Bachelor Degree Project

Energy Consumption of Behavioral Software Design Patterns

Authors: Albert Henmyr, Kateryna Melnyk
Supervisor: Prof. Dr. Mauro Caporuscio
Semester: VT 2023
Subject: Computer Science

Abstract

The environmental and economic implications of the increase in Information and Communication Technology energy consumption have become a topic of research in energy efficiency. Most studies focus on the energy estimation and optimization of lower tiers of the hardware and software infrastructures. However, the software itself is an indirect driver of energy consumption, therefore, its energy implications can be to some extent controlled by the software design. Software design patterns belong to high-level software abstractions that represent solutions to common design problems. Since patterns define the structure and behavior of software components, their application may come at energy efficiency costs that are not obvious to the software developers. The existing body of knowledge on energy consumption of software design patterns contains a number of gaps, some of which are addressed within the scope of this thesis project. More specifically, we conducted a series of experiments on the estimation of energy consumption of Visitor and Observer/Listener patterns within the context of non-trivial data parsing in Python. Furthermore, we considered a Patternless alternative for the same task. Additionally, our measurements include runtime duration and memory consumption. The results show that the Visitor pattern led to the largest energy consumption, followed by Observer/Listener and finally the Patternless version. We found a strong relationship between runtime duration and energy consumption, thus coming to the conclusion that the longest-running pattern is the most energy-consuming one. The findings of the current study can be beneficial for Python software developers interested in the energy implications of software design patterns.

Keywords: *energy consumption, energy efficiency, software design patterns, Visitor, Observer*

Contents

1 Introduction

The Information and Communication Technology (ICT) sector represents a source of growing carbon emissions [1]. To offset this, at a society level, we can reduce our reliance on ICT; at a hardware level, we can design more efficient equipment; and at a software level, we can make our programs carbon-aware, postponing non-time critical operations until the system knows it is running on green energy [2]. Or, we can make the programs themselves less energy-consuming and more energy efficient.

This subject is more aggressively explored in mobile development, where battery usage is an important consideration, than it is when designing for stationary devices, where inefficiency 'merely' results in a larger electricity bill and sustainability may end up being ignored in favor of other quality attributes or simply development speed. In both fields, the concept of a design pattern [3] is known to developers as a way to describe time-tested solutions to common problems.

As far as non-mobile development goes, the energy efficiency of various design patterns has been examined for widely-used [4] languages such as Java and C++, yet Python has received little attention despite its popularity. The research will be focused on the implications of patterns' application in Python. Particularly, this thesis project will explore the CPU and DRAM energy consumption of the Visitor and Observer as one of the most extensively researched software design patterns in experimental studies [5], as well as runtime and memory use, and compare them to each other and the Patternless approach.

1.1 Background

Software design patterns are code abstractions that developers use as a shared vocabulary to express known ways to solve common problems. Observer and Visitor belong to the behavioral software design patterns that are focused on the dynamic interactions between the system components [3].

Observer allows an object to broadcast updates to interested subscribers. The main idea is to decouple the event/change handling logic from the code that causes the events or changes itself by having an abstract *Subject* that holds a certain state and abstract *Observers* that are notified by *Subject* about changes. A classic version of the Observer pattern also includes concrete classes that are the specific implementations of *Subject* and *Observer* abstractions [3]. Figure 1.1 illustrates the typical structure of the Observer pattern. We will be frequently referring to our Observer version as Listener [6–8] in this report due to the specifics of the experimental setup.

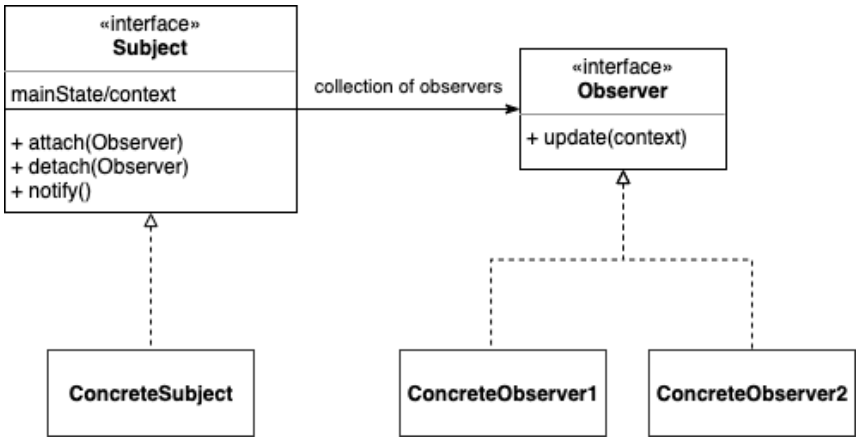

Figure 1.1: Observer UML

Visitor allows developers to gather algorithms in one place and automatically use the one that suits the object being visited. Gamma et al. [3] explain the application of Visitor with the example of "static semantic" analysis using abstract syntax trees (ASTs). They motivate that distributing the logic of performing operations on the AST across heterogeneous nodes is a rather questionable solution that potentially will lead to issues with the system's maintainability and modifiability. Therefore a separate entity called Visitor can be used to package related operations from each AST node and separate the algorithms from the objects on which they operate [3, 8].

Figure 1.2 represents the typical structure of the Visitor pattern. Classes that implement the *Element* interface accept a *Visitor* as a parameter. At the same time classes that implement the *Visitor* interface have *visit* methods specific to each *Element* variation. Subsequently, the respective *visit* method, which contains certain logic of what operations to do in the context of the current *Element*, can be invoked, once the *Visitor* is accepted. The *Client* class represents the context in which the *Visitor* interacts with the *Element* implementations on which it operates. [3, 8].

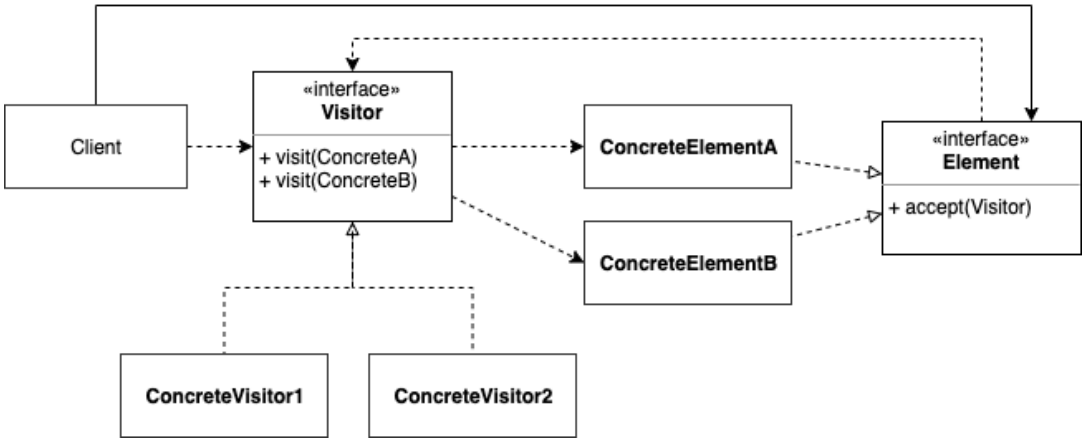

Figure 1.2: Visitor UML

Being common solutions, it is of interest to know what impact software design patterns have on energy consumption, speed and memory. The former is an aspect not well explored, especially in Python. Research in this area will aid developers in considering these dimensions when deciding on how to solve problems for which these patterns may be considered.

Python itself is not an energy-efficient language [9]. Given this, research in this area is important to help save what energy can be saved. Choosing between solutions in Python from the perspective of sustainability is a challenge, given that there seems to exist little to no published work in the field.

1.2 Related Work

Feitosa et al. deeply explored the energy consumption of the State/Strategy and Template patterns in Java [10]. They found that these patterns consumed more power than their non-pattern alternatives in most cases, and that the patterns mostly excelled in complex code involving many method invocations.

Bree and Cinnéide made a similar investigation into the Visitor pattern, also in Java [11]. Their conclusion was that using no pattern improved energy efficiency, but that a solution called reflective dispatch was even better if the load was sufficiently small. Reflective dispatch means that when iterating over visitable objects, the programmer directly

checks what type of class they are (using *instanceof* in Java) and treats them accordingly, instead of sending a Visitor.

The same two authors explored the Decorator pattern in Java [12]. Removal of the pattern from a textbook example reduced energy consumption by up to 96%, whereas in a more realistic scenario involving JUnit, they achieved 3–5% reduction by removing just one instance of Decorator.

Bunse and Stiemer investigated Facade, Abstract Factory, Observer, Decorator, Prototype and Template Method in Java, in the context of mobile software development [13]. Comparing each to its non-pattern counterpart, they found that the patterned versions of Decorator, Prototype and Abstract Factory were about 133%, 33% and 15% worse, respectively. This is with regards to both execution time and energy consumption. The other three patterns exhibited no significant difference.

Sahin et al. examined 15 design patterns in C++, comparing them to unpatterned approaches [14]. The worst ones were Decorator, Observer and Abstract Factory, exhibiting roughly a 713% [sic], 62% and 22% energy consumption increase respectively. Flyweight and Proxy were the most prominently positive with a 58% and 36% decrease, again respectively.

1.3 Problem Formulation

The energy consumption of design patterns has received a considerable amount of attention for both C++ and Java. These two languages are, along with C and Python, among the most popular in the world [4], yet Python is highly lacking when it comes to such research.

We will help fill this gap by examining Observer and Visitor, the two most popular behavioral design patterns [5]. We will compare them to each other and a Patternless approach (algorithm sprawl), in which the nodes themselves contain the algorithms, as advised against by Gamma et al. [3] in our Visitor description in the Section 1.1. Note that Patternless is not the name of a common design pattern; it is merely a label by which we refer to this solution. We describe it in detail in the implementation section and clarify how it differs from the other two.

There are many ways in which a solution can be evaluated. Two of these are readability and maintainability, expressing how easy the code is to work with. While important, these attributes are not easily quantified. On the other hand, memory usage (space complexity) and runtime (time complexity) can be easily determined, and these are typical aspects to consider. Therefore, we will measure these, along with the energy consumption that is of primary interest. Knowing all three will help inform developers about tradeoffs involved. Therefore, we will answer these questions:

1. How do Observer, Visitor and Patternless compare for energy consumption?
2. How do Observer, Visitor and Patternless compare for execution time?
3. How do Observer, Visitor and Patternless compare for memory usage?
4. Do different data set sizes favor different patterns?

