energy efficiency, however the size of the improvement depends on how frequently the visitor interacts with the elements of the structure being visited. This consideration is even more important if developing in Java, where the alternate pattern sees greater energy consumption reductions when elements are interacted with less

frequently. For researchers, we have further highlighted the impact of the Visitor pattern on energy efficiency, and worked towards clarifying the aforementioned contradictory findings of existing literature. These findings can guide future research towards automated approaches to removing the design pattern to reduce energy consumption. The proof-of-concept experiment using a textbook style application also highlights the difficulties in trying to draw definite conclusions from small experiments, and indicates that future research should focus on experimentation with larger applications.

9 Threats to Validity

In this section we present potential threats to the validity of the experiments conducted. In line with Wohlin et al. (2012), we consider Internal Validity, Construct Validity, External Validity, and Conclusion Validity.

Internal Validity refers to the extent to which we can present a cause-and-effect relationship between our treatments and observed effects. In our experiments we apply a series of refactorings to software and subsequently record the application's run time and power consumption. A Watts Up Pro power meter is used to record the power consumption of the device executing the application under test, which has a reported accuracy of $\pm 1.5\%$. This solution may record the energy consumed by background processes or third-party programs. To reduce this threat, a minimal installation of the operating system was used.

Construct Validity is concerned with the relationship between theory and observation, both in the treatment and the outcome. In the case of our experiments, we have two treatments: (i) the complete removal of the Visitor pattern, (ii) the transformation of the pattern to an alternate implementation based on reflective dispatch. There are no exact guidelines regarding the removal of the Visitor pattern from software and other researchers may have taken a different approach. Our approach involves removing the visitor classes entirely and removing the double dispatch which is certainly one possible interpretation of removing the Visitor pattern. There are likely other alternate implementations of the Visitor pattern with potentially different energy profiles. The implementation we investigated, taking a similar approach to that of Büttner et al. (2004), was the only alternative approach we encountered in the existing literature. In the case study examining CppParser, we implemented the alternate pattern using the robust and idiomatic `dynamic_cast` operator. There are other approaches that could be taken, such as with the `typeid` operator. However, `typeid` provides a less general solution as it would not work in the common situation where the type of some of the elements in the structure being visited is a shared superclass of the elements.

External Validity concerns the generalisability of our results. We first study a simple textbook example to show that our approach has merit. We then apply it to a more realistic example, JavaParser, a large, open source application written in Java. While this is a realistic example, general claims cannot be made from a single sample. To improve the external validity further, we apply a similar approach to an open source application written in C++. This provides an additional example of the pattern's impact in software, and provides the bonus of testing the approach in a different language. Applying our approach to a large number of applications in many programming languages would be ideal to mitigate this threat, but this is unrealistic as finding a variety of examples of Visitor in open source is not easy, and each example found requires significant manual processing and the creation of a suitable test load.

The execution scenario employed in testing introduces another threat to the generalisability of results as it targets the parts of the software reliant on the pattern instance. An alternative approach is the use of the provided test suite, however, a threat to validity would remain as it is possible certain parts of the application are heavily tested though rarely used in practice. The realistic testing of software is a complex issue; in practice we would anticipate profiling software in deployment to target transformations on frequently used paths.

Nonetheless, JavaParser and CppParser include a typical use case of the Visitor pattern, and our results highlight the impact its removal can have on energy efficiency if the pattern is a core part of the application being used at run-time.

Conclusion Validity is concerned with our ability to draw a correct conclusion about the relation between our treatments and the data recorded during an experiment. For our experiments, we use the Wilcoxon rank-sum test to calculate statistical significance in energy consumption before and after the application is transformed. This test does not assume normality in the distribution of values, and the results of our experiments are ordinal and independent enabling our use of this test.

In sets of experiments where multiple comparisons are made, we use a Bonferroni correction to adjust our alpha levels appropriately to reduce the likelihood of Type I errors. Each set of experiments is executed 200 times to reduce the impact of potential noise or outliers in recorded data.

10 Future Work

There are several unexplored avenues of research in this field that are worthy of further work. Firstly, additional studies of open source examples of the Visitor pattern would further test our conclusions and would improve the generalisability of research results in this area. Specifically on the language front, our experiments were performed using Java and C++. Further explorations with software applications written in other languages would test the validity of the underlying theory that design patterns and object-oriented design in general may negatively impact energy efficiency. Our results indicate that the Visitor design pattern may cause excessive energy consumption if used on an execution path that is frequently traversed at run-time. We have not looked further at how automated support could help in the removal of the Visitor pattern. Research in the design patterns space has focused on the identification and application of design patterns, but it is clear that design pattern removal is an area that is also worthy of study.

