Investigating the effect of design patterns on energy consumption

Daniel Feitosa*,†,1, Rutger Alders[1], Apostolos Ampatzoglou[1], Paris Avgeriou[1], Elisa Yumi Nakagawa[2]

[1]*Department of Mathematics and Computer Science, University of Groningen, the Netherlands*
[2]*Department of Computer Systems, University of São Paulo, Brazil*

ABSTRACT

GoF patterns are well-known best practices for the design of object-oriented systems. In this paper we aim at empirically assessing their relationship to energy consumption, i.e., a performance indicator that has recently attracted the attention of both researchers and practitioners. To achieve this goal, we investigate pattern-participating methods (i.e., those that play a role within the pattern) and compare their energy consumption to the consumption of functionally equivalent alternative (non-pattern) solutions. We obtained the alternative solution by refactoring the pattern instances using well-known transformations (e.g., replace polymorphism with conditional statements). The comparison is performed on 169 methods of two GoF patterns (namely State/Strategy and Template Method), retrieved from two well-known open source projects. The results suggest that for the majority of cases the alternative design excels in terms of energy consumption. However, in some cases (e.g., when the method is large in size or invokes many methods) the pattern solution presents similar or lower energy consumption. The outcome of our study can be useful to both researchers and practitioners, since we: (a) provide evidence on a possible negative effect of GoF patterns, and (b) can provide guidance on which cases the use of the pattern is not hurting energy consumption.

KEY WORDS: energy efficiency; design patterns; GoF patterns; template method pattern; state pattern; strategy pattern

1. INTRODUCTION

There has been an increase of energy demand within the ICT domain [1]. This is a multi-faceted problem, as one can consider the effects of networks, hardware, drivers, operating systems, and applications on energy consumption. In this paper we focus on applications and, particularly, how they can be optimized in terms of energy consumption. Software optimizations in this context have been discussed at three levels of granularity:

- ***at architectural level,*** e.g., research that deals with energy efficient architectures for networked systems (e.g., data centers, cloud computing, etc.) [1]–[3].
- ***at design level,*** e.g., identification of differences in energy efficiency when applying design patterns (cf. Section 2).
- ***at source code level,*** discussions on topics such as multi-threading [4], [5], refactoring [6]–[9] and related algorithms [10]–[13].

The scope of our work lies at the design level, as we *look into the effect of GoF (Gang of Four) design patterns and their alternative solutions on software energy consumption*. GoF design patterns are recurring solutions to common problems in object-oriented software design [14]. GoF design patterns can be applied in almost any type of software, varying from small devices to large data-centers. In Java applications it has been reported that up to 30% of system classes participate in one or more GoF design pattern occurrences [15], [16], leading to a significant influence on overall energy consumption. Solutions provided by these patterns exploit object-orientation mechanisms (e.g., polymorphism) to enforce more flexible and maintainable designs.

The effect of applying a pattern is not uniform across all of its instances, and all quality attributes [17]. In particular, several studies [17]–[19] report that the effect of a pattern on a quality attribute depends on certain pattern-related parameters, like the number of classes, number of methods invoked, or number of polymorphic methods. Therefore, it is reasonable to expect that GoF design patterns have a potential impact (positive or negative) on the energy consumption of software-intensive systems,

*Correspondence to: Daniel Feitosa, Department of Mathematics and Computer Science, University of Groningen, the Netherland
†E-mail: d.feitosa@rug.nl

depending on certain pattern-related parameters. In the case where a pattern is not the optimal design solution, alternative (non-pattern) design solutions can be employed. Alternative design solutions have been proposed by several authors, including GoF design pattern advocates [14], [20]–[23]. More details on GoF design pattern alternatives can be found in a recent literature review [24]. We note that knowing the impact of patterns on energy efficiency can be beneficial in both green- and brown-field software development. In Greenfield projects (i.e., fresh development), such a knowledge can support the monitoring of energy efficiency, whereas in Brownfield projects (e.g., refactoring of system to new purpose), it can support the decision making process on what parts of the system to refactor and how.

In this paper we investigate the effect of GoF patterns and their alternatives on energy consumption, as well as the pattern-related parameters that might influence this effect. Specifically, we focus on two GoF design patterns, namely Template Method, and State/Strategy [14]; we note that State and Strategy patterns have a similar structure [25] and, therefore, a similar expected effect on energy consumption. Therefore, the two patterns are discussed as one (for more details, see Section 3.1). The rationale for selecting the specific patterns is twofold:

- ***Usage frequency:*** behavioral patterns are the most commonly used patterns, accounting for about half of the design pattern usages in a system [26]. Additionally, State/Strategy patterns are the most used patterns among all, and Template Method the third. Therefore, the accumulated impact of these patterns on energy consumption is expected to be high;
- ***Main object-orientation mechanism:*** object-orientation has three pillars[1]: encapsulation, inheritance, and polymorphism [27]. Polymorphism is the most commonly explored principle within the GoF patterns (19 out of 23 patterns uses polymorphism). However, it is important to highlight that encapsulation and inheritance, although less explored, are also present in the solution of many patterns. From these mechanisms, polymorphism potentially influences energy consumption the most, as it comprises a complex procedure to map the polymorphic calls to the correct implementation [28]. Both State/Strategy and Template Method use polymorphism as their main mechanism to provide the pattern solution and, therefore, have potentially high impact on the energy consumption. The two studied patterns use polymorphism with different goals: State/Strategy pattern uses it to define the interface to interact with the states/strategy, while Template Method pattern uses it to define the points of specialization to be implemented by the concrete classes. In particular, the State/Strategy pattern encapsulates the different states/strategies, whereas the Template Method pattern exploits inheritance, since concrete classes extend the functionality of the abstract class. For that reason, we point that other pillars are part of our investigation, although polymorphism is the main mechanism.

To investigate the energy consumption, we compare the energy efficiency of pattern solutions with the energy efficiency of their alternative designs (one for each pattern), through a crossover experiment. We note that the alternative designs were developed in a standardized way (see Section 3.2 and 3.4). In the experiment, we focus our investigation on *pattern-related methods*[2] so as to enable a fine-grained analysis of the energy consumption. In addition to exploring the differences between pattern and alternative solutions, we also investigate some pattern-related parameters that can cause the pattern to be either beneficial or harmful with respect to energy consumption. For the experiment, we selected two large well-known open source software (OSS) systems.

The remainder of this paper is organized as follows. In Section 2, an overview of the related work on energy consumption in design patterns, and alternatives to design patterns is provided. Section 3 presents background information necessary for understanding the experiment, i.e., the selected design patterns and their alternative solutions. Section 4 presents the experiment planning, which describes the research questions, hypotheses, the used tool and collected variables. Section 5 overviews the execution of the experiment (i.e., data collection and validation). In Section 6, we elaborate on our analysis and answer the research questions. In Section 7, we discuss the obtained findings, by focusing on the most important observations and presenting implication for researchers and practitioners. The threats to the validity of our study are discussed in Section 8, followed by the conclusion of this paper in Section 9.

2. RELATED WORK

This section presents research efforts that discuss the effects of design patterns on energy consumption. We focus on the consumption of design patterns, the types of patterns being investigated, and the

[1]Some authors advocate a fourth pillar: abstraction. However, this is a higher level concept, which is provided as combination of the other three pillars and, therefore, is not relevant for our argumentation.

[2]Pattern-related methods are methods that play a role within the design pattern.

proposed alternatives for patterns. After discussing the related work, an overview of how our research compares to related work is provided.

In the work of Bunse et al. [29], a case study on the overhead of design patterns compared to "clean software" is presented. In this context, "clean software" is a chunk of design that could be refactored into a pattern solution. The software in this study mainly targets mobile devices. The design patterns discussed are Facade, Abstract Factory, Observer, Decorator, Prototype, and Template Method. This initial investigation shows that each of these design patterns has overhead when compared to their "clean" counterparts. Most of the patterns have a relative small overhead, except for the Decorator pattern, which, based on this study, consumes more than double the amount of energy compared to the "clean" counterpart.

Additionally, Sahin et al. [30] performed a more extensive investigation on the impact of design patterns on energy usage. In particular, this study takes into account the feasibility, impact, consistency, and predictability of the energy consumption of 15 design patterns, from all GoF pattern categories. The creational design patterns discussed are the Abstract Factory, Builder, Factory Method, Prototype, and Singleton. The structural patterns discussed are the Bridge, Composite, Decorator, Flyweight, and Proxy pattern. Finally, the behavioral patterns that were selected are the Command, Mediator Observer, Strategy, and Visitor. Results of the study suggest that the use of design patterns, either increases or decreases the amount of energy used. Additionally, there are no relations of the category of the design pattern and the impact on energy usage. Finally, this study shows that it is not possible to precisely estimate the impact of design patterns on energy consumption when only considering artifacts on design level.

Litke et al. [31] conducted an initial exploration of the energy consumption of design patterns. This paper includes an analysis of five design patterns, for which the energy consumption and performance are described. These design patterns were tested by the use of six example applications written in C++. These applications were first tested as clean, i.e., without the usage of design patterns, and then transformed with the designated design pattern. The design patterns discussed are the Factory Method, Adapter, Observer, Bridge, and Composite. For Factory Method, Adapter, and Observer, differences were found between the original application and the one containing the specified design pattern. The results show that applying Factory Method or Adapter patterns does not necessarily impose a serious threat to the energy consumption. However, a significant overhead was identified by employing the Observer pattern, but additional research is still required to investigate the cases when Observer is indeed a threat to energy consumption. Since the Bridge and Composite pattern had no significant difference in power consumption, the authors suggest further analysis.