1.4 Motivation

The motivation of our work from the scientific perspective is associated with the gap in the existing body of knowledge on energy consumption of software design patterns in the Python context.

Our work has industrial relevance as it aids in reasoning about design choices. This can lead to reduced system energy consumption, or to the insight that because a necessary solution will require a certain amount of energy, there will be that much less available for the rest of the system to use, assuming some kind of constraint on total usage. Considering the software design process, the more information available for a decision, the better the result should be. We thus contribute to the general challenge in this field.

Society will always benefit from reduced software energy consumption. Therefore, we expect our work within the scope of this thesis project to be beneficial, even if to a modest extent.

1.5 Results

In this thesis project, we measure the energy consumption, execution time and memory usage across three versions (Patternless, Visitor, Observer) of the same functionality. The measurements are taken over the course of 25 runs for the same input size for each version. Our results from these experiments reveal a strong relationship between runtime duration and energy consumption for all three versions. The general picture shows that the Visitor pattern leads to the largest energy consumption, followed by Observer/Listener and finally the Patternless version. On the other hand, the memory measurements do not show significant differences across the three versions and stay approximately equal for the same input sizes.

1.6 Scope/Limitation

We limit ourselves to comparing Observer, Visitor and the Patternless approach.

We compare the patterns in data parsing tasks involving abstract syntax trees constructed with ANTLR [15]. While there are many open source programs we could use, we have restricted ourselves to this because it is easy to induce heavy usage of Observer, Visitor, and no pattern in this scenario. This gives us confidence that differences in our measurements are the result of pattern changes.

Note that Observer in this context means observing an object called a *ParseTreeWalker* as it walks the tree. It does not mean subscribing to individual nodes for updates.

1.7 Target Group

Our research is suited for Python developers and architects and anything in between, essentially anyone empowered to decide on software design. As we explore energy consumption along with time and memory usage, our research should help make informed decisions in situations where these may relate to constraints or important quality attributes.

1.8 Work Organization

This thesis project represents the collaborative efforts of Albert Henmyr and Kateryna Melnyk. The conceptualization and majority of the theoretical background, development of the experimental design (methodology) and the basic implementation with the subsequent refinements of the testing prototypes are contributed by Kateryna. Refinements of the testing prototypes, preparation of the experimental setup including the testing data division and finally the implementation of the alternative Patternless version are credited to Albert. The actual experiments were run and verified by Albert. The raw results of the conducted experiments were organized and visualized for the subsequent analysis by

Kateryna. Analysis of the experimental results and the writing of the thesis report are equally contributed by the two authors.

1.9 Outline

This report covers the following sections. The methodological framework, research methods (experimental research), reliability and validity of the current study as well as ethical considerations are discussed in Section 2. Section 3 provides context for the conducted experiments by overviewing the related theoretical background. Section 4 describes the concrete implementation steps that were taken while preparing the experimental setup and conducting the actual experiments. The results of the conducted experiments are consolidated and organized in Section 5. Further on, Section 6 analyzes the obtained experimental findings by using statistical methods among other things whereas Section 7 delves deeper into the discussion about the generalizability, relation to other studies and comparison of these findings. Section 8 finalizes the report by summarizing the answers to the research questions and provides insights into potential areas of future research.

2 Method

The methodology employed in this project is the subject of this section. It outlines the taken experimental design decisions and planning associated with them. Subsequent sections will delve into the experiment implementation in more detail.

2.1 Research Project

In order to address the research gap identified in the previous subsection, we need to gather and analyze the data about the energy consumption of Visitor and Observer software design patterns in the context of Python. Moreover, the data itself will only be of usage if compared to that of its design alternative (our Patternless approach). Furthermore, it is not feasible to evaluate software design patterns in isolation; rather some functionality that heavily utilizes them is required as well as control over this functionality in order to get accurate measurements. Therefore the assessment of the energy consumption of the Visitor and Observer design patterns implies a set of activities that comprise the essence of the experimental research method.

As Wohlin et al. [16] stated, a complete systematic literature review is not needed in the context of experimental research, although being systematic in analyzing the related work is rather beneficial for the design of the experiment. Thus literature research as part of the project methodology will set the stage for the subsequent empirical activities and outline the methodologies of the previous related works in the Theoretical Background subsection.

When it comes to the experiment, the main interest involves the cause-effect relationship between a method (called a factor) and attribute of interest as specified by Zelkowitz and Wallace [17] and Basili [18]. In other words, the project's experimental setup will involve two kinds of variables: independent (a factor that can be manipulated) and dependent (changes in response to independent variable) variables [16]. The general design and flow of our experiment will focus on the following independent variables: 1) type of design pattern (no pattern version included) and 2) size of input data. The set of dependent variables of interest includes the following: 1) energy consumption, 2) memory use, and 3) runtime duration.

2.2 Research Methods

We will run an experiment to determine the difference in energy consumption, execution time and memory usage of using Visitor, Observer and Patternless (henceforth "the patterns") for the same task. As these are highly measurable aspects and we can control the independent variable, we see little merit in using other research methods such as surveys and interviews, expecting them to yield highly subjective results, if yet perhaps more generalizable. To our knowledge and based on our searches, the energy consumption of software design patterns in Python - our most important measurement - is an unexplored field in academia, so there is nothing to base a large-scale literature review on.

We will use ANTLR 4.12.0 [15] in Python 3.10.6 to facilitate our experiment. ANTLR is a tool for generating abstract syntax trees (ASTs) that can be traversed using Visitor, Observer or no pattern. In the first two cases, interfaces are provided, to which we will write our own implementations. For Patternless, we will insert our own traversal code into the tree.

The tree will represent syntax to parse research publications from the Linnaeus University DiVA portal [19]. The data of research publications will have the form of comma-

separated values in CSV documents; the tree for each file will simply be a Python program in RAM. A line in a CSV file may look like the following example in Table 2.1:

Table 2.1: DiVA Publication CSV Entry Example

```
"1656737","Pagliari, Lorenzo (Gran Sasso Science Institute,
Italy);D'Angelo, Mirko [midaab] (Linnéuniversitetet [4853],
Fakulteten för teknik (FTK) [12354], Institutionen för
datavetenskap och medieteknik (DM) [879987]);Caporuscio,
Mauro [macaab] (Linnéuniversitetet [4853], Fakulteten för
teknik (FTK) [12354], Institutionen för datavetenskap och
medieteknik (DM) [879987]);Mirandola, Raffaela (Politecnico di
Milano, Italy); Trubiani, Catia (Gran Sasso Science Institute,
Italy)","Performance modelling of intelligent transportation
systems :  Experience report","Konferensbidrag","Refereegranskat",
"eng","","","","","","","ICPE '21 :  Companion of
the ACM/SPEC International Conference on Performance
Engineering","155","160","2021","","","","Association for
Computing Machinery (ACM)","","","","9781450383318","","",
"10.1145/3447545.3451205" ,"","","2-s2.0-85104943502",
"urn:nbn:se:lnu:diva-112446","","","Intelligent Transportation
Systems;Model-based Performance Analysis;Petri Nets;Intelligent
systems;Experience report;Performance modelling;Physical
systems;Intelligent vehicle highway systems", "Kommunikationssystem
(20203)","Computer and Information Sciences Computer Science",
"","","<p>Modern information systems connecting software, physical
systems and people, are usually characterized by high dynamism.
These dynamics introduce uncertainties, which in turn may harm
the quality of systems and lead to incomplete, inaccurate, and
unreliable results.  To deal with this issue, in this paper
we report our incremental experience on the usage of different
performance modelling notations while analyzing Intelligent
Transportation Systems.  More specifically, Queueing Networks
and Petri Nets have been adopted and interesting insights are
derived.</p>","","","","","","","","","","","","","2022-05-06",
"2022-05-06","2022-05-06"
```

We will use CSV files containing 1 000, 2 000, 4 000 and 8 000 lines, excluding the header. Note that each line will result in its own subtree. Thus, larger data sets will not produce a deeper tree, but a wider one. The root of the tree is a node that contains no data, but has the header, each row and the end-of-file token (EOF) as its children.

Using each of the patterns, we will traverse the tree and assemble a Python dictionary mapping research institutions and organizations to the number of papers they are associated with, by merit of having a contributing researcher represent them. Having multiple researchers from the same institution will not cause it to be counted additional times for any one paper. Figure 2.3 exemplifies the part of the AST with the parsed information about the authors and authors' affiliations (by no means this example is a complete AST of one publication record from the CSV file).

Before measuring, we will verify that all three patterns produce the same dictionary, also inspecting it to make sure it looks reasonable. We cannot make a fair comparison if the patterns do not arrive at the same results.

The test program will have two parts: building the tree and traversing it. We will run measurements on the traversal only. Each version of the program will only have the functionality required to run with one pattern, so as to eliminate any risk of redundant functionality requiring resources and making a pattern seem slower or hungrier.

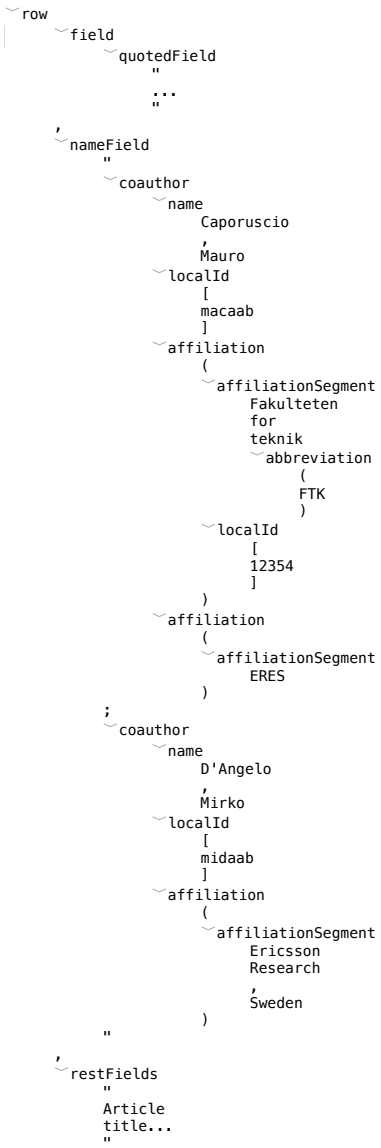

Figure 2.3: Illustrative Abstract Syntax Tree Example

We will use pyRAPL [20] to simultaneously measure energy consumption (CPU and DRAM). It is based on RAPL [21], which has been found to be reliable [22–24], and will give us high confidence in our results. Each experiment will be repeated 25 times to give us sufficient data to analyze. Note that pyRAPL will calculate the CPU and DRAM energy consumption of the entire system, not just our program, so a large sample size is needed to filter out outliers caused by background processes [20]. We will use RAPL itself to estimate the baseline consumption of the system and compare our results against this.