Lastly, Gamma et al. (1995) described 23 design patterns in their original work, few of which have been explored in detail from an energy consumption perspective. There are also many other design patterns that are yet to be explored, even in a preliminary fashion. This presents a substantial amount of work to be done in the exploration of these patterns: theoretical considerations of their energy performance, exploratory studies, and finally, examination of the patterns in real-world software applications.

11 Conclusions

In this paper we have presented a detailed study of the energy performance of the Visitor design pattern. We initially used a small textbook-style example to explore the energy impact of the Visitor pattern. Three cases were considered: (1) the full pattern is applied, (2) the pat-

tern is removed entirely, and (3) an alternative pattern implementation is employed involving a type check and cast, which we termed *reflective dispatch*. Results when the JVM is operating normally revealed no difference between the patterned and the unpatterned versions of the software, but saw energy consumption reduce by over 7% when the reflective dispatch implementation was used.

We subsequently examined the impact of the Visitor pattern on the open source software library JavaParser, the first experiment looking at Visitor with a large open source project to our knowledge. These experiments yielded greater reductions in energy consumption than were seen in the textbook example, with reductions in energy consumption in most cases of almost 8% when the pattern is removed completely, and over 10% when the alternative, reflective dispatch, solution is employed.

To explore the pattern further, we examined its impact in the open source library CppParser. These experiments yielded even greater reductions in energy consumption when the pattern is removed, ranging from 28% to 66% depending on whether the visitor is interacting with all or few elements respectively. Notably, the alternate, reflective dispatch style of visitor drastically *increases* energy consumption when applied in C++, with energy increases ranging between 132% and 2012%.

We draw a number of conclusions from our results:

(1) More pronounced energy improvements were achieved in our real-world studies than in the textbook example in Java, suggesting that studies of real-world applications are essential, rather than relying only on small textbook examples as existing studies have done.
(2) In Java, greater reductions in energy consumption are typically achieved when the less common, reflective dispatch, approach is applied to the software. In C++ however, a reflective dispatch approach actually worsens energy performance. These divergent results indicate that in seeking energy-optimal solutions, we must be mindful that removing the pattern entirely may not be the best approach, and that the generalisability of findings across languages and applications must always be considered.
(3) In Java, in the scenario where the visitor interacts with every element in the tree, it transpires to be better to remove the pattern entirely rather than to employ reflective dispatch (the optimal approach in all other cases considered). This indicates that in optimising software for energy consumption, the best solution to choose may also depend on the nature of the load being processed.
(4) By way of summary conclusion, we observe that complete removal of the Visitor pattern improves the energy performance of software written in both Java and C++ in all scenarios. This supports the fundamental thesis that the indirection that is characteristic of object-oriented design in general, and design patterns in particular, contributes to an increase in energy consumption at run-time.