In a recent paper, Noureddine and Rajan [32] performed a comparison on the energy consumption overhead caused by 21 design patterns and explored in details the effects of two design patterns (Observer and Decorator pattern). The effects discussed in this paper are the energy consumption of applications using the pattern solution, the non-pattern solution, and an optimized alternative for the design patterns. The optimized solutions for the alternatives are integrated into the applications by making changes to compilers, so that the optimizations are automatically processed when compiling. This study suggested that simple transformations to the Observer and Decorator patterns are able to provide reductions in energy consumption in the range of 4.32% to 25.47%. We clarify that the patterns investigated in our study are included among the 21 patterns initially investigated by Noureddine and Rajan. However, the comparison of these results (from the initial investigation) to ours is limited, since some extra details (e.g., implemented alternatives, source code properties) would be necessary to further elaborate the discussion (see Section 7.1).

To ease the comparison of our work to the aforementioned studies, we summarize the main differences in Table I, according to the following aspects: (a) Design patterns addressed; (b) Number of non-trivial systems used; (c) Number of pattern instances analyzed; (d) Number of pattern-related methods analyzed; (e) Level of energy measurement[3] (process level or method level); (f) Level of investigation[4] (instance level or method level); and (g) Number of investigated parameters that influence energy consumption. Based on Table I, the main contributions of this study compared to the research state-of-the-art are the following:

- **Usage of non-trivial systems—**our investigation is performed considering two non-trivial systems and a considerable amount of pattern instances and pattern-related methods. This setup allows allow us to observe realistic results that are more representative to the population of existing software-intensive systems;

[3]Measurement at process level considers the energy consumed by the operating system process of the running software; measurement at method level considers the energy consumed by a specific method within the software process.

[4]Investigation at instance level considers pattern instances as subjects for analysis while the method level considers the pattern-related methods as subjects.

- **Exploitation of a method-level approach for measuring energy consumption**—in addition to the more traditional approach of process-level measurement. Being able to isolate the energy consumed by specific method calls, we obtain measurements with lower overhead, allowing a more in-depth investigation of both pattern and alternative solutions, in the sense that we focus on pattern-related methods of each pattern instance; and
- **Exploration of parameters of the processed patterns—**in this study, we investigate not only the energy efficiency of State/ Strategy and the Template Method design pattern, comparing them against their respective alternative (non-pattern) design solutions, but also the parameters of their application that render them either beneficial or not. We clarify that related work has indicated parameters as possible causes for greater energy consumption, but without any investigation of these parameters.

Table I. Overview of related work

Reference	Design patterns	Non-trivial systems	# of instances	# of methods	Measurement level	Investigation level	# of parameters
[29]	6[a]	0	6	0	Process	Instance	0
[30]	15[b]	0	15	0	Process	Instance	0
[31]	5[c]	0	5	0	Process	Instance	0
[32]	21[d]	0	N/A*	0	Process	Instance	0
This study	3[e]	2	21	169	Process and Method	Method	3

[a] Facade, Abstract Factory, Template Method, Prototype, Decorator, and Observer.
[b] Abstract Factory, Builder, Factory Method, Prototype, Singleton, Bridge, Composite, Decorator, Flyweight, Proxy, Command, Mediator, Observer, Strategy, and Visitor.
[c] Factory Method, Observer, Adapter, Bridge, and Composite.
[d] Decorator, Observer, Mediator, Strategy, Template Method, Visitor, Abstract Factory, Builder, Factory Method, Prototype, Singleton, Bridge, Flyweight, Proxy, Chain of Responsibilities, Command, Interpreter, Iterator, State, Adapter, and Composite.
[e] State, Strategy, and Template Method.
[*] Not available, the authors only mention "several small examples".

3. DESIGN PATTERNS AND ALTERNATIVES

In this section we present background concepts that facilitate the understanding of our experiment. In particular, we discuss the GoF design patterns that are explored in this study (State, Strategy, and Template Method), elaborating on their design structure and an overview of their uses and consequences. Additionally, we present and discuss their alternative solutions (referred in this paper as State/Strategy Alternative and Template Method Alternative). The identification of design pattern alternatives can be a non-trivial activity, since some GoF design patterns have no reported alternatives in the literature [24]. To consider a design as a design pattern alternative, it should:

- originate from the ***literature***;
- provide exactly the ***same functionality*** as the pattern; and
- have notable ***structural differences*** compared to the pattern.

We used two main sources to find alternatives: the seminal book on design refactoring by Fowler et al. [20] and a systematic literature review conducted by Ampatzoglou et al. [24], in which an overview of GoF design pattern alternatives are presented and discussed. Based on the aforementioned criteria, we selected well-known alternative solutions from the literature, as they are expected to be more recurrent in existing software. Although we acknowledge the existence of design patterns and alternatives that are optimized for energy efficiency (which would obviously lead to better solutions), we have deliberately not included them in our study. The reason for this decision is that we intend to focus on widely-known solutions that have been applied to various software projects, by developers who are not aware of energy optimization mechanism. Investigating such optimized solutions can potentially introduce bias to our results, since neither patterns nor alternatives would be in their standard form.

3.1. State/Strategy

The State pattern allows an object to change its behavior by switching from one state to another [14]. One classic example for the State pattern are traffic lights that turn from green to yellow, yellow to red and red back to green. The collection of all states defines the space in which the context (traffic light)

is able to change its behavior. This behavior is implemented by each of the states separately. The context class has at least one state instance object (i.e., a concrete state) that represents its current state and thus functions as a central interface for clients to communicate with (see model on the left in Figure 1). This context delegates the handling of requests to its current state object. The State pattern is used in scenarios where either the behavior of an object depends on its state and needs to be changed during run-time, or the operations have large, multipart conditional statements that depend on the object's state [14]. Applying the State pattern has a number of consequences: the specific behavior for each state is localized; the state transitions are made explicit; and State objects can be shared when they have no instance variables.

The Strategy pattern allows for the encapsulation of certain families (such as algorithms), allowing them to be interchangeable depending on client requests or specific behaviors of the context [14]. The context class has at least one object of the concrete strategy that provides its (unique) functionalities, which are implemented according to a template defined by the strategy interface (see model on the right in Figure 1). The Strategy pattern can be used in a number of different situations [14], e.g., when a class has different behaviors (depending on a specific situation) or when there are multiple implementation options to be chosen. Consequences of using this pattern include [14]: it becomes an alternative for sub-classing the context directly or using conditional statements, by decoupling the algorithms into their own family; and it may cause memory and computational overheads, because it increases the number of used objects, and concrete strategies may not use all information they receive when called.

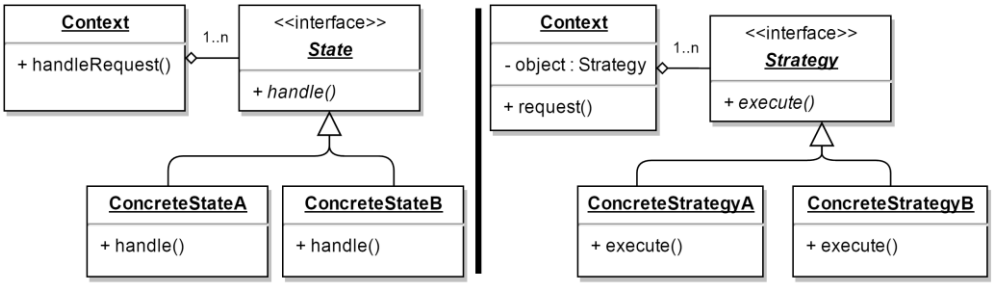

Figure 1. UML model of State (on the left) and Strategy (on the right) patterns

By inspecting the class diagrams of State and Strategy patterns (see Figure 1), we observe that they have an equivalent structure (i.e., skeleton design) [14], [25]. Both patterns have a context that is called by an external client and a family that consists of an interface with concrete classes. Both contexts contain an object that represents at least one or more states/strategies that can be uniformly handled. The main difference is the logic beneath the patterns, i.e., the behavior is fundamentally different. In the case of the State pattern, the current object (state) within the context is updated after the execution of every behavior (the method *handle*, in the diagram). This is not necessary for the Strategy pattern, as strategies may be interchangeable during runtime. Additionally, the change of strategies is more an additional feature than a rule for the Strategy pattern, whereas for State this is the basic concept of the pattern. In this study, we treat both patterns mutually, since the expected changes to measure energy consumption is focused on the design, i.e., structure and the use of their common object-orientation mechanisms. The aforementioned fundamental differences regard the behavior of the pattern instance and, thus, are not expected to be a confounding factor for our study, unless these fundamental differences systematically change design attributes (e.g., method size). Nevertheless, we have not identified such cases in our dataset (see Section 5.2).