For measuring memory usage, we will use memory-profiler (a module for monitoring memory usage of a Python program) [25]. Unlike pyRAPL, this will be isolated to only what our program does and not the memory occupied by the entire system. We expect there will be little to no fluctuation involved, but we will still repeat each experiment 25 times.

Lastly, the Python *time* library will be used for execution time. While pyRAPL does keep track of time spent, it is the time spent measuring, not necessarily the time spent executing, even though we limit the measuring to our function.

All in all, this means a total of nine experiments per CSV file, separating the three

measurement methods to minimize overhead. As mentioned, our data sets will be of sizes 1 000, 2 000, 4 000 and 8 000. For each of the patterns Observer, Visitor and Patternless and each data set size, we will:

- Measure energy consumption (CPU and DRAM) 25 times
- Measure memory usage 25 times
- Measure execution time 25 times

The measurements will cover only the traversal of our abstract syntax tree to assemble the dictionary. The building of the tree is not included. The tree used in each experiment will not be burdened by containing the functionality required by the two patterns not currently being tested. While the measurements will all be performed 25 times, they will not be done together due to overhead.

All experiments will be run on a laptop with these specifications:

- Operating System: Ubuntu 22.04.2 LTS
- Kernel: Linux 5.19.0-40-generic (x86_64)
- Memory: 8GB RAM
- Processor: Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz

2.3 Reliability and Validity

In this section, we outline various potential threats to the validity and reliability of our experiment and detail the measures we have taken to mitigate their impact. According to the categorization represented in [16] the validity threats can be viewed in the context of:

- conclusion validity is concerned with the ability of the researchers to arrive at correct conclusions [16];
- internal validity specifies whether there are any interfering factors that can affect the relationship between independent and dependent variables [16];
- construct validity defines whether a method measures what it was supposed to measure, meaning that the main threats to construct validity are the threats related to the design of the experiment [16];
- external validity is concerned with whether the research results can be generalized outside the context of the current study [16].

Threats to conclusion and internal validity are firstly associated with the measurements of energy consumption using the RAPL technology [20, 21]. The recorded measurements will include not only the energy consumption of the patterned or unpatterned processes but also the other processes that were running during the same period of time. Therefore mitigation of energy noise can be achieved with the trivial shutting down of any extra applications that may distort the measurements of the main process of the experiment interest [20]. Moreover, each combination of independent variables will be run a sufficient number of times, thus providing enough data for realistic average results that are approximated to the real state of things as closely as possible.

On top of this, it is certainly possible that our implementations of Visitor, Observer and Patternless are not optimal. If written by someone more experienced with Python and/or ANTLR, a fairer comparison could be expected. While we deliberately make every pattern process every node for simpler comparison, our usage of sets and dictionaries could be refined.

A smaller concern could be our ANTLR grammar, which dictates how the tree is built. We consider it smaller because all three patterns have to deal with the same tree, and we only organize the tree to the extent that we can use it to solve the task described in 2.2: Tallying university contributions to each paper. That means we do not make our program create nodes for publication date, abstract, keywords, etc., but rather let these stay lumped together as a large chunk of text in a single node. By not subdividing them, we not only save the program unnecessary work when building the tree, but we also do not turn one uninteresting node into many still-uninteresting nodes that each would have to be explored by the patterns.

As software design patterns do not exist in isolation, but are part of some functionality that employs them, the measurements of the energy consumption of the Visitor and Observer design patterns are in essence the measurements of some applications. This fact could be a potential threat to the construct validity due to the assumption that the design of the experiment is not constructed to narrow down the measurements to the specific lines of code that represent pattern. However, a software design pattern is the way to design functionality, therefore it is integral to it and should be researched in the respective application context.

Our experiment measures the energy consumption of the Visitor and Observer design patterns when traversing abstract syntax trees using Python because of how heavily this traversal is dependent on patterns. It may seem that such research could be considered a threat to external validity due to the narrow application scope. Nevertheless, our findings can provide insights into the energy consumption of Visitor and Observer design patterns for the cases where the patterns are planned to be heavily utilized. Therefore the generalizability of the experiment lies in the fact that it is not about design patterns as applied to the traversal of the ASTs but about the multiple frequent calls to the pattern functionality.

The main threat to reliability is associated with the reliability of energy measurements because as mentioned earlier, RAPL technology records the global energy consumption of all running processes, therefore the exact replication of the energy consumption in the other similar setting will most likely not be the same. Nevertheless, under otherwise equal conditions including an absence of extreme environmental conditions that can affect the hardware, the overall ranking of the patterns' energy usage should be in line with our findings.

2.4 Ethical Considerations

As this is an experiment involving no human subjects, there are no ethical considerations worth mentioning. The data we use are publicly available. The libraries we rely upon are free and we are using them for non-profit purposes. Anything involving bias will in this case be a validity concern.

3 Theoretical Background

In order to provide the theoretical context for our experimental research, this section starts with the motivation behind the estimation of energy consumption of the ICT sector; further, the focus is narrowed down to the research of software energy consumption. Finally the software design patterns, as high-level abstractions of the software design stack, are explored from the energy efficiency point of view.

3.1 Software Energy Consumption

Over the past 70 years, the ICT sector has grown to the point where its significance raises the question about the impact that it has, in addition to the increase in energy costs [26], on the environment. Freitag et al. [27] examined peer-reviewed sources that research greenhouse gas (GHG) emissions, which generally estimate that ICT is responsible for 1.8%–2.8% of yearly global emissions. However, Freitag et al. provide arguments for a different estimate ranging from 2.1% to 3.9%. On the surface, both value ranges do not seem to be critically large, but the biggest concern lies in the fact that ICT emissions will inevitably continue to grow [27].

The issue of GHG emissions is not a standalone factor, but a result of energy generation and consumption. However, according to Krey et al. [28] the main driver of GHG emissions is still energy consumption because it creates the demand for energy generation and raises the question of energy efficiency by the end consumer.

Georgiou et al. [29] analyzed research studies concerning energy efficiency throughout different stages of the software development lifecycle and emphasized GreenIT as the way to address the issues of ICT's increase in energy consumption. However, the sustainable practices of GreenIT were mainly considered with the hardware level (e.g. energy optimizations of CPU) [29, 30].

Along with the work by Georgiou et al. [29], Pinto and Castor [31] state that the majority of the research related to energy efficiency in computing has been focused on the lower tiers of the hardware and software infrastructures, and emphasize the importance of software energy efficiency research due to the fact that software is an indirect driver of energy consumption and there exists substantial evidence that energy consumption can be impacted by the software design itself [29, 31].

According to the surveys by Pinto and Castor [31] and Pang et al. [32] regarding developers' knowledge of software energy consumption, the respondents had insufficient understanding of energy efficiency and the energy implications of the decisions taken during the software development process due to a lack of studies that focus on energy-aware software.

Despite the focus on energy optimization at the hardware level, the following studies represent the research interest in the estimation of the energy consumption of the software stack.

Bozzelli et al. [33] described metrics of software energy consumption through a systematic literature review of related primary studies. Among other units, the list includes joule (J) as the derived unit of energy and watt (W) as the unit of power that generates or consumes 1 joule in 1 second.

Couto et al. [34] classified 10 programming languages based on their energy efficiency, which was measured using the RAPL technology. They used a suite of small self-contained programs as their benchmarking dataset and monitored CPU energy as well as runtime.

Similarly, Pereira et al. [9, 35] continued research by analyzing the energy efficiency of 27 programming languages and its correlation with memory and runtime using Couto et al. [34] methodology as the basis and adding DRAM energy and memory consumption measurements. They extended their benchmarking dataset with the Rosetta Code chrestomathy repository for [9].

As has already been mentioned, Georgiou et al. [29] categorized studies within the period from 2010 to 2017 related to energy efficiency in the different stages of the software development cycle. One of the aspects of their categorization is the empirical evaluation of design patterns and their optimization, as design patterns define software components and their interactions that may affect the energy consumption of the programs in which the patterns are applied.

Moreover, Moises et al. [36] conducted a literature review to examine the studies that provide the practices of sustainable energy consumption and concluded that, among the practices related to CPU and memory usage, choosing energy-efficient software design patterns as well as identification of the energy efficiency of design patterns could be applied to the sustainable development process.

3.2 Research Interest in Software Design Patterns

Feitosa et al. [37] authored a chapter on patterns and energy consumption with a focus on mobile applications in a book that overviews software sustainability from different perspectives. Similar to the studies in the previous subsection, the authors emphasize that ICT energy consumption, as a many-sided problem, should be researched not only on the hardware but also on the software levels. The chapter begins with an introduction that highlights the importance of patterns as abstracted solutions to recurrent problems. The authors divide patterns into patterns that specifically address energy issues and patterns that indirectly affect energy efficiency. According to such categorization, the software design patterns as defined by Gamma et al. [3] belong to those that can potentially affect energy consumption due to the number of created objects and invoked methods. Therefore the implications of software design patterns application need to be studied [37].

Zhang and Budhen [5] conducted a web-based survey to investigate the usefulness of software design patterns from the points of view of developers and maintainers. They noticed an interesting conformity; the three most extensively studied patterns (Composite, Observer and Visitor) appeared among the most highly favored patterns according to the survey results, though respondents' reactions about Visitor were slightly controversial due to difficulties with understanding the pattern's abstraction and relative complexity [5].

3.3 Energy Estimation of Software Design Patterns

In this subsection, we will take a look at the existing primary studies that have empirically measured the energy implications of software design patterns, as defined by Gamma et al. [3].