Appendix A: Textbook Example

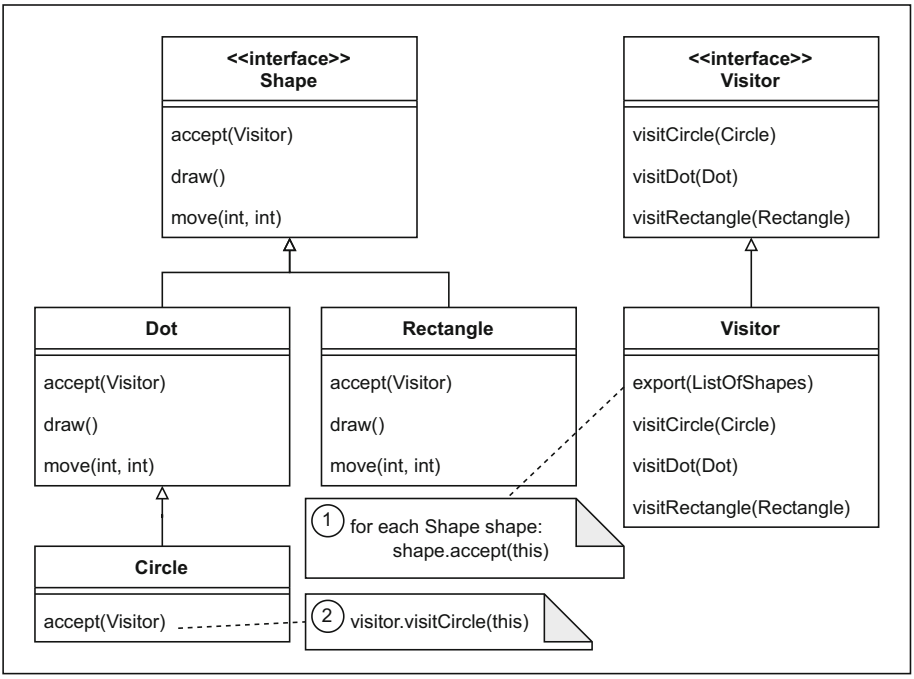

Fig. 5 Patterned textbook example

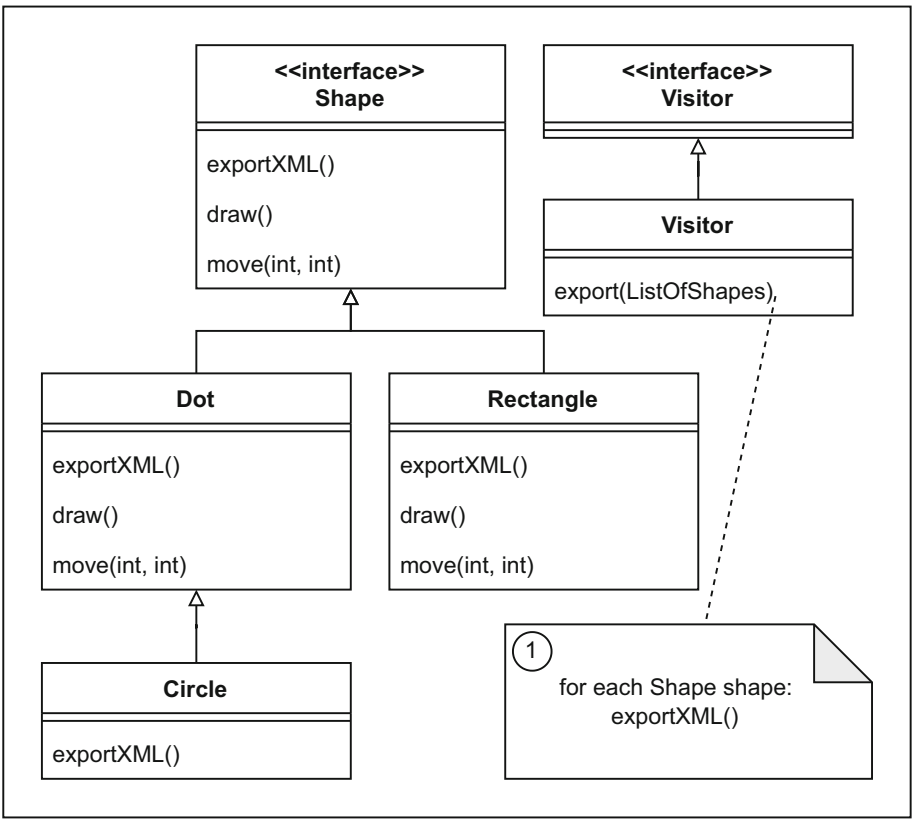

Fig. 6 Unpatterned textbook example

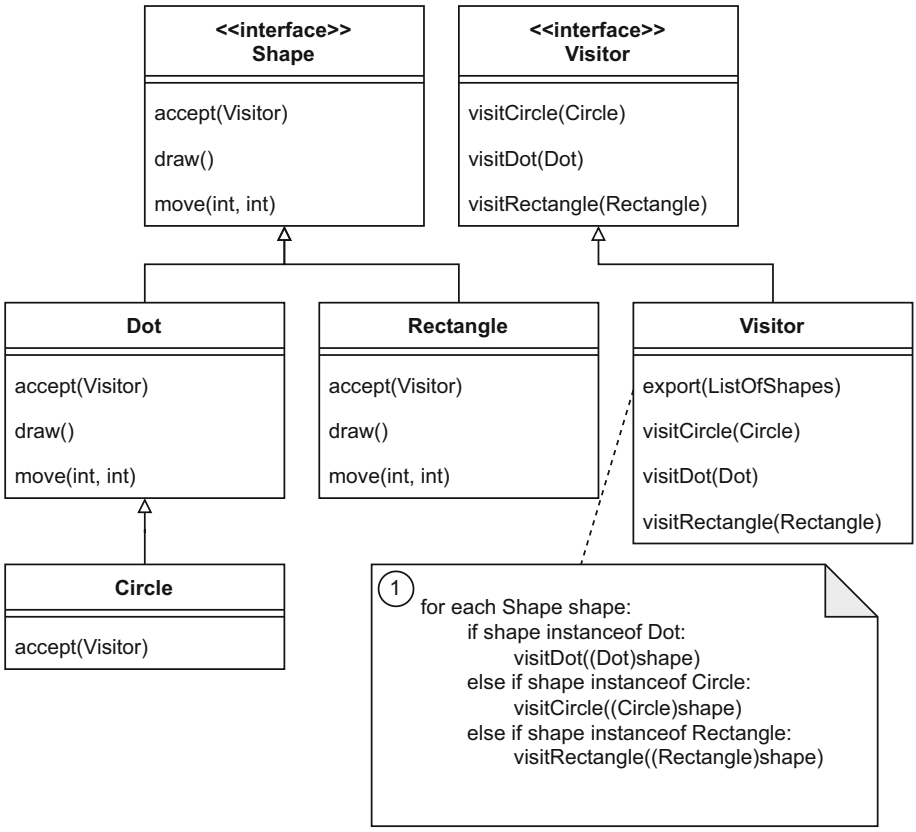

Fig. 7 Alternately patterned textbook example

Appendix B: JavaParser

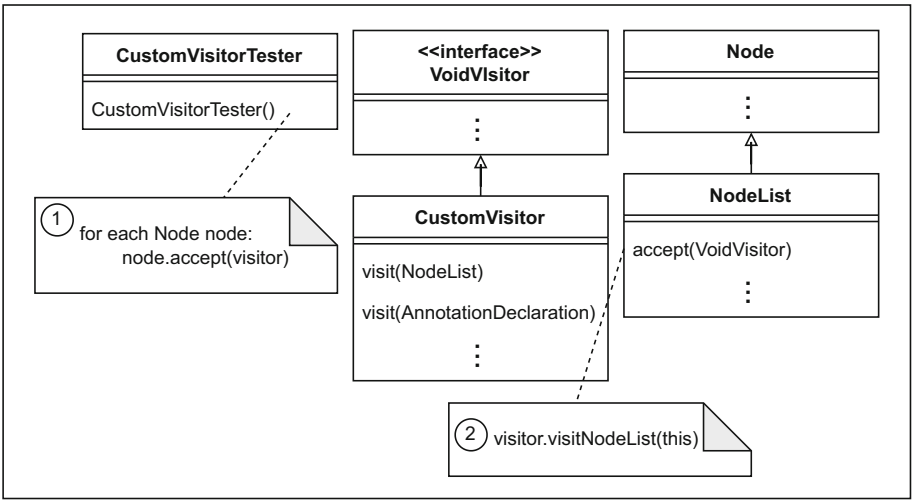

Fig. 8 Patterned JavaParser

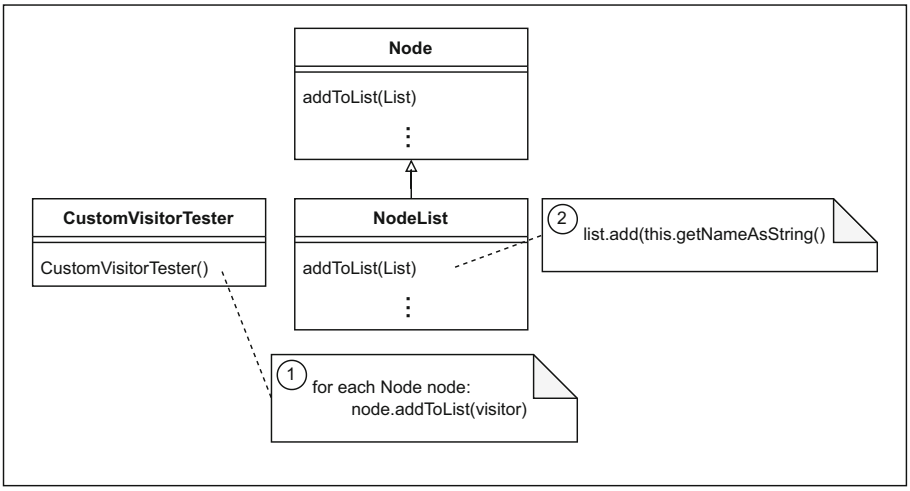

Fig. 9 Unpatterned JavaParser

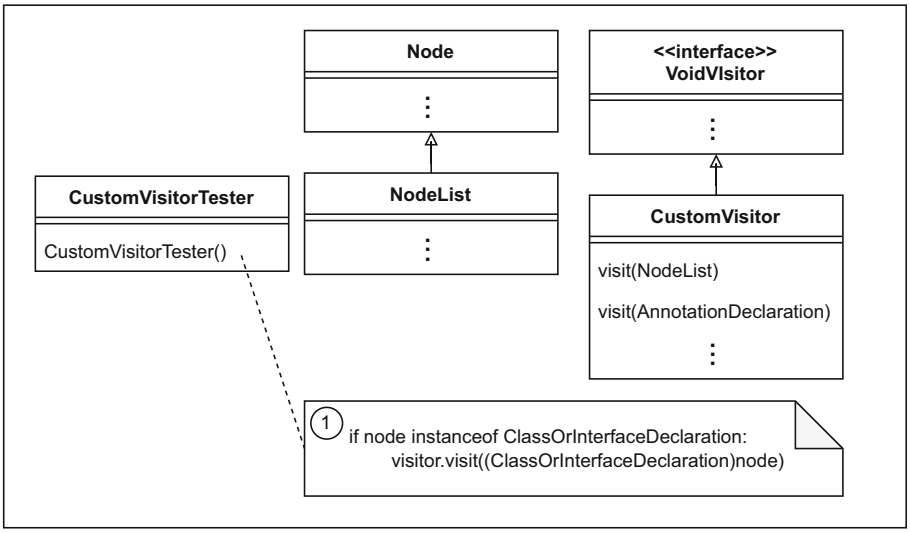

Fig. 10 Alternately patterned JavaParser

Appendix C: CppParser

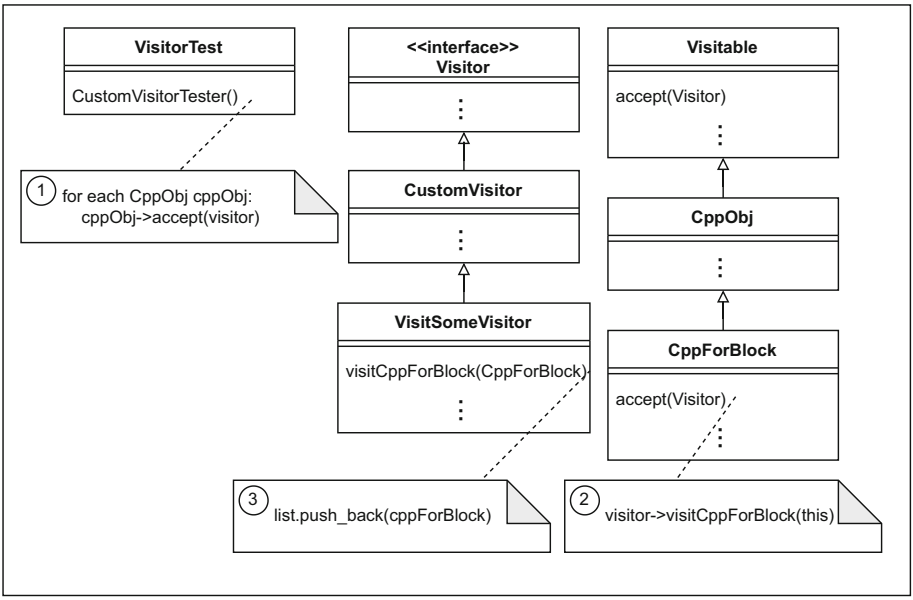

Fig. 11 Patterned CppParser

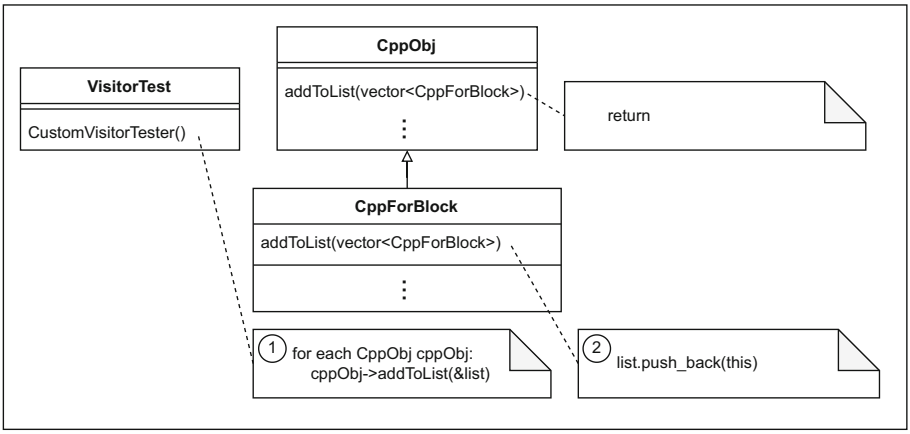