3.2. State/Strategy Alternative

In a literature review performed by Ampatzoglou et al. [24] many alternatives for the State/Strategy pattern are presented [21], [22], [33]–[37]. Similarly, Fowler et al. [20] discuss several alternatives for these two patterns. Among these available options, we have chosen to replace the use of polymorphism with the use of conditional statements. In this solution, the entire structure of the State/Strategy pattern is removed and the complete logic is implemented in the context, which now has a local enumerator object that enables the shifting between the different behaviors. Listing I shows an example of alternative implementation for a Strategy pattern instance. While implementing an alternative design, the implementation of each concrete strategy would be replaced with the behavior of the corresponding state and the state update.

Listing I. Example implementation of Strategy alternative

```
public class Strategy {
  public enum Strategies{
         Strategy1,
         Strategy2,
         Strategy3
  };

  private enum currentStrategy;

  public int[] sort(int[] list) {
         switch(currentStrategy) {
                case Strategy1:
                       // Implementation of Strategy 1.
                break;
                case Strategy2:
                       // Implementation of Strategy 2.
                break;
                case Strategy3:
                       // Implementation of Strategy 3.
                break;
                case default:
                       return 0;
                break;
         }
  }
}
```

Despite the simplicity of the recommended changes, creating alternatives requires some effort, as design patterns may be implemented in various different ways. These variations should be reflected into the alternative designs. Based on our experience, one specific type of variation had direct impact in the implementation of the alternative: the structure of the implemented pattern may differ from the originally proposed structure [25]. Specifically, the proposed structure of State/Strategy has a standard Interface-Class (IC) hierarchical structure; however, it may also be implemented with an abstract class between the interface and the class (an intermediate level of inheritance), becoming an Interface-AbstractClass-Class (IAC) hierarchical structure. Such a structure may contain several abstracts classes in the middle. To deal with abstract classes in the alternative, each behavior defined in a concrete class would be combined with the abstract class behavior. If that is not possible, e.g., when a class or abstract class is used from the Java library, an additional object would be created to be able to access its functionalities. We clarify that other, less recurrent, variations are possible, but they are not handled in this study. For example, a State/Strategy may comprise multiple interfaces, which are partial responsibilities, and concrete classes may implement all or some of them.

3.3. Template Method

Similarly to Strategy, the Template Method isolates different algorithms or operations to their own subclass. However, this pattern allows the subclasses to alter certain steps of an algorithm without changing the structure of the algorithm. An abstract class has at least two operations, one primitive, which is used by the concrete subclass to implement the steps of an algorithm, and a template method that contains the default structure (see Figure 2). The Template Method pattern can be used to avoid code duplication, and to control or restrict any extensions of an abstract class, so that an abstract function or hook function can only be called on certain locations.

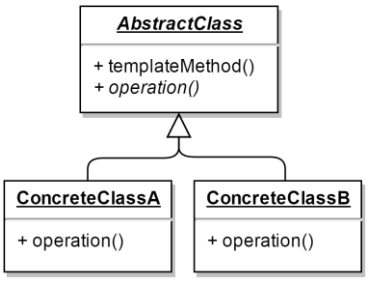

Figure 2. UML model of the Template Method pattern

3.4. Template Method Alternative

Fowler et al. [20] presents several alternatives for the Template Method and Ampatzoglou et al. [24] discuss one alternative [23]. From these options, we chose the starting point from the Form Template Method (FTM) refactoring, presented by Fowler et al. [20]. Generally, FTM transforms a non-pattern code into a Template Method (see Figure 3). In contrast to State/Strategy alternative (Section 3.2), in which we completely eliminated polymorphism, the alternative for Template Method does use polymorphism, but in a different fashion. Therefore, this study design cannot be considered appropriate for comparing the effect of using polymorphism on energy efficiency.

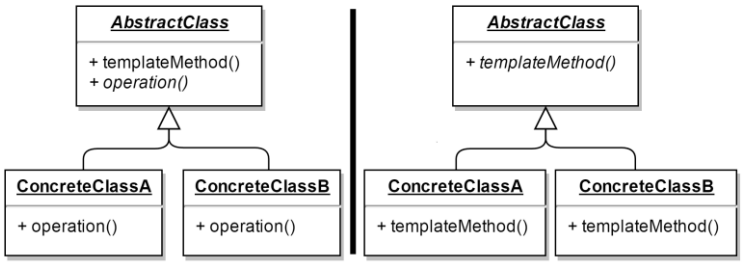

Figure 3. Comparison of the Template Method pattern (on the left) against its alternative (on the right)

By using this alternative, both primitive operations and specific behavioral operation now reside in each concrete class. However, the Template Method also leaves room for variants in its implementation. In such cases, the adjustments that would be applied in the alternative to handle these variations are described below. Similarly to State/Strategy, the Template Method allows all or none of these adjustments to be included.

- **Depth of Inheritance Tree:** Even though the Template Method uses only one abstract class, it is possible that the methods are already defined in an interface. This makes it harder to remove the primitive methods when creating the alternative implementation. In these cases, the primitive method is not removed, but it is moved to the concrete class. This allows us to both keep the IAC structure and to implement the alternative.
- **Private methods:** It is possible for a template method to call private methods within the abstract class. If this is the only case, the private method is called, the private method is also moved down to the concrete class. When this is not possible, the operations within the method are moved inside the template method. This is not feasible in cases the operations rely on multiple other methods or sources. In such a case, the private method is changed to protected.

As for State/Strategy, other, less recurrent, variations are possible, but are not handled in this study. For example, a concrete class may aggregate the abstract class, possibly creating recursive calls, which are not originally intended for template method pattern instances.

4. EXPERIMENTAL PLANNING

In this section we present the design and materials of the experiment reported in this paper. This experiment is reported based on the guidelines of Wohlin et al. [38] and on the structure proposed by Jedlitschka et al. [39]. Initially, the research objective, questions and respective hypotheses of the study are discussed, followed by the process to select objects of study and experimental units. Next, an overview of the variables and instruments used to the data collection are presented. Finally, the analysis procedure is described. For presentation purposes, we report the data collection procedure along with the execution process, in Section 5.

4.1. Objectives, Research Questions, and Hypotheses

The goal of this study is defined according to the Goal-Question-Metrics approach [40], as follows: "*Analyze* instances of State, Strategy, and Template Method patterns *for the purpose of* evaluation *with respect to* their energy consumption *from the point of view of* software developers *in the context of* open source systems". To achieve this goal, we set three research questions (RQs):

RQ1 What is the difference between the application of the Template Method pattern and an alternative design solution in terms of energy consumption?

RQ2 What is the difference between the application of the State/Strategy pattern and an alternative design solution in terms of energy consumption?

RQ3 What are the parameters that influence the energy consumption of State, Strategy, and Template Method pattern instances?

RQ1 and **RQ2** aim at investigating whether the energy consumption of patterns and alternative solutions is significantly different. Such information is of paramount importance to make more informed decisions when selecting patterns over alternatives, while developing energy efficient software. To answer RQ1 and RQ2, we formulated the following hypotheses:

H0: There is no difference between the energy consumed by software using a design pattern solution and software using an alternative design solution.

H1: The energy consumed by software using a design pattern solution is significantly lower than the energy consumed by software using an alternative solution.

H2: The energy consumed by software using a design pattern solution is significantly higher than the energy consumed by software using an alternative solution.

RQ3 aims at exploring if there are pattern-related parameters that affect the energy consumption of the patterns, and for which ranges of these parameters the pattern can be characterized as beneficial or harmful. Such thresholds can serve as guidance for decision making on when to apply a design pattern or not. To answer this research question, we isolate groups (e.g., A and B) of pattern-participating methods whose members have a similar difference in the energy consumption (compared to the alternative solution) and investigate specific structural characteristics of the pattern solution (for more details, see Section 4.4). To test the difference between every two groups, we formulated the following hypotheses:

H3: There is no difference between the parameter values of the two groups (A and B).

H4: The parameter value of group A is higher than the value of group B.

H5: The parameter value of group B is higher than the value of group A.

4.2. Design Type and Experimental Units

To answer the research questions and test the hypotheses, we designed a crossover experiment [38], in which pattern-related methods are the experimental units. Pattern-related methods are methods of pattern instances that play a role within the design pattern. For our two selected patterns, these methods are the template method (Template Method pattern) and the methods implementing the behavior of states or strategies (States/Strategy pattern). We selected this unit for three reasons: (a) units with finer granularity facilitate a more detailed investigation of parameters (i.e., design characteristics) that influence the energy efficiency of design pattern solutions; (b) to standardize the data collection, since patterns may have multiple pattern-related methods, each one implementing different responsibilities; and (c) the alternative solutions provide the same functionality compared to pattern-related methods, but with a different implementation, what also promotes standardization of the data collection. For each experimental unit (i.e., a pair of pattern and alternative solutions), we record all data needed to answer the research questions, i.e., the energy consumption measurements for both pattern and alternative solutions, and design characteristics of the pattern solution.

To collect data for the experiment, it is necessary to select software systems and pattern instances from which to sample pattern-related methods. Regarding software systems, we decided to use OSS that met the following criteria:

- are written in the Java programming language, since the tool for retrieving design pattern instances (see Section 4.3.1) is limited to Java;
- are non-trivial systems that are either widely used or known, so as to avoid the use of toy examples; and
- contain instances of both the Template Method or the State/Strategy patterns.