3.3.1 Embedded Systems and C++

Sahin et al. [14] researched the energy consumption of 15 software design patterns, with 5 patterns in each category (creational, structural, and behavioral) applied to different programs running on the Spartan-6 board. The choice of patterns was mainly dictated by the availability of testing code having patterned and unpatterned versions. The authors developed a custom hardware system based on an FPGA board with multiple power monitors

that allowed them to easily measure the power consumption of individual components (CPU, memory, etc.). Moreover, measuring patterns on the Spartan-6 board eliminated the effect of operating system processes because the board is an embedded system. The main power measurements (wattage) of applications with and without having patterns were subsequently converted into energy (joules) and served as the basis for finding the total energy difference between them as well as the energy difference per iteration and other indicators. As a result, Flyweight and Proxy showed energy-saving results, whereas Decorator, Observer and Abstract Factory significantly increased energy usage [14].

Litke et al. [38] conducted an experimental study on the energy consumption of 5 design patterns (Factory Method, Adapter, Observer, Bridge and Composite) using 6 code examples in C++ in the context of an embedded system. Their energy consumption research consisted of a compiled C++ code analysis where they examined the use of specific instructions and memory access operations, and from that derived the total energy measurements for CPU and memory. The results showed that Factory Method and Adapter pattern had minimal consumption overhead [38].

3.3.2 Mobile Platforms

A study conducted by Bunse and Steimer [13] researched the effect of 6 design patterns (Facade, Abstract Factory, Observer, Decorator, Prototype, and Template Method) on the energy consumption and runtime of Java applications (small applets based on textbook pattern implementations and no pattern applets with equal functionality) on Android platforms. They used the PowerTutor diagnostic software tool for Android-based mobile platforms. The experimental results showed that the patterned versions of Decorator, Prototype and Abstract Factory were about 133%, 33% and 15% worse (with regards to execution time and energy consumption), respectively. The other three patterns exhibited no significant difference [13].

3.3.3 Java and Patterns in the Open Source Software

Feitosa et al. [10] investigated the energy consumption of State/Strategy and Template design patterns in the context of pattern-participating methods of 2 Java open-source software systems (JHotDraw and Joda Time) compared to functionally equivalent alternatives. The pattern-relevant methods were detected with a tool that recognizes the presence of a pattern structure with the help of a similarity scoring algorithm. PowerApi and Jalen were used to measure the energy consumption of the CPU. pTop was used to confirm the accuracy of these measurements. The results showed that alternative versions are more energy efficient, however, patterned versions demonstrated similar or lower energy usage in complex cases involving many method invocations [10].

Bree and Cinnéide [11] investigated the energy implications of the Visitor design pattern applied to Java programs. The implementation of Visitor in Java requires two single dispatches (accept method call and visit method call), therefore the authors were interested in the energy costs of such behavior. The study consisted of measuring the textbook pattern examples as well as the Visitor pattern in the context of traversing ASTs of parsed open source software source code (JHotDraw) with the help of JavaParser. The experiments with textbook examples were aimed to measure the following three cases: (1) patterned version, (2) unpatterned version, and (3) alternatively patterned version they called reflective dispatch (instead of sending a visitor, use Java's *instanceof* to find the object's class). At the same time, the same concept was applied to traversing the JHotDraw source code AST: (1) traversal of the tree with the default Visitor implementation

that visits every node of the tree, (2) unpatterned traversing of the AST, and (3) alternative patterned version which checks the type of the tree node and decides whether to visit it (reflective dispatch). The Wattsup Pro Power Meter hardware was used in the experimental measurements. As for the textbook examples, the results showed a reduction of over 7% in energy consumption when using reflective dispatch. In the case of the open-source software (JHotDraw) and JavaParser, the complete removal of Visitor from the AST traversal resulted in an 8% reduction in energy consumption whereas application of reflective dispatch version led to a 10% decrease [11].

The same two authors examined the energy implications of the Decorator structural software design pattern in Java [12]. Bree and Cinnéide were interested in investigating the energy cost of the indirection that comes with the application of Decorator. The study investigated Decorator in the context of textbook examples and a JUnit (unit testing framework in Java) case study. In general, the experiments showed that the removal of the pattern led to a 96% decrease in energy consumption in the textbook example measurements and a 5% decrease in the JUnit case [12].

3.3.4 Further Pattern Energy Estimation Studies

As part of their research on the improving energy efficiency of software design patterns through compiler optimization, Noureddine and Rajan [39] measured the CPU energy overhead (program with pattern compared to no pattern version) of 14 patterns in C++ and 7 patterns in Java running on a Lenovo Thinkpad X220. They used the Jolinar 2 tool, which is based on the Intel RAPL technology, for their measurements of energy consumption. The experiment showed that the application of Observer, Decorator and Mediator resulted in an energy overhead of over 10% [39].

Maleki et al. [40] investigated the impact of object-oriented concepts (inheritance, polymorphism, dynamic binding, and overloading) and some software design patterns (Decorator, Facade, Flyweight, Prototype, and Template method) on the performance and energy consumption of software. In the case of the pattern experiments, the authors measured and compared metrics of the patterned and unpatterned versions with the same functionality. The experiments ran on a high-performance computing system codenamed Marcher. Maleki et al. developed a power and energy measuring API that uses the techniques of Intel RAPL interface among other tools. Concerning the patterns, the experiments showed that the Flyweight pattern can improve energy usage along with performance whereas the application of Decorator has a negative impact on energy and performance metrics [40].

3.4 Application of Software Design Patterns for Parsing Tasks

According to Ortin et al. [41], the concept of parsing in software development can be applied in multiple areas such as compiler implementation, processing data of various formats, natural language processing, etc. The general idea of parsing of any kind is to enable a syntactic analysis by conforming to the rules of the predefined grammar and building a parse tree that represents a structuring of the input according to the grammar rules. Further on, by having the tree organized according to the grammar it is possible to interact with it. The authors mention that the importance of parsing in software development has led to the existence of a variety of different parsing tools (parser generators) that facilitate the process of parser generation and come with certain tool-specific ways in which to use the generated parser. The authors highlight ANTLR [15] to be among the most commonly used parser generators [41].

The creator of ANTLR, Terence Parr, in his book about the ANTLR v4 [42] states that ANTLR supports 2 main mechanisms of parse tree traversal that both walk the tree using depth-first search: Listener and Visitor.

As for the Listener implementation, ANTLR uses it as the default mechanism of parse tree traversal. ANTLR provides a *ParseTreeWalker* class that acts like the observable entity walking through the structure of the tree. *ParseTreeWalker* starts at the root and goes down recursively in a depth-first search as vizualized in more detail in Figure 3.4 . At the same time, ANTLR generates a grammar-specific Listener class (that can be extended and modified by the developer according to certain tasks) that eventually observes the traversing activity of the *ParseTreeWalker*, meaning that *ParseTreeWalker* eventually invokes the respective callbacks of the Listener class once *ParseTreeWalker* enters or exits a node of the tree [43]. Terence Parr writes about the beauty of this approach, emphasizing that the developer does not have to write some custom implementation that is responsible for traversing the tree (*ParseTreeWalker* already does it). The Listener mechanism observes the updates of an entity that automatically walks the tree [42].

On the other hand, more control over the actual traversal activity can be gained using the Visitor mechanism that explicitly performs the whole tree-walking sequence by invoking the visit methods of the respective children on each tree level. Each node having an *accept()* function for a visitor is in line with the Visitor software design pattern specification by Gamma et al [3]. In the same manner as with the Listener, ANTLR generates a grammar-specific Visitor class whose methods can be modified by the developer [42].

The main difference between these 2 main mechanisms lies in the fact that in Visitor the developer can control the tree-walking process by explicitly guiding the visitation order whereas the Listener approach just reacts to the changes/events that are happening in the *ParseTreeWalker* activity. Despite these differences, both approaches make use of the depth-first search and are heavily utilized as ways to interact with the parsed tree [42].

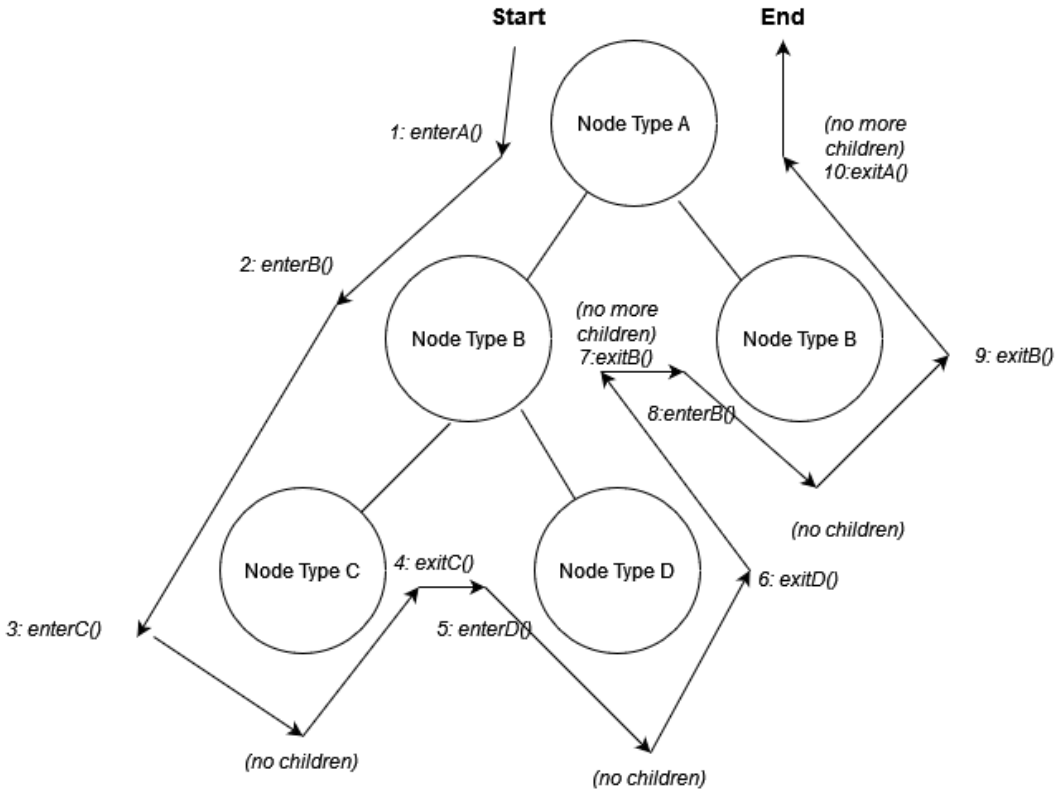

Figure 3.4: Visualization of the processes of a *ParseTreeWalker*.