Fig. 12 Unpatterned CppParser

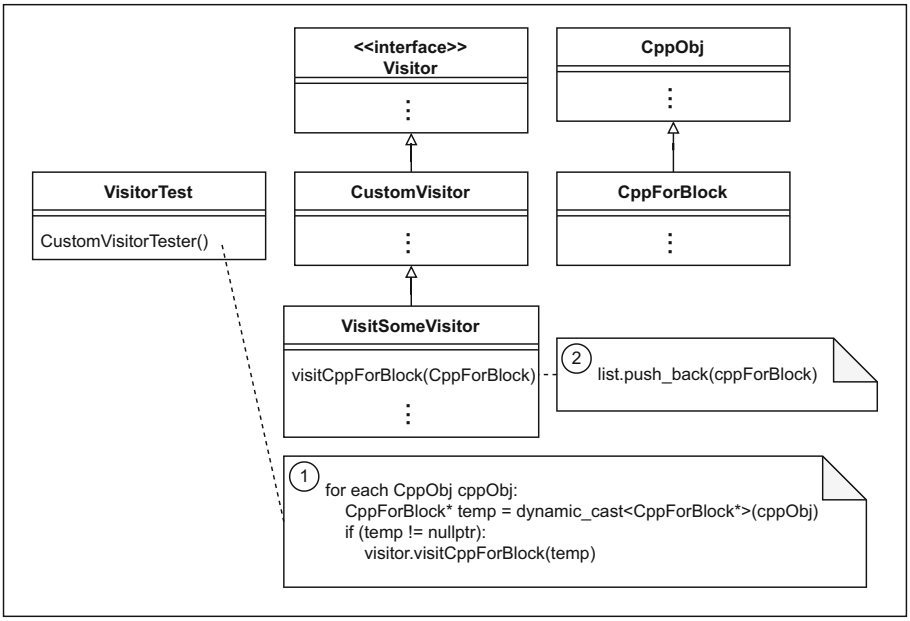

Fig. 13 Alternately patterned CppParser

Acknowledgements 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 Science Foundation Ireland Research Centre for Software.

Funding Open Access funding provided by the IReL Consortium

Data Availibility The software and resulting data generated in the development and execution of the applications are available in a public replication repository (Generated data sets/replication package 2023).

Declarations

Conflict of Interests The authors have no conflict of interest.

References