Two OSS projects were selected for the study. Selecting more projects would be unrealistic as all alternative solutions had to be manually implemented by us, which is a time-consuming task. However, we do investigate a sufficient number of pattern instances (more than related work) and pattern-related methods. For further discussion, please see how we deal with threats to validity (Section 8). The first OSS system is ***JHotDraw***[5], a Graphical User Interface (GUI) framework written in Java that allows the creation of technical and structured graphical images. The project started in 2000, having about 80,000 downloads at this point, and the current version (7.6) has 680 Java source files, containing 80,535 SLOC. JHotDraw was developed as a design exercise, for applying GoF design patterns, becoming a powerful framework that is acknowledged by the software engineering community as a benchmark for GoF design patterns detectors [41], [42]. The second OSS system is

[5]http://www.jhotdraw.org/

Joda Time[6], an Application Program Interface (API) that can replace the standard date and time classes, providing better quality and in-depth functionalities. The project started in 2003, having almost 500K downloads at this point, and the current version (2.9.2) has 329 Java source files, containing 85K SLOC. Joda Time has a high rating on GitHub and has also been used for research purposes [43].

Despite the careful selection of representative software for the study, we acknowledge that non-trivial (complex) systems may have associated risks, in the sense that the transformation of a non-trivial pattern instance to an alternative solution might not be uniform. To mitigate this risk, we developed a strategy while selecting pattern instances / pattern-related methods, and implementing the alternative solutions. Firstly, to select pattern instances for the study, we consider only those that meet the following criteria:

- **Used within the application**: It is possible that the found pattern instances are not used within the applications themselves, e.g., functionalities provided as an API, whose pattern instances are partially implemented by the API user;
- **Reachable:** Some pattern instances are not reachable directly, imposing a long (and hard to predict) sequence of calls, what may bias the measurement process. One option is to modify the source code to make the pattern instance easier to reach, but it would bias the results as well;
- **Performing deterministic tasks:** Certain pattern instances may perform non-deterministic tasks, such as saving data to files or transferring data over the network. This could interfere with the actual measurement process; and
- **Not too complex**: In some cases, the pattern instances could have a relatively high number of members, e.g., twenty or more concrete states/strategies or are variants of the original pattern that are not handled in our study (see Sections 3.2. and 3.4). These pattern instances would make the process of implementing the alternatives infeasible. On top of that, such pattern instances would represent a threat to study validity, as these comprise exceptional cases.

Regarding method selection, the same criteria applied to pattern instances is used. We believe that the pattern instances and pattern-related methods filtered by these criteria are representative of the population, as excluded cases are mostly exceptional. Finally, concerning the implementation of alternative solutions, we have to ensure that the original business logic is preserved, avoiding unnecessary changes to the original source code. As the alternatives preserve the original business logic and only the difference in the energy consumption is analyzed, we believe that we have mitigated much of the risk associated with the usage of non-trivial programs.

4.3. Variables and Instrumentation

To answer the research questions and test the hypotheses stated in Section 4.1, a number of variables are derived. These variables are divided into two distinct categories: (a) *pattern-related information* (pattern, method and m-* in Table II, which are explained in Section 4.3.1); and (b) *measurements of energy consumption* (*-ptt and *-alt in Table II, which are explained in Section 4.3.2). These variables are recorded for each unit of analysis (i.e., pattern-related methods). The entire process of identifying and measuring the units of analysis culminates in the creation of a dataset of all extracted variables for each unit. This dataset is recorded as a table in which the columns correspond to collected variables. In the following subsections, we present and discuss the variables and the tools used to extract them.

Table II. List of collected variables

Variable	Description	Tool
pattern	Pattern Type (Template Method or State/Strategy)	SSA
method	The pattern-related method that is measured	
m-sloc	SLOC of the pattern-related method	-
m-mpc	MPC of the pattern-related method	
papi-ptt	Energy consumption (in Joules) of the pattern solution, at process level	PowerAPI
papi-alt	Energy consumption (in Joules) of the alternative solution, at process level	
jalen-ptt	Energy consumption (in Joules) of the pattern solution, at method level	Jalen
jalen-alt	Energy consumption (in Joules) of the alternative solution, at method level	
ptop-ptt	Energy consumption (in Joules) of the pattern solution for triangulation	pTop
ptop-alt	Energy consumption (in Joules) of the alternative solution for triangulation	

[6]http://www.joda.org/joda-time/

4.3.1. Pattern-related Information. To collect the necessary data for all units of analysis, we first find all the pattern occurrences within the OSS applications. To detect the design patterns occurrences, we use a tool developed by Tsantalis et al. [25]. This tool uses a Similarity Scoring Algorithm (SSA) for detecting design structures similar to a desired GoF design pattern. Among the 12 detectable patterns are Template Methods and State/Strategy (identified jointly due to structural similarity). The extraction of the design patterns is done by isolating subsystems of a given application through static analysis, which enables the identification of relationships between the elements of each separate subsystem. The SSA tool has been assessed by several studies (such as Kniesel et al. [44] and Pettersson et al. [45]), which have positively evaluated its performance, precision, and recall rates. SSA was, therefore, selected for this study because of the following:

- *it provides detection of the design patterns of interest*, i.e., Template Method and State/Strategy; and
- *it provides acceptable performance*, as described by Tsantalis et al. [25], also when compared to similar tools [44], [45].

SSA is limited to the Java programming language, since the similarity analysis is performed on compiled Java class files. After the application of the pattern detection tool on a project, the results are compiled into one Extensible Markup Language (XML) file that contains all the instances found within a given application.

Additionally, a set of metrics has to be extracted, which are used to investigate parameters that influence the energy consumption of pattern instances (see Section 4.4). In order to select these metrics, we considered the SQualE platform [46], as it summarizes a broad and comprehensive list of metrics from the literature. From this list, we identified two metrics that could be measured at method level: SLOC and MPC[7]. SLOC is measured as the amount of source line of code of the method, while MPC is measured as the amount of calls, within the method, to other methods (these calls do not include those to methods of the same class, even if inherited). We clarify that the parameters SLOC and MPC are calculated for the pattern solution only. For answering RQ3, we are interested in identifying characteristics of the pattern design solution that are related to energy efficiency. In addition, SLOC and MPC do not change considerably in the alternative solution, since the transformation mostly causes a reorganization of the code and how methods are called. In other words, our goal is not to evaluate the change of complexity, but how the complexity of the pattern solution influences the difference of energy consumption between the solutions, especially because this complexity is dictated by the business logic, which is not modified.

4.3.2. Assessment of Energy Consumption. To measure the energy consumption of software applications, there are multiple tools based on both software and hardware [47]. In this study we, opted to use software tools, as they allow finer-grained measurements (i.e., at the method level) [47]. Although hardware measurement offers a higher precision, it estimates the energy consumed by the whole machine, and our study investigates the consumption difference at the methods level. Therefore, we prioritized a finer-grained technique over a more precise one. In addition, selecting and configuring a hardware measurement tool may represent a complex and expensive task [48], which if not accurately performed can introduce additional bias. In order to select the appropriate tools, we searched the literature and identified nine software tools for measuring energy consumption. We analyzed two comparative studies that included these tools [47], [49], in addition to other literature, so as to verify their theoretical and empirical validity in scientific setups. Based on this analysis, two tools presented the highest precision, namely PowerAPI and pTop; a third tool, namely Jalen, although with lower precision, is able to deliver finer-grained measurements. Other tools that we considered either do not have sufficient validation or present lower precision regarding their respective granularity of measurement, or require additional hardware investments.

PowerAPI is an API that enables real-time profiling of the energy consumption at the level of operating system (OS) processes [10]–[12], [47]. This tool currently supports measuring energy from CPU and network, which are represented through power modules. The available implementations that are provided for this tool are created for GNU/Linux distributions, but they are independent of the hardware. To measure the energy consumption of the CPU, the Thermal Design Power (TDP) is taken into account, which is the maximum amount of heat (which is generated by the CPU) that requires to be dissipated by the cooling system. The precision for measuring the power consumption of software applications with PowerAPI was estimated by Noureddine et al. [11] by comparing it against a power meter. This estimation showed that the calculated margin of error vary from 0.5% to 3%.

[7]MPC consists of the number of invocations to methods that are not owned or inherited by the class being measured.

Jalen is an energy consumption profiler, which was created by the same developers of PowerAPI [11], [12], [47], [50]. Jalen can collect energy consumption on different levels of granularity such as the method level. Similarly to PowerAPI, Jalen is limited to the use on GNU/Linux distributions due to the sensors used for the hardware components. Since Jalen injects monitoring code through the bytecode instrumentation, it reduces the precision. In a comparison of tools performed by Noureddine et al. [11], the measured time for individual Tomcat's server requests was 57% higher in average. However, since we are comparing two different versions of the same applications (i.e., pattern and alternative solutions), this cannot be considered as a confounding factor.

pTop is a profiler that can determine energy consumption on the OS process-level and is designed to work solely on GNU/Linux distributions [47], [51]. pTop calculates the energy consumption through a daemon that profiles the resource utilizations for all processes, whereas the power consumption of the system CPU, network interface, memory and hard drive are tracked. Each different system component needs to be configured (possibly calibrated as well) according to its specifications. Just like PowerAPI, it uses the TDP to calculate the energy consumed by the CPU. The precision of pTop was analyzed by comparing its results to a wattmeter [47]. Results of this analysis show that the average median error for pTop was less than 2 watts.