Figure 3.4 is understood like so. The walk begins at the root, and function calls are triggered whenever the walker enters or exits a node. These function calls are how the walker informs its observers.

1. The walk begins. The walker encounters a node of type A and triggers the *enterA()* function.
2. The walker looks for children of this node and finds a node of type B, triggering the *enterB()* function.
3. The walker again looks for children of this node (B) and finds a node of type C, triggering the *enterC()* function.
4. The walker again looks for children of this node (C) and finds none. It is done with the node and triggers the *exitC()* function as it leaves it.
5. The walker looks for more children of the parent B node and finds a node of type D, triggering the *enterD()* function.
6. The walker again looks for children of this node (D) and finds none. It is done with the node and triggers the *exitD()* function as it leaves it.
7. The walker looks for more children of the parent B node and finds none. It is done with the node and triggers the *exitB()* function as it leaves it.
8. The walker looks for more children of the parent A node and finds another node of type B, triggering the *enterB()* function.
9. The walker again looks for children of this node (B, the second one) and finds none. It is done with the node and triggers the *exitB()* function as it leaves it.
10. The walker looks for more children of the A node and finds none. It is done with the node and triggers the *exitA()* function as it leaves it. This concludes the walk.

4 Research Project – Implementation

4.1 Implementation Data and ANTLR Grammar

We began by gathering 40 CSV files from the Linnaeus University DiVA portal [19], each containing 250 lines of data. We made a simple search with an empty query and exported the first 10,000 entries using the format *CSV all metadata*. We combined this data into files containing 1 000, 2 000, 4 000 and 8 000 lines, manually fixing anything too malformed to be parsed.

Our grammar is designed to decompose a line in a CSV to the degree that it can be used to tally university contributions. As such, anything after that part (abstract, publication date, etc.) in the files is lumped together. Figure 2.3 illustrates how one line might be decomposed into a tree, with the *restFields* representing the uninteresting data.

We wish to point out two things. Firstly, the DiVA portal is constantly updated. The curated data is available on our Git repository, along with our test code. Secondly, the data is used to have our program achieve something meaningful and increase validity, but our project is about comparing patterns for parsing the data, not about drawing conclusions from the DiVA data itself.

As such, while our process led us to replace some of the DiVA data and our ANTLR grammar is likely not optimal, we argue that this is generalizable. As our experiment is about measuring three different patterns walking the same tree, we disregard the construction process.

4.2 General Pattern Information

Again, looking at Figure 2.3 and bearing in mind that for each line in the CSV (the *row* at the root of the subtree) we only wish to count unique institution names, one might be tempted to utilize regularities that the trees might exhibit, in order to avoid processing irrelevant nodes.

We considered this, but elected not to do it for three reasons:

1. It would have meant more work fixing the CSV files, as malformed data would not contain these regularities.

2. It would involve editing the ANTLR library code to an uncomfortable degree, as its Listener and Visitor by default process all nodes. It may be fixable for Visitor without changing the library, but not Listener.

3. We feel our results are easier to put into perspective if the reader does not have to make too many assumptions about how well we can optimize the patterns.

Therefore, all three patterns will process all nodes, including those consisting of only a comma or parenthesis, even when we know that this is not necessary. All three patterns walk the tree in a depth-first search fashion.

To perform its task, each pattern builds a dictionary where each key is the name of an affiliation and the value is the number of occurrences of the respective affiliation in the data (that is the count of publications per affiliation). All three of them accomplish this by, in their own way, making each row responsible for populating a set with institution names, which the dictionary then parses.

The information we are looking for is held by *affiliationSegment* nodes. Each node has a *getText()* function, provided by ANTLR, that is recursively called on all its children

to produce a string, in this case the name of the institution. Note that calling *getText()* is not what we refer to by processing a node with a pattern; that is, for example, invoking the node's *accept()* function with a Visitor.

4.3 Patternless (Algorithm Sprawl)

In this implementation, the tree itself contains the logic to assemble the dictionary. Each node is given a *getValue()* function whose invocation equates to processing the node. Examples to follow in Listings 1 and 2:

Listing 1: Node specific functions

```
# root of the tree; the function called by the test
def csvFileContextGetValue(self):
  dict = {}
  nameSet = set()
  for child in self.children:
    child.getValue(nameSet)
    for name in nameSet:
      if name in dict:
        dict[name] += 1
      else:
        dict[name] = 1
    nameSet.clear()
  return dict

# row, but also the default implementation seen in most named nodes
def rowContextGetValue(self, set):
  for child in self.children:
    child.getValue(set)

# here, an institution name is added to the set
def affiliationSegmentContextGetValue(self, set):
  set.add(self.getText())
  for child in self.children:
    child.getValue(set)

# default implementation for nodes such as commas
def defaultGetValue(self, set):
  pass
```

These are injected into the ANTLR classes, becoming *getValue()* functions like so:

Listing 2: Patching/adding node specific functions to the parsed tree

```
# patches the ANTLR-generated parser class
def patchParserWithGetValueFunction(CSVParser):
  CSVParser.CsvFileContext.getValue = csvFileContextGetValue
  CSVParser.HdrContext.getValue = hdrContextGetValue
  CSVParser.RowContext.getValue = rowContextGetValue
  # ...

# patches the ANTLR library itself
def patchAntlrBaseClass(antlr):
  antlr.tree.Tree.TerminalNode.getValue = defaultGetValue
```

As seen in the example code, the root iterates over the *row* nodes and has each of them build a set of strings naming institutions involved in the publication, then uses this set to update the dictionary. This is achieved by passing the set down the tree. Nodes with children send the set to them, leaf nodes pass, and *affiliationSegment* nodes add to

the set before sending it to their children. It may be noted that this set is sent to a lot of needless locations, but as we mentioned, we make all patterns process all nodes to make the analysis simpler.

We gather the relevant functions in our own class *patternless.py* and inject them into the ANTLR-generated parser class, as well as a default function into the parent class (*TerminalNode*) of the tree nodes in the ANTLR library. The former is to avoid editing the parser itself, which while is not recommended but has worked, would have to be repeated each time the grammar was recompiled. The latter is to allow it to process nodes such as commas without crashing, as these are not explicitly named in our grammar and thus do not get their own node type.

Since Python allows this function injection, it could be argued that this is not true algorithm sprawl from the perspective of maintainability, but as far as runtime effects go, it is the same: The objects contain and perform the algorithms themselves, rather than delegating that responsibility to a Visitor or Observer.

4.4 Listener/Observer

The ANTLR Listener allows us to observe a *ParseTreeWalker* and catch relevant information from it. Note that the tree itself is not what we are observing. The ANTLR AST is built once and not changed; we are subscribing to updates from the walker as it explores the tree. This means our measurements also include the activities of the *ParseTreeWalker* (the observable).

The Listener has two triggers for each node: the walker entering the node when trying to reach greater depth, and the walker exiting the node after returning from greater depth, on its way back to the node's parent. By default, each is a *pass*. We override the behavior of entering *affiliationSegment* nodes to add their names to the set, and the behavior of exiting *nameField* nodes to make the dictionary parse the set.

We described the mechanics of the *ParseTreeWalker* in detail in Figure 3.4.

4.5 Visitor

The default behavior of the ANTLR Visitor is that unnamed nodes - such as commas - simply return *None*, and named nodes - such as *row* nodes - iterate over their children and send the Visitor to each of them. We overrode this behavior for *nameField* and *affiliationSegment* nodes. Our Visitor contains a dictionary and a set. *nameField* nodes, after visiting all their children (and grandchildren, etc) will populate the dictionary using the set. *affiliationSegment* nodes will contribute their text to the set.

4.6 Separating the Patterns

In summary, this is the difference between the patterns:

- In Patternless, we ask the tree itself to assemble the dictionary, and each node knows how to contribute.
- In Listener/Observer, we observe a ParseTreeWalker object as it explores the tree. Every new node it enters or leaves, it notifies the Observer, and we define behavior for interesting nodes to help us assemble the dictionary.
- In Visitor, we let a Visitor object walk the tree, and each node "greets" the Visitor, causing any relevant functions to be invoked.

- In all cases, all nodes are processed. While this simplifies comparisons, it also means all three patterns are playing by Listener's rules.

4.7 Running the Experiment

As mentioned, we measure energy consumption, memory usage and execution time, in that order. We do this for Listener, Patternless and Visitor, in automated sequence. We use the Ubuntu terminal to run the Python scripts, after first making a best effort to close any unnecessary background processes.

As executing the entire set of measurements takes over three hours (memory measurements consuming the bulk of this), we also disable any power save functions on the laptop, so that no pattern looks better due to a dimmed/suspended screen reducing energy consumption.

5 Results

This section will focus on a detailed overview of the experimental results. The represented data is organized into tables to facilitate subsequent analysis. The raw unstructured data is available on our Git repository [44]. The analysis itself will happen in subsequent chapters; the results are only presented and described here.