"Amazon best sellers: Best software reuse." [Online]. Available: https://www.amazon.com/Best-Sellers-Books-Software-Reuse/zgbs/books/4018

Abtahizadeh SA, Khomh F, Guéhéneuc Y-G (2015) "How green are cloud patterns?" in 2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC). IEEE pp. 1–8

Alexander C (1977) A pattern language: towns, buildings, construction. Oxford University Press

Andrae AS, Edler T (2015) On global electricity usage of communication technology: trends to 2030. Challenges 6(1):117–157

Avgerinou M, Bertoldi P, Castellazzi L (2017) Trends in data centre energy consumption under the european code of conduct for data centre energy efficiency. Energies 10(10):1470

Ayala I, Amor M, Fuentes L (2019) "An energy efficiency study of web-based communication in android phones." Sci Program 2019

Bunse C, Stiemer S (2013) "On the energy consumption of design patterns," Softwaretechnik-Trends: 33(2)

Büttner F, Radfelder O, Lindow A, Gogolla M (2004) "Digging into the visitor pattern," in Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004), Banff, Alberta, Canada, June 20-24, 2004, F. Maurer and G. Ruhe, Eds. pp. 135–141

Connolly Bree D, Ó Cinnéide M (2022) "The energy cost of the visitor pattern," in 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME) pp. 317–328