All the aforementioned energy measurement tools are suitable candidates to obtain reliable results. However, PowerAPI and Jalen are designed to specifically measure the energy consumption of Java applications, not including the overhead caused by the Java Virtual Machine (JVM). Due to the granularity of the energy measurement of Jalen (i.e., method level), the output is not influenced by the energy expenditure of other parts of the system, which makes it a more suitable tool. However, in order to compare the related work to ours, it is also necessary to consider the same perspective used in related work, i.e., process level measurements, in this case by using PowerAPI. Therefore, we decided to use both PowerAPI and Jalen for the study. We clarify that both tools have a limitation of being able to measure energy consumed by the CPU only. Therefore, among other reasons, we restricted the experimental units to those that do not use extra resources (e.g., hard drive, or network). Additionally, we decided to use pTop, which is more commonly known in the scientific community, for triangulation purposes, to validate the measurements obtained from PowerAPI and Jalen, and to verify the memory energy consumption (see Section 5.2).

4.4. Analysis Procedure

During the data analysis, the previously described variables (see Table II) are used to answer the research questions. As mentioned in Section 4.3.2, we collect data using two different tools (PowerAPI and Jalen) and, therefore, every task of the analysis is performed for the data of each tool separately, and results are compared. In addition, the data regards two design patterns (Template Method and State/Strategy) and every step of the analysis is repeated for both patterns separately. The data analysis is twofold, described in the following.

4.4.1. General Analysis of Energy Consumption. Initially, we compare the energy measurements (**-ptt* and **-alt*) to test the hypotheses posed by research questions **RQ1** and **RQ2**. For evaluating whether or not the pattern solution is significantly different from the alternative solution, we perform two steps:

1) *Check distribution*. To decide whether to use parametric or non-parametric tests, we verify the distribution of each dependent variable metric (i.e., *papi-ptt, papi-alt, jalen-ptt,* and *jalen-alt*) by employing the Shapiro–Wilk test [52]. If not normal, a Wilcoxon signed ranks test [52] is used for assessing the difference between pattern and alternative solutions; otherwise, paired sample t-test [52] is used; and
2) *Compare energy consumption*. Next, we compare whether the difference between pattern and alternative solutions is statistically relevant. For that, we employ the dependent sample test for investigating the data obtained by PowerAPI and Jalen (i.e., *papi-ptt* vs. *papi-alt* and *jalen-ptt* vs. *jalen-alt*).

4.4.2. Analysis of Design Parameters. Once the difference in the energy consumption between pattern and alternative solutions is observed, we want to investigate parameters that may influence this difference. For that, we isolate controlled groups (i.e., clusters) with similar difference in the energy consumption and test the hypotheses posed by **RQ3**. This analysis comprises the following steps:

1) *Create clusters based on consumption*. First, we create clusters based on the difference between the energy measurements for PowerAPI (i.e., *papi-diff = papi-ptt – papi-alt*) and Jalen (i.e., *jalen-diff = jalen-ptt – jalen-alt*). For that, we employ the agglomerative hierarchical

clustering technique, considering the average linkage method (or between-groups linkage) and using squared Euclidian distance [53];

2) *Merge clusters based on design parameters*. Next, we investigate whether or not the clusters are statistically different with regards to the analyzed design parameters (*m-sloc* and *m-mpc*). As the clusters comprise independent samples, we employ Mann-Whitney tests [52] for this investigation. The analysis for each parameter is performed separately and clusters that are not statistically different are merged; and
3) *Verify trends*. Finally, based on the final disposition of the clusters, we verify trends with regards to both SLOC and MPC.

It is important to clarify that during the analysis we noticed cases in which the pattern solution was more energy efficient than the alternative solution and, however, the clustering algorithm did not separate these units (see Section 6.4). Therefore, aiming at complementing the answer for **RQ3**, an additional analysis is performed, which comprises the following steps:

1) *Group units*. Based again on the difference between the energy consumption, we separate the experimental units into two categories: (a) pattern solution consumed more energy than the alternative solution; and (b) pattern solution consumed less energy than the alternative solution; and
2) *Compare parameters*. Next, we analyze if the design parameters (SLOC and MPC) may have an influence on determining which solution is more energy-efficient. For that, we employ Mann-Whitney tests for investigating whether each parameter is statistically differ between the two groups created in the previous step.

5. EXECUTION

In this section we explain how data for the experiment was collected. Firstly, we describe the data collection procedure, showing details of the most relevant aspects. Next, we present and discuss the validation of the collected data according to the planned experiment.

5.1. Data Collection

The data collection is composed of four steps. Firstly, we extracted the pattern instances and selected the pattern-related methods (i.e., experimental units). To collect the experimental units, a set of pattern occurrences were extracted from JHotDraw and Joda Time, and were manually inspected to decide whether pattern instances could be included or excluded (see Section 4.2). Table III distinguishes between the number of pattern occurrences that were included and excluded (according to the process described in Section 4.2) for each OSS and GoF design pattern. For each included pattern instance, a set of units of analysis was collected. The total number of collected units for each OSS and GoF design pattern is presented between parentheses in Table III. We clarify that, despite the limited number of included pattern instances, we believe that the number of experimental units (95 and 74) is satisfactory, providing statistically significant results (see Section 6). Moreover, the effort required to implement the alternatives (as described in Sections 3.2 and 3.4) also restricted the amount of experimental units that could be collected.

Table III. Descriptive of identified pattern occurrences and pattern-related methods

OSS	**Included occurrences**		**Excluded occurrences**	
	TM	**SS**	**TM**	**SS**
JHotDraw	7 (**15**)	6 (**56**)	5	25
Joda Time	7 (**80**)	1 (**18**)	5	17
TOTAL	14 (**95**)	7 (**74**)	10	42

TM = Template Method, SS = State/Strategy

Next, for each unit, we calculated the parameters SLOC and MPC (i.e., based on the pattern solution, see Section 4.3.1). Before starting the measurement process, we implemented the alternative solution for each pattern instance as described in Sections 3.2 and 3.4. Then, to measure the energy consumption of the units, a standard measurement process was defined. This measurement process needed to be consistent throughout the whole test run, so no external interference is introduced to the results. First, a selection was done for the hardware system to be used for the analysis, along with the OS and distribution. For the hardware system, we chose the MSI wind box DC100 minicomputer due

to its simplicity, availability, and compatibility with the measurement tools. The MSI wind box contains the following components:

1) AMD Brazos Dual Core E-450 (1.65GHz) with a TDP of 18 Watts;
2) 4GB of DDR3 memory; and
3) AMD Radeon HD 6320 graphics adapter.

Since the measurement tools are tailored for GNU/Linux system, we used one of the distributions released for that OS. As we wanted less interference during the measurement process, a clean installation of Ubuntu is used, which contains only the essential packages and has no user interface. However, since JHotDraw requires a graphical shell to call certain functionalities, a simplistic window manager, i3[8], was installed on top of this distribution. For orchestrating and standardizing the execution of the measurement tools and pattern related methods, a script was created for performing the following procedure: start the measurement tool, wait a few seconds for the tool to load, execute the usage scenario containing the pattern-related method, wait for the application to finish and stop the measurement tool. Each usage scenario embedded multiple executions of a part of the application that called one pattern-related method (i.e., experimental unit), guaranteeing measurable energy consumption (i.e., more than 30 seconds). Any selected part of the application was the simplest possible and was fully checked to guarantee no hard external bias (e.g., read/write operations). Each usage scenario was executed with the pattern solution and the alternative solution. For reliability purposes, the aforementioned procedure was executed 100 times for every pair scenario-solution, obtaining 100 measurements for each experimental unit. Finally, we obtained the final value for each unit of analysis by excluding outlier measurements and calculating the average between the remaining measurements.

5.2. Validation of the Collected Data

There were three main assumptions in the experimental design that needed validation. Firstly, two researchers verified every manual data collection task. These tasks were the selection of the patterns instances and pattern-related methods, the calculation of the SLOC and MPC parameters, and measurement of energy consumption. Secondly, as we considered experimental units from State and Strategy pattern instances mutually, we verified whether there was a difference between the energy consumed by them. Our results suggest no visual or statistically relevant differences. Last, the energy consumption data was validated by triangulation.

As mentioned in Section 4.3.2, the energy consumption was obtained by two tools, one working at process level (PowerAPI) and another working at method level (Jalen). Our motivation for selecting these two tools was that they both estimate the energy consumption based on the JVM, therefore, reducing the bias from the overhead caused by the OS. In addition, PowerAPI has higher precision when compared to other tools, while Jalen, although having a lower precision, provides more fine-grained measurements (as it captures only the energy consumption of the method). By obtaining the two different perspectives, we aimed at comparing our study to related work, as well as verifying the results w.r.t. the different levels of measurements.