Table 5.2: RAPL, memory and runtime measurements of 3 patterns

Data sizes	Runtime, ms	Memory, MiB	CPU, mJ	DRAM, mJ	Energy measurement duration, ms
Patternless					
1000 rows	Mean: 366.3 SD: 2.1 Median: 365.9	466.6	Mean: 3390.6 SD: 16.5 Median: 3391.7	Mean: 363.3 SD: 4.5 Median: 362.8	Mean: 363.2 SD: 2.2 Median: 363
2000 rows	Mean: 468.4 SD: 1.2 Median: 468.1	1121.50	Mean: 4352.8 SD: 29.3 Median: 4351.9	Mean: 480.6 SD: 3 Median: 480.8	Mean: 470.5 SD: 2.9 Median: 470
4000 rows	Mean: 707.8 SD: 1.7 Median: 707.6	1912.1	Mean: 6448.6 SD: 36.1 Median: 6439.9	Mean: 736.8 SD: 4.1 Median: 737.2	Mean: 702.5 SD: 3.1 Median: 702
8000 rows	Mean: 1126.9 SD: 1.2 Median: 1126.8	3478.1	Mean: 10286.8 SD: 58.3 Median: 10279	Mean: 1199.6 SD: 7.8 Median: 1198.1	Mean: 1121.1 SD: 2.3 Median: 1121
Listener					
1000 rows	Mean: 783.8 SD: 2.1 Median: 783.6	466.7	Mean: 7147.4 SD: 60.7 Median: 7148.1	Mean: 657.3 SD: 6 Median: 657.5	Mean: 780.4 SD: 2.5 Median: 780.8
2000 rows	Mean: 1120.8 SD: 2.7 Median: 1120.6	1121.5	Mean: 10273.8 SD: 41.3 Median: 10271.5	Mean: 945.6 SD: 4.1 Median: 945	Mean: 1121 SD: 2.4 Median: 1121
4000 rows	Mean: 1861.6 SD: 3.8 Median: 1861.5	1915.5	Mean: 17046.3 SD: 104.3 Median: 17010.3	Mean: 1561.7 SD: 6 Median: 1560.6	Mean: 1855.3 SD: 4 Median: 1856.6
8000 rows	Mean: 3195.3 SD: 7.6 Median: 3194.3	3455	Mean: 29099.4 SD: 229.3 Median: 29183.5	Mean: 2695.1 SD: 14.9 Median: 2694.8	Mean: 3207.1 SD: 9 Median: 3206.8
Visitor					
1000 rows	Mean: 1004.2 SD: 2.3 Median: 1004.2	466.6	Mean: 9114.6 SD: 34.8 Median: 9110.2	Mean: 811.3 SD: 3.6 Median: 810.5	Mean: 1003.8 SD: 1.4 Median: 1003.6
2000 rows	Mean: 1498.6 SD: 2.6 Median: 1498.1	1120.8	Mean: 13788.7 SD: 46 Median: 13784.6	Mean: 1216.1 SD: 4.5 Median: 1215	Mean: 1518.7 SD: 3 Median: 1518.3
4000 rows	Mean: 2585.9 SD: 4.5 Median: 2585.8	1932	Mean: 23713.8 SD: 104.3 Median: 23709.3	Mean: 2053.6 SD: 10 Median: 2054.7	Mean: 2579.1 SD: 9.7 Median: 2580.2
8000 rows	Mean: 4566.3 SD: 22 Median: 4556.8	3492.5	Mean: 41951.3 SD: 495.5 Median: 41930.8	Mean: 3675 SD: 88.8 Median: 3663.3	Mean: 4634.9 SD: 28 Median: 4631.6

Table 5.2 consolidates all the measurements that were obtained as the result of our conducted experiments. The data is organized according to the patterns and subsequently

according to the sizes of the sample testing data (the number of publication records in each CSV file). Each table row corresponds to the results of 25 runs of AST traversal with the respective data size (e.g. 1000 CSV rows). The mean is the arithmetic mean.

Column “Runtime, ms” denotes the runtime of the AST traversal in milliseconds. Columns “CPU, mJ”, “DRAM, mJ” and “Energy measurement duration, ms” represent the data that was measured with RAPL. pyRAPL [20] measured CPU and DRAM energy consumption in microJoules which were transformed into milliJoules for easier presentation. The mean and median indicators in each cell of these columns are intended to bring the data of 25 runs to the central tendency. The value of the standard deviation “SD” gives an idea about the variance of the acquired results.

Column “Memory, MiB” is obtained with the help of memory-profiler tool [25]. This data does not have a mean, standard deviation and median calculation, because the same consistent values were received every 25 runs.

Figure 5.5 is intended to visualize the relationship between the data set sizes and combined mean energy (column “CPU, mJ” + column “DRAM, mJ” from Table 5.2) that patterns consume while traversing ASTs of different sizes. Plotting Patternless, Listener and Visitor results in one figure can be used to observe the patterns’ energy consumption in comparison to each other and depict the general trend of energy usage for each of them.

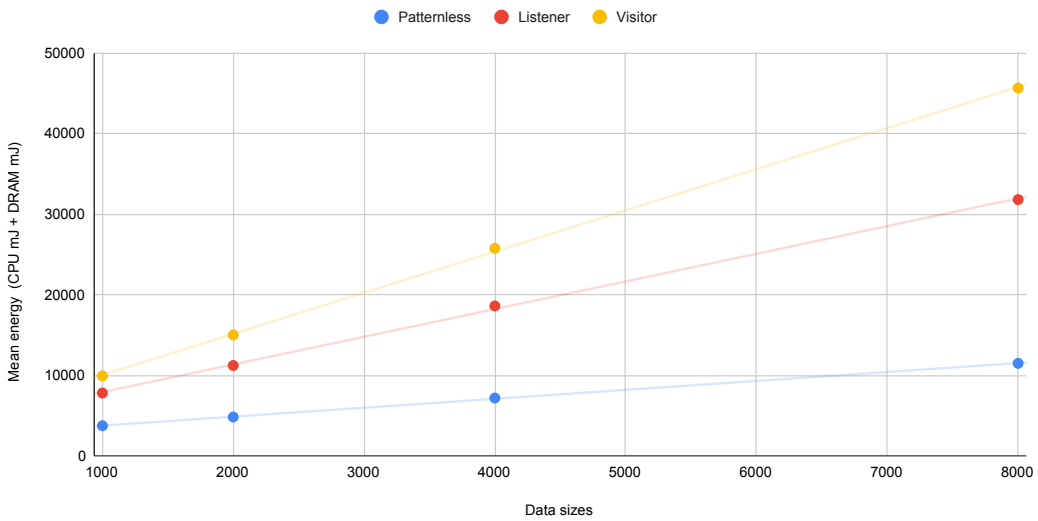

Figure 5.5: Combined mean CPU + DRAM, mJ of 3 patterns and 4 data sizes

In the same manner, Figure 5.6 demonstrates the relationship between the data set sizes and the mean runtime of each pattern traversing the AST structure of the respective size. As was mentioned before, the runtime measurements account for the complete execution of the test program.

While we demonstrate the runtime in Figure 5.6, we chose to use the time (ms) metrics from the column “Energy measurement duration, ms” of Table 5.2 for the subsequent calculations because these mean values represent the duration of intervals that were monitored by pyRAPL.

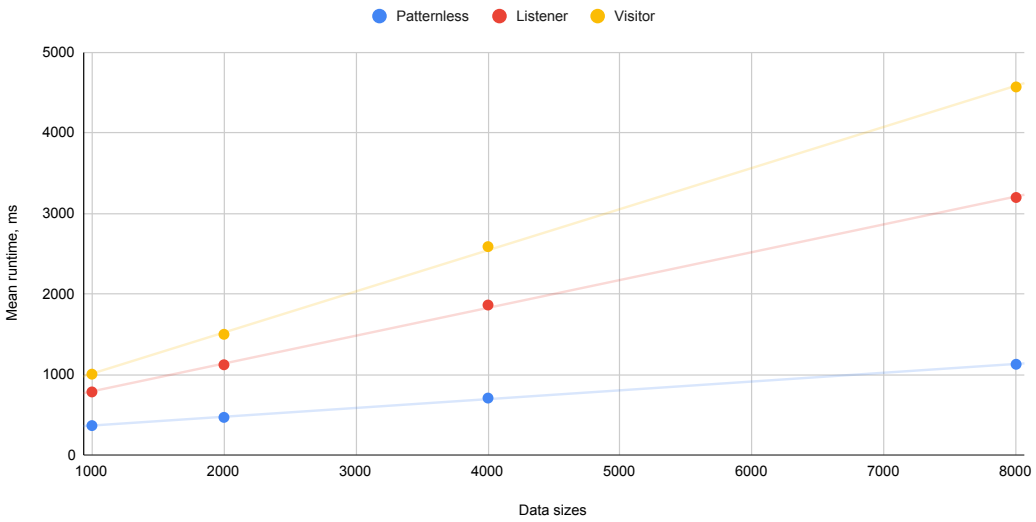

Figure 5.6: Mean runtime, ms of 3 patterns and 4 data sizes

Table 5.3: Performance and energy change

Data sizes	**Patterns**	**Mean energy measurement duration, ms**	**Combined mean energy, mJ**	**Time diff., %**	**Energy diff., %**
Patternless vs Listener					
1000 rows	patternless listener	363.2 780.4	3753.9 7804.7	114.9	107.9
2000 rows	patternless listener	470.5 1121	4833.4 11219.4	138.3	132.1
4000 rows	patternless listener	702.5 1855.3	7185.4 18608	164.1	159
8000 rows	patternless listener	1121.1 3207.1	11486.4 31794.5	186.1	176.8
Patternless vs Visitor					
1000 rows	patternless visitor	363.2 1003.8	3753.9 9925.9	176.4	164.4
2000 rows	patternless visitor	470.5 1518.7	4833.4 15004.8	222.8	210.4
4000 rows	patternless visitor	702.5 2579.1	7185.4 25767.4	267.1	258.6
8000 rows	patternless visitor	1121.1 4634.9	11486.4 45626.3	313.4	297.2
Listener vs Visitor					
1000 rows	listener visitor	780.4 1003.8	7804.7 9925.9	28.6	27.2
2000 rows	listener visitor	1121 1518.7	11219.4 15004.8	35.5	33.7
4000 rows	listener visitor	1855.3 2579.1	18608 25767.4	39	38.5
8000 rows	listener visitor	3207.1 4634.9	31794.5 45626.3	44.5	43.5

In order to provide more comparative data for the subsequent analysis and facilitate the relative assessment of the patterns' energy consumption, Table 5.3 depicts the percentage change or relative percentage difference between the time and energy metrics of the pattern pairs (Patternless - Listener, Patternless-Visitor, and Listener-Visitor). The percentage of change is calculated according to the following formula: $Change(\%) = \frac{Metric(Pattern2) - Metric(Pattern1)}{Metric(Pattern1)} \times 100\%$. The positive percentages denote an increase in the respective metrics of time and energy consumption.

Finally, Table 5.4 is purposed to give an idea about the actual rate of energy consumption across the three patterns while traversing different sizes of ASTs. The rate as the ratio between the energy and measurement duration is represented in watts (W) as per the formula: $Power(watts) = \frac{Energy(mJ)}{Time(ms)}$. Both metrics of energy and measurement duration are the combined mean values that are represented in Table 5.2.

For comparison, the base power consumption of the idle system used to run the tests is 1.09W, with a standard deviation of 0.04, based on RAPL command-line interface tool readings.