Connolly Bree D, Ó Cinnéide M (2021) "Automated refactoring for energy-aware software," in 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME) pp. 689–694

Connolly Bree D, Ó Cinnéide M (2020) "Inheritance versus delegation: which is more energy efficient?" in Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops pp. 323–329

Connolly Bree D, Ó Cinnéide M (2022) "Removing decorator to improve energy efficiency," in 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE

Couto M, Saraiva J, Fernandes JP (2020) "Energy refactorings for android in the large and in the wild," in 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER) pp. 217–228

Crestani A, Tetu R, Douin J-M, Paradinas P (2021) "Energy cost of iot design patterns," in 2021 8th International Conference on Future Internet of Things and Cloud (FiCloud). IEEE pp. 383–387

Cruz L, Abreu R (2019) Catalog of energy patterns for mobile applications. Empirical Softw Eng 24(4):2209–2235

Cruz L, and Abreu R (2017) "Performance-based guidelines for energy efficient mobile applications," in 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft). IEEE pp. 46–57

da Silva WG, Brisolara L, Corrêa UB, Carro L (2010) "Evaluation of the impact of code refactoring on embedded software efficiency," in Proceedings of the 1st Workshop de Sistemas Embarcados pp. 145–150

Feitosa D, Alders R, Ampatzoglou A, Avgeriou P, Nakagawa EY (2017) Investigating the effect of design patterns on energy consumption. J Softw: Evolution Process 29(2):e1851

Flucker S, Tozer R (2013) Data centre energy efficiency analysis to minimize total cost of ownership. Build Serv Eng Res Technol 34(1):103–117

Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley

Gamma E, Helm R, Johnson R, Vlissides J (1993) "Design patterns: Abstraction and reuse of object-oriented design," in European Conference on Object-Oriented Programming. Springer pp. 406–431

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Pearson Education

Generated data sets/replication package (2023). [Online]. Available: https://doi.org/10.6084/m9.figshare.22191283.v1

Georgiou S, Rizou S, Spinellis D (2019) Software development lifecycle for energy efficiency: techniques and tools. ACM Comput Surv (CSUR) 52(4):1–33

Gottschalk M, Jelschen J, Winter A (2014) "Saving energy on mobile devices by refactoring." in EnviroInfo pp. 437–444

Hasan S, King Z, Hafiz M, Sayagh M, Adams B, Hindle A (2016) "Energy profiles of java collections classes," in Proceedings of the 38th International Conference on Software Engineering - ICSE '16. ACM Press pp. 225–236

Hecht G, Moha N, Rouvoy R (2016) "An empirical study of the performance impacts of android code smells," in Proceedings of the International Workshop on Mobile Software Engineering and Systems - MOBILESoft '16. ACM Press pp. 59–69. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2897073.2897100

Hindle A (2015) Green mining: a methodology of relating software change and configuration to power consumption. Empirical Softw Eng 20(2):374–409

Hurbungs V, Bassoo V, Fowdur T (2022) Software design pattern on the edge. 2022 International Conference on Electrical. Computer, Communications and Mechatronics Engineering (ICECCME), pp 1–6

"Javasoft ships java 1.0," (1996) http://web.archive.org/web/20070310235103/http://www.sun.com/smi/Press/sunflash/1996-01/sunflash.960123.10561.xml, accessed: 2023-02-05

Kerievsky J (2005) Refactoring to patterns. Pearson Deutschland GmbH

Li D, Halfond WGJ (2014) "An investigation into energy-saving programming practices for android smartphone app development," in Proceedings of the 3rd International Workshop on Green and Sustainable Software - GREENS 2014. Press pp. 46–53. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2593743.2593750

Litke A, Zotos K, Chatzigeorgiou A, Stephanides G (2005) "Energy consumption analysis of design patterns," in Proceedings of the International Conference on Machine Learning and Software Engineering pp. 86–90

Maleki S, Fu C, Banotra A, Zong Z (2017) "Understanding the impact of object oriented programming and design patterns on energy efficiency," in 2017 Eighth International Green and Sustainable Computing Conference (IGSC). IEEE pp. 1–6

Malmodin J, Lundén D (2018) The energy and carbon footprint of the global ict and e&m sectors 2010–2015. Sustainability 10(9):3027

Mancebo J, Calero C, García F (2021) Does maintainability relate to the energy consumption of software? a case study. Softw Q J 29(1):101–127