As expected, the tools provided measurements of different magnitudes, which are related to the different characteristics of the tools. In addition, PowerAPI and Jalen use a similar mechanism for exploring the JVM to calculate the results, which could be biased. Besides, both tools can collect the energy consumed by the CPU only and, although we restricted the experimental units to those not requiring additional resources (e.g., hard-drive, network), not considering the energy consumed by the memory could still represent a bias. Therefore, we sought to provide further validation of the estimated measurements. To this end, we selected a process level tool, pTop, which can estimate the energy consumed by both CPU and memory, as well as has a higher precision, but estimates measurements by exploring the process management of the OS.

The data collected by pTop suggest that the energy consumed by the memory is negligible (approx. 0.0001% of the total energy consumed for every experimental unit). In addition, to verify that our data collection was consistent, we triangulated the measurements. For that, we performed eight Spearman correlation tests. For each pattern (*pattern* = Template/Method or State/Strategy), we tested the correlation between each design solution (pattern and alternative) of PowerAPI/Jalen and pTop (i.e., *papi-ptt* vs. *ptop-ptt*; *papi-alt* vs. *ptop-alt*; *jalen-ptt* vs. *ptop-ptt*; and *jalen-alt* vs. *ptop-alt*) . By observing that all tests proved a rather very strong correlation (see Table IV), we considered all the measured data to be consistent and reliable for data analysis. Finally, it is interesting to notice that Jalen has a lower correlation to pTop, compared against PowerAPI. This is yet another evidence of the

[8]https://i3wm.org/

consistency of the results, as Jalen is a method level tool and, thus, do not have the overhead caused by the rest of the application.

Table IV. Pearson correlation test for validating estimated measurements from PowerAPI and Jalen

Pattern	Tool	Pattern solution (pTop)			Alternative solution (pTop)		
		N	Correlation Coefficient	Sig.	N	Correlation Coefficient	Sig.
Template Method	PowerAPI	95	0.946	< 0.01	95	0.947	< 0.01
	Jalen	89	0.893	< 0.01	87	0.877	< 0.01
State/Strategy	PowerAPI	74	0.963	< 0.01	74	0.929	< 0.01
	Jalen	71	0.933	< 0.01	71	0.791	< 0.01

6. ANALYSIS

In this section we present the results of the experiment. Firstly, we show the descriptive statistics of the dataset. Next, we present the results of the analysis carried out for each research question, which was executed as described in Section 4.4. We clarify that every statistical test was performed using the tool IBM SPSS Statistics[9] and are reported based on the guidelines suggested by Field [52].

6.1. Descriptive Statistics

For every experimental unit, pattern-related variables were collected (variables *pattern*, *method*, *m-sloc* and *m-mpc*), and an alternative solution was implemented as described in Section 3. Afterwards, the tools PowerApi, Jalen, and pTop were used to collect the energy consumption from both pattern and alternative solutions (**-ptt* and **-alt*). We remind that an experimental unit comprises a pair of pattern and alternative design solutions. A summary of all numeric variables (i.e., SLOC, MPC, and energy consumption measurements) is presented in Table V and Table VI, showing relevant descriptive statistics for Template Method and State/Strategy, respectively. As can be seen in Table V and Table VI, few measurements were performed by Jalen for the pattern and/or the alternative solution. This is due to a limitation from Jalen, which tries to measure a specific method, but it is unable to encapsulate the entire process. This is caused when either the length in time that the method uses is too little, or when the method delegates its functionality in a way that Jalen cannot track. Such cases were properly treated during the statistical analyses, which are discussed in the following subsections.

Table V. Descriptive statics of numeric variables for the Template Method pattern (*pattern* = Template Method)

Variable	N	Min	Max	Mean	Std. Error (Mean)	Std. Deviation
m-sloc[a]	95	2.00	36.00	6.03	0.59	5.75
m-mpc[b]	95	0.00	12.00	1.33	0.20	1.97
papi-ptt[c]	95	92.30	1086.77	327.88	27.11	264.23
papi-alt[c]	95	92.12	924.09	270.84	23.77	231.67
jalen-ptt[c]	89	43.85	799.88	200.38	14.57	137.47
jalen-alt[c]	87	22.58	777.32	150.13	10.38	96.84
ptop-ptt[c]	95	189.78	2198.84	719.86	59.68	581.72
ptop-alt[c]	95	193.85	2185.72	594.85	47.37	461.66

[a] Measured in number of uncommented lines in the pattern solution
[b] Measured in number of method invocations in the pattern solution
[c] Measured in Joules

Before performing the data analysis based on the energy measurements from PowerAPI (*papi-**) and Jalen (*jalen-**), these measurements were checked against the measurements from pTop (*ptop-**). The details of this validation process are presented and discussed in Section 5.2. When observing the measurement from the three tools, one can notice that they are different, following the order Jalen < PowerAPI < pTop. This difference in the measurements is expected. Jalen measures the consumption at a method level (i.e., not considering the consumption of the whole program); PowerAPI measures

[9] http://www-03.ibm.com/software/products/en/spss-statistics

the consumption of the Java process (i.e., the program); and pTop measures the consumption of the OS's process (i.e., which also include the overhead of the JVM). When ordering the values, it is possible to notice that greater overheads result in greater values, i.e., Jalen < PowerAPI < pTop.

Table VI. Descriptive statics of numeric variables for the State/Strategy pattern (*pattern* = State/Strategy)

Variable	N	Min	Max	Mean	Std. Error (Mean)	Std. Deviation
m-sloc[a]	74	0.00	36.00	5.68	0.69	5.93
m-mpc[b]	74	0.00	29.00	2.13	0.54	4.66
papi-ptt[c]	74	157.58	1664.17	738.17	56.51	499.11
papi-alt[c]	74	136.37	1002.25	341.95	19.88	175.54
jalen-ptt[c]	68	27.38	1260.11	486.34	42.54	350.75
jalen-alt[c]	66	20.20	635.89	186.96	14,72	119,56
ptop-ptt[c]	74	316.08	4124.87	1640.06	129.72	1145.63
ptop-alt[c]	74	273.21	2260.22	786.15	46.77	413.04

[a] Measured in number of uncommented lines in the pattern solution
[b] Measured in number of method invocations in the pattern solution
[c] Measured in Joules

6.2. RQ1: Template Method

The first research question aims at exploring the energy consumption of Template Method pattern instances and their alternative solutions, focusing on identifying if there is a statistically significant difference between the solutions (pattern and alternative) regarding energy consumption. For that, we considered the energy consumption measured by two different tools, one at process level (*papi-**) and one at method level (*jalen-**). While the former tool provides a more traditional and system-wide measurement, the latter provides a more fine-grained measurement allowing us to focus on the point of interest (pattern-related method), excluding any interference from the rest of the system. Although we did not expect to find differences in the results obtained from the two tools (because both pattern and alternative solutions are subject to the same interference), the method-level measurements should provide lower overhead (i.e., smaller energy measurements).

To answer this research question, we examined the pair of variables obtained from PowerAPI and Jalen (i.e., *papi-ptt* vs. *papi-alt* and *jalen-ptt* vs. *jalen-alt*). In order to decide if we were using parametric or non-parametric tests for assessing the statistical significance of the differences between the pair of variables, we employed the Shapiro–Wilk test to check the distribution of data for each variable. The results of the test suggest that data were not following the normal distribution and, thus, a non-parametric test had to be employed. Therefore, we used the Wilcoxon signed ranks test to evaluate the hypotheses posed for RQ1 (see Section 4.1) and, thus, investigate the energy consumption data from PowerAPI and Jalen. In addition, to support visualizing the difference in the energy consumption, Figure 4 shows the box-plot for each compared variable.

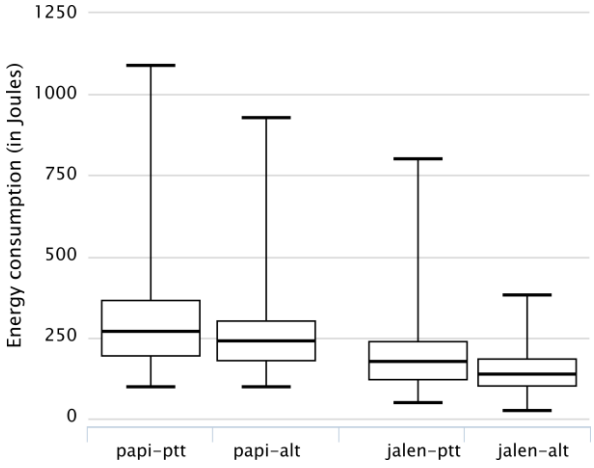

Figure 4. Visual comparison of the energy consumption for Template Method

From the analysis results, two main findings can be highlighted. First, pattern solutions consumed more energy than their alternatives based on the results of both PowerAPI ($p < 0.01$, $z = -4.92$) and

Jalen ($p < 0.01$, $z = -5.57$). This is evident in the results from both tools (following the result of the descriptive statistics—Table V), which suggest a decrease of 17.4% (PowerAPI) and 24.34% (Jalen) on the energy consumption of the alternative solution. Second, Jalen showed a greater difference than PowerAPI (by comparing the z-score), which also follows the trend observed by comparing their descriptive statistics. It is also important to highlight that, as expected, method level measurements (from Jalen) showed lower consumption then process level (from PowerAPI). These findings corroborate that: (a) method level measurements have lower overhead, since they isolate application and OS noises, and (b) pattern solutions indeed show increased energy consumption when compared against their alternatives. Summarizing, we can answer RQ1 by affirming that, for ***Template Method, pattern solutions tend to consume more energy than the alternative solutions (implemented as described in Section 3) and that this observation becomes more evident when analyzing at method level***. However, further investigation on this assertion is presented in Section 6.4 (see RQ3).