Table 5.4: Energy measurements and corresponding power values of 3 patterns

Data sizes	**Combined mean energy, mJ**	**Mean energy measurement duration, ms**	**Power, W**
Patternless			
1000 rows	3753.9	363.2	10.3
2000 rows	4833.4	470.5	10.3
4000 rows	7185.4	702.5	10.2
8000 rows	11486.4	1121.1	10.2
Listener			
1000 rows	7804.7	780.4	10
2000 rows	11219.4	1121	10
4000 rows	18608	1855.3	10
8000 rows	31794.5	3207.1	9.9
Visitor			
1000 rows	9925.9	1003.8	9.9
2000 rows	15004.8	1518.7	9.9
4000 rows	25767.4	2579.1	10
8000 rows	45626.3	4634.9	9.8

6 Analysis

6.1 Memory Observations

Any memory difference is seemingly overshadowed by the memory requirements of the tree itself. Even in the most dramatic case, Patternless vs. Visitor for size 4 000, the 20 MiB difference represents only a 1% change. As such, we cannot claim any pattern is superior in this regard; there is not even one that consistently wins or consistently loses. As the memory usage is nearly identical for the two smallest data set sizes, we also cannot suggest that each pattern has a certain "base" memory footprint, the differences between which would be more pronounced without having the tree involved. In short, changing the size of the data does not seem to impact which pattern does the best job, because memory usage remains virtually equal.

6.2 Runtime and Energy Consumption Observations

A similar trend of growth can be observed in Figures 5.5 and 5.6 regarding the values of the mean combined energy consumption and the mean values of the runtime in relation to the data set sizes across the patterns. Therefore we can explore the relationship between these two metrics by using linear correlation analysis [45].

More specifically, we used the Pearson product-moment correlation coefficient as computed within Google Sheets. Like Bunse and Stiemer [9], we have a strong correlation between runtime and energy consumption. For each of the patterns, the correlation coefficient is greater than 0.9999: (1) 0.999957 for Patternless, (2) 0.999992 for Listener, and (3) 0.999993 for Visitor. We explain this by the system simply mustering all available resources to complete the task at hand. It runs at full speed for as long as it has to; the best pattern is the one that requires it to run the shortest duration. In other words, the fastest pattern is also the most energy-efficient pattern.

The power values in Table 5.4 support the notion about the execution time being the most important factor that influences energy efficiency and overall energy consumption according to our findings because the rate of the energy consumption stays consistent and is almost equal ($\approx$10 W) throughout all experiments.

While the scatterplots in Figure 5.6 look fairly linear, Table 5.3 suggests that Patternless has a greater relative (as well as absolute) advantage as the size of the data set increases. Bearing in mind that our data is heterogeneous, and that a row in a CSV is thus not a constant amount of work to process, we ran an informal post-experiment runtime test with homogeneous data; all files had the same row duplicated over and over. Under these conditions, all three patterns exhibited heavily linear behavior, to the point that we could almost say that runtime (ms) is simply the number of CSV lines multiplied by some pattern-specific constant: Approximately 0.102 for Patternless, 0.340 for Listener and 0.555 for Visitor.

We will not draw too many conclusions from an informal test with an arbitrary CSV line, especially about the differences between the patterns. There is also the possibility that Python or ANTLR makes optimizations about this repeated data. We still feel it adds some context to the analysis.

Instead, what Table 5.3 does show is that the relative advantages of Listener over Visitor, and of Patternless over both of them, grow as the size of the data set does. As this is based on heterogeneous data, we argue it is the more realistic conclusion to draw.

6.3 Further Statistical Observations

The mean and median that are calculated based on the measurements consolidated in Table 5.2 have almost identical values signifying that the experimental findings have an approximately symmetrical normal distribution. The relatively small values of the standard deviation in relation to the mean identify the high precision and consistency of the gathered experimental data.

While the differences in the energy consumption across the pattern alternatives are evident for the same respective data sizes, as presented in Tables 5.2, 5.3 and 5.4 as well as Figure 5.5, we conduct a basic form of statistical hypothesis testing in order to find out if the differences between these observations are statistically significant.

We have conducted two-tail, two-sample unequal variance distribution t-test [46] within Google Sheets for the metrics of energy CPU and DRAM consumption as those represent the most interest for the purpose of our research. The results are presented in Table 6.5.

Table 6.5: T-test for the measurements of energy consumption (CPU and DRAM, mJ)

Data sizes	**CPU energy test p-value**	**DRAM energy test p-value**
Patternless vs Listener		
1000 rows	7.01E-50	2.32E-67
2000 rows	4.91E-86	4.21E-82
4000 rows	2.82E-59	2.52E-84
8000 rows	1.46E-52	4.69E-69
Patternless vs Visitor		
1000 rows	9.94E-74	2.43E-82
2000 rows	1.73E-88	1.32E-85
4000 rows	1.45E-65	1.63E-66
8000 rows	4.66E-46	7.77E-37
Listener vs Visitor		
1000 rows	1.65E-53	4.65E-51
2000 rows	2.49E-78	2.22E-73
4000 rows	2.55E-74	1.65E-61
8000 rows	8.61E-46	8.82E-28

All of the respective p-values are much smaller than the typical significance levels (alpha levels) of 0.05 or even 0.01, indicating statistically significant differences between the sample means, that is, our observations for the three patterns [46].

7 Discussion

7.1 Pattern Comparison

Having established that the patterns do not differ significantly in the memory aspect, and that lower runtime strongly correlates with lower energy consumption, we will simply examine the patterns from the perspective of efficiency, which then means both speed and energy consumption.

With effectively only one metric for comparison, and the outputs of this metric remaining consistent with data set size, Patternless is a clear winner. We have made every effort to make it a fair competition: As we keep repeating, every pattern processes every node, the tree is the same, and even the method for assembling the dictionary is the same (make the subtree fill out a set). Still, "our" pattern has proven the most efficient.

Our best explanation is lack of indirection. In every pattern, we have the tree itself. In Patternless, we do not have anything else. The tree simply has functions added to it, and accomplishes with these functions what Visitor accomplishes by having the tree constantly talk to a Visitor object, and what Listener achieves by having the tree constantly interact with a *ParseTreeWalker*, which in turn interacts with its Observer.

Also, because Python allows function injection, our Patternless solution is just as maintainable as the other two.

Yet, based on this logic, should we not expect Visitor to outperform Listener? It looks like it should cause less object message traffic.

We can only suspect there are optimizations at work, as Listener is intended to walk the whole tree; we could not make it do otherwise without changing the ANTLR library itself. The power we have is deciding what nodes we want updates about, not whether it walks them.

Visitor, on the other hand, exists for more controlled traversal. By default it visits every node. ANTLR allows us to change this behavior, but we have not done so to allow for simpler comparison. It still visits every node, but in some, it also gathers the data we need. Note that this is Visitor playing by Listener's rules.

Our best theory is that Listener causes less message traffic because the *ParseTreeWalker* does not need to consult any other object about how to continue traversing. It reaches a node, notifies the Observer (who usually does not care and simply has a *pass* reaction), then continues. The Visitor, on the other hand, is greeted by a node which tells the Visitor what to do next (visit any children of the node). Listener and Visitor both talk to the tree, but the tree also talks back to Visitor.

7.2 Compared to Related Work

We have a much more pronounced advantage for Patternless as compared to Visitor than Bree and Cinnéide observed [11]. This could be both due to Java and Python differences, as well as the test program used.

Bunse and Stiemer [13] found practically no difference between Observer and no pattern, but it is unclear how they tested it. The ANTLR Listener, while qualifying as an Observer, is still observing something inspecting static data, a process for which there are a few alternatives. If on the other hand our AST could be updated and we could subscribe to updates to single nodes, Observer would probably be the best pattern for keeping the dictionary accurate. We suspect that Bunse and Stiemer's Observer accomplished a task like that, making our results difficult to compare.

Sahin et al. [14] had a slight (≈7%) improvement from using Visitor, while Observer was worse by 62%, each compared to no pattern. Again, the exact implementations are difficult to infer. The authors did document an increase in object creation and messages from using these patterns, which may support our own theories about the costs of Visitor and Listener in our project.

Unfortunately, design pattern energy efficiency is not a richly explored subject, and so it is rather hopeless to say what results to expect. This may very well be the first paper of its kind in the context of Python. We remind the reader that Python is a notoriously inefficient language when it comes to energy consumption and execution time [9], which could very well mean our results must be placed in the context of a wildly different landscape than these Java/C++ studies.

7.3 Generalizability

To begin with, we will use the context of ANTLR ASTs. The results for Listener and Visitor were to be expected (in the sense that Listener is better when the task is to walk the entire tree), but the superiority of Patternless means that this can certainly be considered for ANTLR AST parsing in general. We hope the reader will agree that once function injection is understood, it is not much more difficult to implement than a Listener or Visitor.

We deliberately chose AST parsing so that we could be confident that we could induce heavy usage of our patterns, and thus be able to say that different results are indeed caused by different patterns. As such, our findings should transfer well to other use cases (outside of ANTLR and ASTs), though bearing in mind what was pointed out about the ANTLR Listener as compared to traditional Observer usage in the previous subsection.

As Python is an inefficient language [9], in particular compared to the alternatives with which it competes for worldwide popularity [4], we dare not make any claims that our results are applicable outside Python. Readers with insight into what makes programming languages different from, and similar to, each other may be able to reason about this well enough to use our findings in other contexts.

8 Conclusions and Future Work

In this thesis project, we decided to investigate the problem of software energy consumption, as this topic seems to be overlooked by the majority of the studies that are focused on the implications of the ICT sector increasing energy consumption. More specifically, we chose to concentrate on software design patterns as those are the high-level design abstractions that can shape the structure and behavior of the software, therefore we were interested in the energy cost that comes with the indirection of pattern application. To help inform tradeoffs, we also measured runtime and memory usage. Our patterns were Visitor, Observer and an unpatterned solution. Based on the received measurements and their analysis, we are able to answer the research questions in Section 1.3.

8.1 Research Questions Consideration

How do Observer, Visitor and Patternless compare for energy consumption? We found that Patternless consistently outperformed the other two, spending between half and a third as much energy as Observer and in the ballpark of a third and a fourth as much energy as Visitor. Observer consistently outperformed Visitor, spending roughly 25-30% less energy.