Manotas I, Pollock L, Clause J (2014) "Seeds: A software engineer's energy-optimization decision support framework," in Proceedings of the 36th International Conference on Software Engineering pp. 503–514

Menghin M, Druml N, Preschern C, Steger C, Weiß R, Bock H, Haid J (2015) "Introduction of design pattern (s) for power-management in embedded systems," in Proceedings of the 18th European Conference on Pattern Languages of Program pp. 1–12

Morales R, Saborido R, Khomh F, Chicano F, Antoniol G (2018) Earmo: An energy-aware refactoring approach for mobile apps. IEEE Trans Softw Eng 44(12):1176–1206

Nayak J, Chandwadkar A (2021) "Green patterns of user interface design: A guideline for sustainable design practices," in International Conference on Human-Computer Interaction. Springer pp. 51–57

Noureddine A, Rajan A (2015) "Optimising energy consumption of design patterns," in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering 2:623–626 ISSN: 1558-1225

Paleczny M, Vick C, Click C (2001) The java hotspottm server compiler. Proceedings of the Java Virtual Machine Research and Technology Symposium 1:1–12
Palomba F, Di Nucci D, Panichella A, Zaidman A, De Lucia A (2019) On the impact of code smells on the energy consumption of mobile applications. Inf Softw Technol 105:43–55
Palomba F, Di Nucci D, Panichella A, Zaidman A, De Lucia A (2017) "Lightweight detection of android-specific code smells: The adoctor project," in 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER). IEEE pp. 487–491
Park JJ, Hong J-E, Lee S-H (2014) "Investigation for software power consumption of code refactoring techniques." in SEKE pp. 717–722
Pereira R, Couto M, Ribeiro F, Rua R, Cunha J, Fernandes JP, Saraiva J (2021) Ranking programming languages by energy efficiency. Sci Comput Program 205:102609
Pereira R, Couto M, Cunha J, Fernandes JP, Saraiva J (2016) "The influence of the java collection framework on overall energy consumption," in 2016 IEEE/ACM 5th International Workshop on Green and Sustainable Software (GREENS). IEEE pp. 15–21
Pérez-Castillo R, Piattini M (2014) Analyzing the harmful effect of god class refactoring on power consumption. IEEE Softw 31(3):48–54
Pinto G, Soares-Neto F, Castor F (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 pp. 29–35
Rodriguez A, Longo M, Zunino A (2015) "Using bad smell-driven code refactorings in mobile applications to reduce battery usage," in Simposio Argentino de Ingeniería de Software (ASSE 2015)-JAIIO 44 (Rosario, 2015)
Rodriguez A, Mateos C, Zunino A (2012) "Mobile devices-aware refactorings for scientific computational kernels," in 13Th Argentine Symposium on Technology, AST
Sahin C, Cayci F, Clause J, Kiamilev F, Pollock L, Winbladh K (2012) "Towards power reduction through improved software design," in 2012 IEEE Energytech pp. 1–6
Sahin C, Cayci F, Gutiérrez ILM, Clause J, Kiamilev F, Pollock L, Winbladh K (2012) "Initial explorations on design pattern energy usage," in 2012 First International Workshop on Green and Sustainable Software (GREENS). IEEE pp. 55–61
Sahin C, Pollock L, Clause J (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 pp. 1–10. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2652524.2652538
Schaarschmidt M, Uelschen M, Pulvermüller E, Westerkamp C (2020) "Energy-aware pattern framework: The energy-efficiency challenge for embedded systems from a software design perspective," in International Conference on Evaluation of Novel Approaches to Software Engineering. Springer pp. 182–207
Tonini AR, Fischer LM, Mattos JCBd, Brisolara LBd (2013) "Analysis and evaluation of the android best practices impact on the efficiency of mobile applications," in 2013 III Brazilian Symposium on Computing Systems Engineering pp. 157–158
Verdecchia R, Saez RA, Procaccianti G, Lago P (2018) "Empirical evaluation of the energy impact of refactoring code smells." in ICT4S pp. 365–383
Vetro A, Ardito L, Morisio M (2013) "Definition, implementation and validation of energy code smells: an exploratory study on an embedded system," in ENERGY 2013: The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies pp. 34–39
"Visitor in Java," refactoring.guru, accessed May 22 2023. [Online]. Available: https://refactoring.guru/design-patterns/visitor/java/example
Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in software engineering. Springer Science & Business Media