6.3. RQ2: State/Strategy

Next, we explored the energy consumption of State/Strategy pattern instances and their alternative solutions, focusing on identifying whether or not there is a statistically significant difference between the solutions (pattern and alternative) regarding energy consumption. For that, we followed the same process as described for RQ1, using, however, data related to State/Strategy (*pattern* = State/Strategy). Thus, first we performed the Shapiro–Wilk test to confirm that the data was not normal, as expected due to the result from the previous data analysis. As the variables were not normal, we used Wilcoxon signed ranks test to investigate the energy consumption data from PowerAPI and Jalen. In addition, to support visualizing the difference in the energy consumption, Figure 5 shows the box-plot for each compared variable.

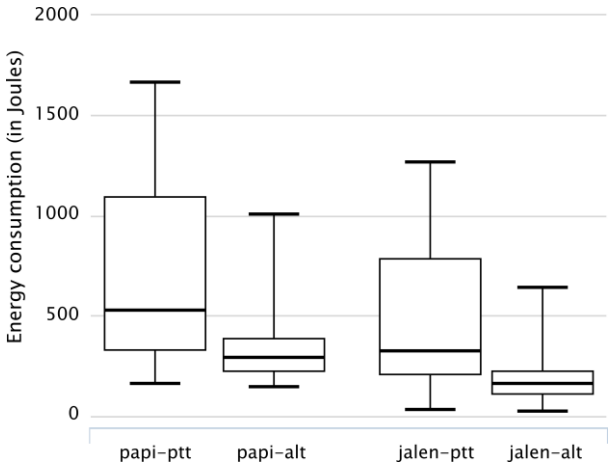

Figure 5. Visual comparison of the energy consumption for State/Strategy

Similarly to RQ1, the pattern solutions consumed more energy than their alternatives based on the results of both PowerAPI ($p < 0.01$, $z = -6.19$) and Jalen ($p < 0.01$, $z = -6.8$). This result is in accordance with what we expected from observing the descriptive statistics in Table VI and the box-plots in Figure 5. However, when looking at the differences between each pair of pattern and alternative solutions, we can highlight some notable aspects. First, results obtained from PowerAPI and Jalen are very close to each other, as it can be observed by the mean value in Table VI and z-score of the test. Second, the average results from Jalen are lower than those from PowerAPI, but still similar, especially by taking into consideration that Jalen only measures the energy consumption at method level. Even though these values are very close, the standard deviation and standard error mean (see Table VI) from Jalen are proportionally higher (in comparison with the mean) than those of PowerAPI. The relatively high standard deviation and standard error for the Jalen pair is caused by differences in measurements on method level. The method level measurements seem to be significantly distant from the mean. Nonetheless, the drop in energy consumption for both pairs is remarkable, as the average decrease in energy consumption is 53.68% for PowerAPI and 55.51% for Jalen. Summarizing, we can answer RQ2 by affirming that, ***concerning State/Strategy, pattern solutions tend to consume more energy than the alternative solution (implemented as described in Section 3), although method level measurements show that this result requires further investigation (due to the high standard deviation and error)***. We present this further analysis in the next section, in which we discuss parameters that influence energy consumption.

6.4. RQ3: Influence of Source Code Parameters

The third research question aims at investigating parameters that influence the energy consumption of design pattern instances. To achieve this goal, we considered two metrics (SLOC and MPC) collected from every pattern-related method (i.e., based on the pattern solution, see Section 4.3.1) to investigate clusters of experimental units via a three-step analysis. First, to cluster the experimental units, we performed an agglomerative hierarchical clustering (using between-groups linkage and squared Euclidean distance, see Section 4.4.2) based on the difference in energy consumption (i.e., **-diff* = **-ptt* – **-alt*). Second, we employed Mann-Whitney tests to evaluate the hypotheses posed for RQ3 (see Section 4.1), verifying whether neighbor clusters (i.e., that are at the same level in the hierarchical tree) are statistically different w.r.t. SLOC and MPC. If no statistically significant difference was found, we merged the clusters and performed the test again with the neighbor of the merged cluster. Finally, we investigated the final clusters to identify trends regarding the studied metrics (SLOC and MPC).

In Figure 6 we present the outcome of the hierarchical clustering for Template Method data. The two charts on the top show the distribution of the experimental units among the clusters. Each point consists of a pair *<pattern solution; alternative solution>*, in which the Y axis is the energy consumption of the pattern solution and the X axis is the energy consumption of the alternative. The clusters can be identified by the different shape and color presented in the legend. The two charts on the bottom of Figure 6 show the centroids of the clusters with regards to SLOC and MPC. The values for SLOC and MPC of each cluster are obtained as the average of the units of the cluster. By checking these charts, it is already possible to notice some separation between clusters w.r.t. the two metrics.

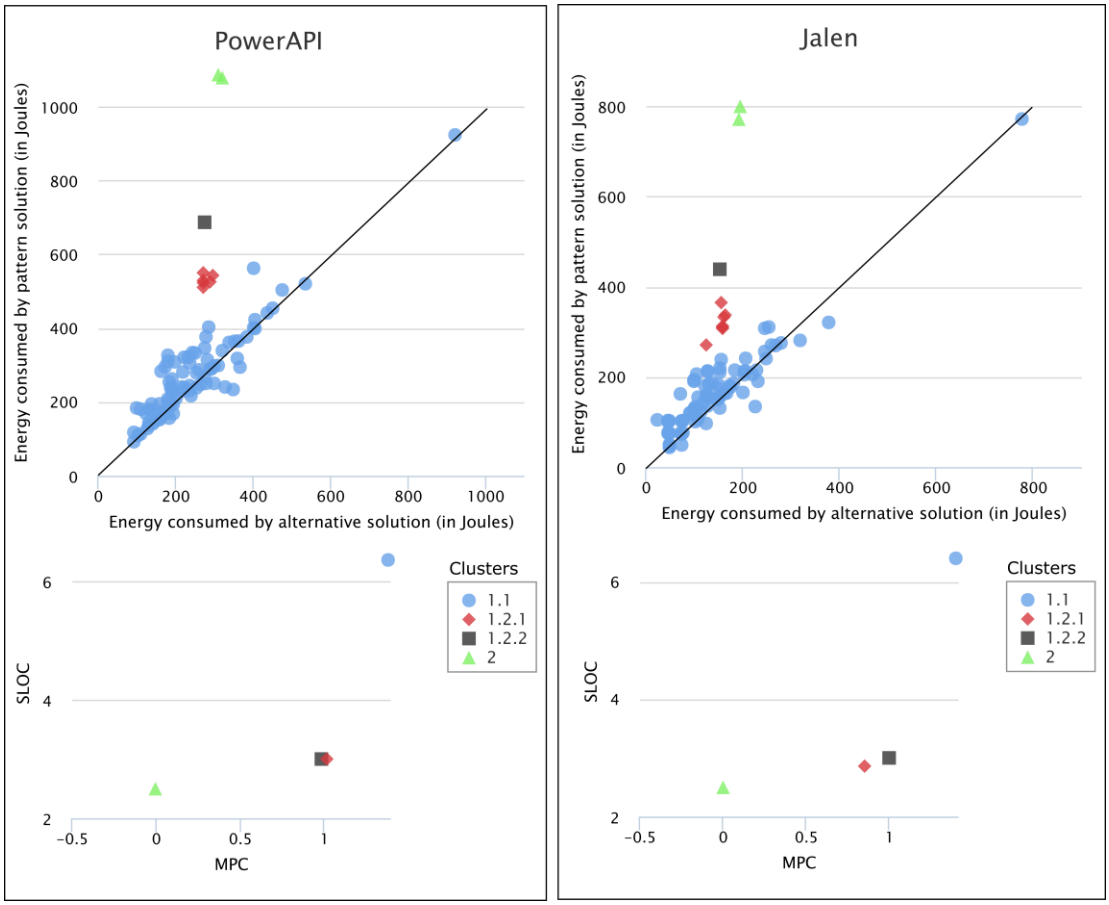

Figure 6. Hierarchical clustering of Template Method units of analysis

We investigated the separation between clusters, and the results of the statistical tests are presented in Table VII (along with the tests for State/Strategy data). As multiple tests were performed for investigating every pair of cluster regarding every metric, we report only the statistically significant results. Based on these tests, one can see which clusters were merged. For example, when comparing the clusters of Template Method (see upper charts of Figure 6) one can see four clusters (1.1; 1.2.1; 1.2.2; 2), from which two are very close (1.2.1; 1.2.2) and the separation was not statistically relevant w.r.t. to SLOC and MPC, forming a merged cluster (1.2). By further inspecting the statistical tests and

charts, one can see three clusters distant from each other (1.1; 1.2; 2) w.r.t. to both SLOC and MPC, and that they follow a trend in which clusters that group more energy-efficient solutions (e.g. 1.1) have bigger SLOC and MPC scores. This observation suggests that the higher the SLOC and MPC, the less advantageous the alternative solutions.