How do Observer, Visitor and Patternless compare for execution time? We found an extremely strong correlation between execution time and energy consumption. Consequently, our answers are the same as those to the previous question: Patternless is 1-2 times faster than Observer and 2-3 times faster than Visitor, while Observer is 25-30% faster than Visitor.

How do Observer, Visitor and Patternless compare for memory usage? This turned out to be a highly irrelevant consideration. There were extremely small differences in memory usage, and no pattern was consistently the best or worst. Their memory footprints are the same for all intents and purposes.

Do different data set sizes favor different patterns? No. Larger data set sizes only make the established differences even more pronounced. Patternless is always the best and Visitor is always the worst.

All of these results provide a rather consistent picture of 3 patterns ranking with respect to different metrics, while no surprising deviations (e.g., Visitor suddenly becoming faster or consuming less energy than Listener or Patternless) were observed.

8.2 Future Work

This experiment could be repeated with the reflective dispatch approach from Bree and Cinnéide's experiment included [11]. It could also be redone with a few sets of homogeneous data, as well as with optimized Patternless and Visitor that do not process unnecessary nodes. We recommend repeat experimenters to use far fewer iterations for memory measurements, as they are slow and do not seem to provide any benefit, given the complete lack of variance in the results.

We implemented our testing prototypes using the default Python interpreter CPython. The other interpreter options can be configured to process the Python script. The choice of interpreter highly impacts the execution of the Python code therefore conducting similar

experiments using different interpreter configurations could be a potentially interesting addition to the research of the software design patterns energy consumption in the Python context.

Zooming out, design pattern efficiency in Python remains a heavily unexplored field, although the extreme correlation between runtime and energy consumption that we observed could very well mean that any knowledge that exists about design pattern performance can be drawn upon to make assumptions about efficiency.

At an even higher level, design pattern efficiency in general is mostly uncharted territory. It has received some attention for Java and C++ as far as desktop environments go. There is no shortage of languages for which this knowledge gap could be addressed, and should the experimenter favor languages that already have some research in this field, the existing conclusions could certainly stand to be verified independently.

References

[1] L. Lannelongue, J. Grealey, and M. Inouye, "Green algorithms: Quantifying the carbon footprint of computation," *Advanced Science*, vol. 8, no. 12, p. 2100707, 2021.

[2] A. Radovanović, R. Koningstein, I. Schneider, B. Chen, A. Duarte, B. Roy, D. Xiao, M. Haridasan, P. Hung, N. Care, S. Talukdar, E. Mullen, K. Smith, M. Cottman, and W. Cirne, "Carbon-aware computing for datacenters," *IEEE Transactions on Power Systems*, vol. 38, no. 2, pp. 1270–1280, 2023.

[3] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994.

[4] TIOBE, "TIOBE index for April 2023," 2023, accessed 16.04.2023. [Online]. Available: https://www.tiobe.com/tiobe-index/

[5] C. Zhang and D. Budgen, "A survey of experienced user perceptions about software design patterns," *Information and Software Technology*, vol. 55, no. 5, pp. 822–835, 2013.

[6] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, "Automatic design pattern detection," in *Proceedings of 11th IEEE International Workshop on Program Comprehension*. IEEE, 2003, pp. 94–103.

[7] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from Java source code," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 2006, pp. 123–134.

[8] A. Shvets, *Dive Into Design Patterns*. Refactoring.Guru, 2018.

[9] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Ranking programming languages by energy efficiency," *Science of Computer Programming*, vol. 205, p. 102609, 2021.

[10] D. Feitosa, R. Alders, A. Ampatzoglou, P. Avgeriou, and E. Y. Nakagawa, "Investigating the effect of design patterns on energy consumption," *Journal of Software: Evolution and Process*, vol. 29, no. 2, p. e1851, 2017.

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

[12] D. Connolly Bree and M. Ó. Cinnéide, "Removing decorator to improve energy efficiency," in *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 902–912.

[13] C. Bunse and S. Stiemer, "On the energy consumption of design patterns," *Softwaretechnik-Trends*, vol. 33, pp. 7–8, 05 2013.

[14] C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh, "Initial explorations on design pattern energy usage," in *Proceedings of the 2012 First International Workshop on Green and Sustainable Software (GREENS)*, 2012, pp. 55–61.

[15] T. Parr, "ANTLR: Another tool for language recognition," 2013, accessed 16.04.2023. [Online]. Available: https://www.antlr.org/

[16] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer Berlin Heidelberg, 2012.

[17] M. V. Zelkowitz and D. Wallace, "Experimental validation in software engineering," *Information and Software Technology*, vol. 39, no. 11, pp. 735–743, 1997.

[18] V. R. Basili, "The experimental paradigm in software engineering," in *Experimental Software Engineering Issues: Critical Assessment and Future Directions*. Springer Berlin Heidelberg, 1993, pp. 1–12.

[19] "Linnaeus University DiVA portal," accessed 16.04.2023. [Online]. Available: https://lnu.diva-portal.org/

[20] C. Belgaid, A. d'Azémar, G. Fieni, and R. Rouvoy, "pyRAPL," 2019, accessed 16.04.2023. [Online]. Available: https://pypi.org/project/pyRAPL/

[21] Intel, "Intel® 64 and IA-32 architectures software developer's manual, volume 3B: System programming guide, part 2," Tech. Rep., Mar. 2023.

[22] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using RAPL," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 3, pp. 13–17, Jan. 2012.

[23] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan, "Power-management architecture of the Intel microarchitecture code-named Sandy Bridge," *IEEE Micro - MICRO*, vol. 32, pp. 20–27, Mar. 2012.

[24] S. Desrochers, C. Paradis, and V. M. Weaver, "A validation of dram rapl power measurements," ser. MEMSYS '16. Association for Computing Machinery, 2016, pp. 455–470.

[25] "memory-profiler 0.61.0," 2022, accessed 16.04.2023. [Online]. Available: https://pypi.org/project/memory-profiler/

[26] S. Flucker and R. Tozer, "Data centre energy efficiency analysis to minimize total cost of ownership," *Building Services Engineering Research and Technology*, vol. 34, no. 1, pp. 103–117, 2013.

[27] C. Freitag, M. Berners-Lee, K. Widdicks, B. Knowles, G. S. Blair, and A. Friday, "The real climate and transformative impact of ict: A critique of estimates, trends, and regulations," *Patterns*, vol. 2, no. 9, p. 100340, 2021.

[28] V. Krey, O. Masera, G. Blanford, T. Bruckner, R. Cooke, K. Fisher-Vanden, H. Haberl, E. Hertwich, E. Kriegler, D. Mueller, S. Paltsev, L. Price, S. Schloemer, D. Uerge-Vorsatz, D. Van Vuuren, T. Zwickel, K. Blok, S. De La Rue Du Can, G. Janssens-Maenhout, D. Van Der Mensbrugghe, A. Radebach, and J. Steckel, "Annex II: Metrics & methodology." Cambridge and New York (UK and USA): Cambridge University Press, 2014, pp. 1281–1328.

[29] S. Georgiou, S. Rizou, and D. Spinellis, "Software development lifecycle for energy efficiency: Techniques and tools," *ACM Computing Surveys*, vol. 52, no. 4, Aug. 2019.

[30] G. Papadimitriou, M. Kaliorakis, A. Chatzidimitriou, D. Gizopoulos, P. Lawthers, and S. Das, "Harnessing voltage margins for energy efficiency in multicore cpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. Association for Computing Machinery, 2017, pp. 503–516.

[31] G. Pinto and F. Castor, "Energy efficiency: A new concern for application software developers," *Communications of the ACM*, vol. 60, no. 12, pp. 68–75, Nov. 2017.

[32] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about software energy consumption?" *IEEE Software*, vol. 33, no. 3, pp. 83–89, 2016.

[33] P. Bozzelli, Q. Gu, and P. Lago, "A systematic literature review of green software metrics," Vrije Universiteit Amsterdam, Tech. Rep., 2014.

[34] M. Couto, R. Pereira, F. Ribeiro, R. Rua, and J. a. Saraiva, "Towards a green ranking for programming languages," in *Proceedings of the 21st Brazilian Symposium on Programming Languages*, ser. SBLP '17. Association for Computing Machinery, 2017.

[35] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. a. P. Fernandes, and J. a. Saraiva, "Energy efficiency across programming languages: How do energy, time, and memory relate?" in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017. Association for Computing Machinery, 2017, pp. 256–267.

[36] A. C. Moises, A. Malucelli, and S. Reinehr, "Practices of energy consumption for sustainable software engineering," in *2018 Ninth International Green and Sustainable Computing Conference (IGSC)*, 2018, pp. 1–6.

[37] D. Feitosa, L. Cruz, R. Abreu, J. P. Fernandes, M. Couto, and J. Saraiva, *Patterns and Energy Consumption: Design, Implementation, Studies, and Stories*. Springer International Publishing, 2021, pp. 89–121.

[38] A. Litke, K. Zotos, A. Chatzigeorgiou, and G. Stephanides, "Energy consumption analysis of design patterns," *World Academy of Science, Engineering and Technology, International Journal of Electrical, Computer, Energetic, Electronic and Communication Engineering*, vol. 1, pp. 1655–1659, 2007.

[39] A. Noureddine and A. Rajan, "Optimising energy consumption of design patterns," ser. ICSE '15. IEEE Press, 2015, pp. 623–626.

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

[41] F. Ortin, J. Quiroga, O. Rodriguez-Prieto, and M. Garcia, "An empirical evaluation of Lex/Yacc and ANTLR parser generation tools," *PLOS ONE*, vol. 17, no. 3, pp. 1–16, 03 2022.

[42] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[43] ANTLR Documentation, “Class ParseTreeWalker,” accessed 16.04.2023. [Online]. Available: https://www.antlr.org/api/Java/org/antlr/v4/runtime/tree/ParseTreeWalker.html

[44] “Project github repository,” accessed 22.05.2023. [Online]. Available: https://github.com/alberthatesnewsletters/bachelor-project

[45] J. Frost, *Regression Analysis: An Intuitive Guide for Using and Interpreting Linear Models*. Statistics By Jim Publishing, 2020.

[46] ——, *Hypothesis Testing: An Intuitive Guide for Making Data Driven Decisions*. Statistics by Jim Publishing, 2020.