Table VII. Mann-Whitney test for comparing clusters

Pattern	Tool	Metric	Mann-Whitney test				
			Clusters			Z	Sig.
Template Method	PowerAPI	SLOC	1.1	&	1.2	-2.46	0.02
	Jalen	SLOC	1.1	&	1.2	-2.94	< 0.01
State/Strategy	PowerAPI	SLOC	2.1	&	2.2	-3.62	< 0.01
		MPC	2.1	&	2.2	-2.86	< 0.01
		SLOC	1	&	2.1	-4.77	< 0.01
		MPC	1	&	2.1	-5.03	< 0.01
		SLOC	1	&	2	-4.31	< 0.01
		MPC	1	&	2	-4.70	< 0.01
	Jalen	SLOC	1.1.1	&	1.1.2	-2.91	< 0.01
		MPC	1.1.1	&	1.1.2	-2.16	0.03
		SLOC	1.1	&	1.2	-2.46	0.01
		MPC	1.1	&	1.2	-3.83	< 0.01
		SLOC	1	&	2	-3.62	< 0.01
		MPC	1	&	2	-3.63	< 0.01

Figure 7 shows the scatterplots that concern the State/Strategy pattern, on which we performed the same analysis described for Template Method. When investigating the clusters (based on the statistical tests—see Table VII), one can deduce that three clusters remain for PowerAPI (1; 2.1; 2.2) and four for Jalen (1.1.1; 1.1.2; 1.2; 2). Although the data of the two tools led to slightly different clusters, the results suggest the same trends, which are similar to the ones observed for the Template Method pattern. In particular, both SLOC and MPC influence the benefit of using an alternative instance instead of the pattern solution. Moreover, the cluster 1.1 from the PowerAPI data (which is similar to cluster 1.1.1 from Jalen) is the closest to the bisect line (i.e., pattern solution = alternative solution) and, by checking the metrics chart, it is clear that this cluster has much higher SLOC and MPC when compared to the others.

The performed analysis is so far able to provide evidence that both SLOC and MPC influence the energy efficiency of a pattern solution and that both parameters should be taken into account when deciding between using a pattern solution or an alternative solution. However, there is one interesting question that has not been answered by the clustering, yet. By observing all the scatterplots that were presented until this point, it is clear that in some cases pattern solutions were more energy efficient than the alternative solutions (i.e., experimental units below the bisect line in the upper charts of Figures 6 and 7). However, the use of automated clustering algorithms did not separate these units. Therefore, we decided to perform a second analysis. We grouped the experimental units into the two categories (pattern > alternative; and pattern < alternative). Next, we were able to investigate whether or not SLOC and MPC may have an influence on determining if a pattern solution is more energy-efficient than the alternative solution. To explore the differences between these groups, in terms of SLOC and MPC, we employed Mann-Whitney tests. Table VIII shows the results of the test for both Template Method and State/Strategy.

Based on the results of Table VIII, it becomes clear that SLOC has a significant influence on the energy efficiency of the pattern instance for both GoF design patterns, suggesting that the longer the method is, the more possible it becomes that the pattern solution is more energy efficient. For the State/Strategy pattern, it is also statistically evident that the number of calls to other methods influences the energy efficiency of the solution, suggesting that more calls are related to a higher possibility of the pattern solution being more efficient than the alternative.

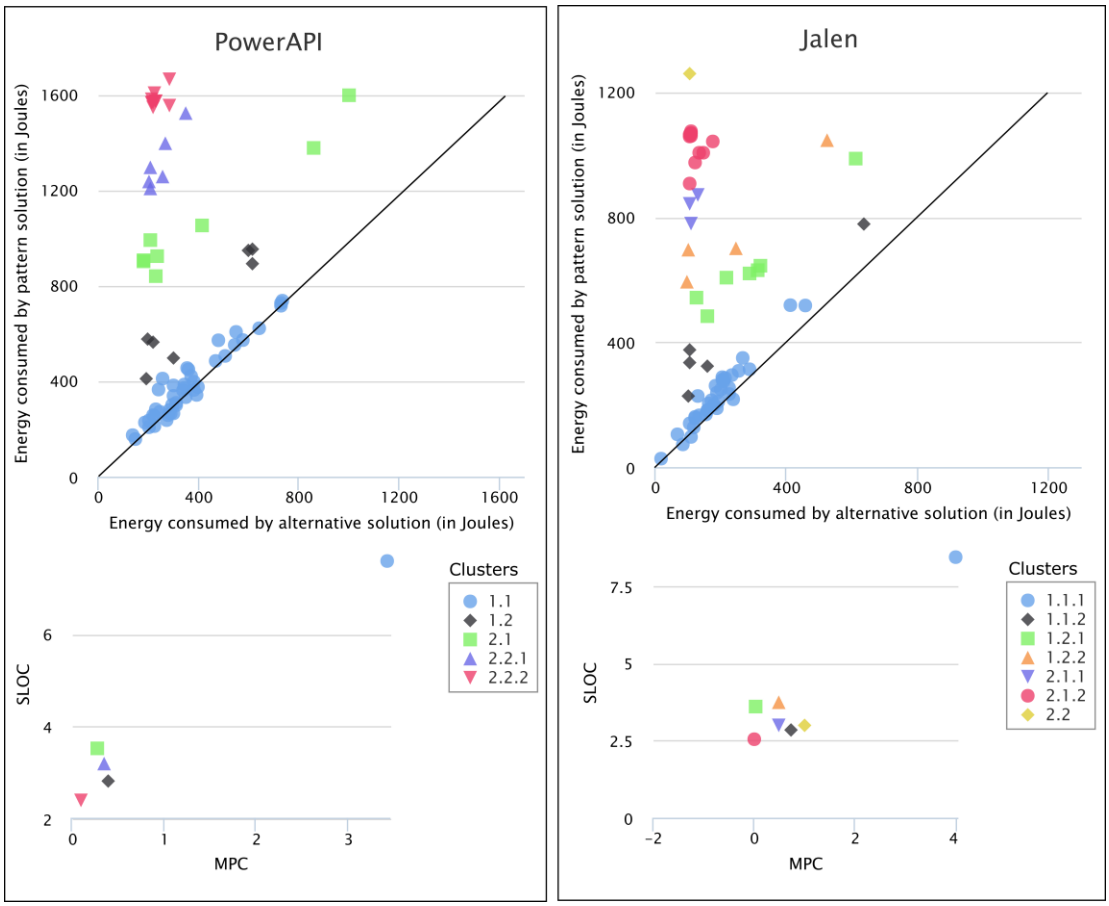

Figure 7. Hierarchical clustering of State/Strategy units of analysis

Table VIII. Mann-Whitney test for comparing most energy efficient solutions

Pattern	Tool	Metric	Mann-Whitney test	
			Z	Sig.
Template Method	PowerAPI	SLOC	-4.06	0.00
		MPC	-0.03	0.98
	Jalen	SLOC	-3.75	< 0.01
		MPC	-1.71	0.09
State/Strategy	PowerAPI	SLOC	-4.05	< 0.01
		MPC	-3.57	< 0.01
	Jalen	SLOC	-2.15	0.03
		MPC	-2.49	0.01

Summarizing the evidence so far, it is possible to answer RQ3 by affirming that ***both parameters, i.e., number of source lines of code and the number of invoked methods, influence the energy efficiency of a pattern solution, suggesting that higher SLOC and/or MPC are related to more energy efficient pattern solutions when compared against their alternative solutions***.

7. DISCUSSION

In this section we discuss the main outcomes of this study. First, we discuss the findings of the experiment, comparing them with related work. Second, we discuss the implications to researchers and practitioners. However, we need to clarify that the discussion presented in this section regards only the Template Method and State/Strategy patterns, as well as that our observations and interpretations are constrained by the limitations of the experimental settings and threats to validity (see Section 8).

7.1. Interpretation of Results

The results of our experiment suggest that the alternative solutions are more energy efficient than the pattern solutions for both Template Method and State/Strategy. This difference is higher for State/Strategy (approx. 54% for PowerAPI and 56% for Jalen) than to Template Method (approx. 17% for PowerAPI and 24% for Jalen). These results are in accordance to related studies (see Section 2), which have reached similar conclusions, i.e., that the alternative solutions tend to be more energy efficient. Specifically, Bunse et al. [29], as well as Noureddine and Rajan [32], also report on the Template Method pattern, and suggest that this pattern presents a small, yet significant, overhead. Noureddine and Rajan [32] also investigate State and Strategy patterns separately, and report a smaller overhead for State (approx. 3%) and an equally small improvement for Strategy (approx. 3%). This difference between results may be related to certain characteristics of the study design (e.g., the used pattern alternative or subjects of the study), but more details regarding these characteristics would be necessary to elaborate on the rationale. To sum up, the differences between pattern and alternative solutions observed in our study are likely to be influenced by the overhead caused by employing polymorphism (i.e., the main mechanism of both patterns). When calling polymorphic methods, the JVM has to dynamically indicate the correct implementation to be used. Commonly, this indication is done by moving the instruction pointer[10] to the memory address containing the right method. Although simple, this kind of operation can become computationally expensive if overused.

While investigating the influence of SLOC and MPC on the energy consumption of pattern solutions, we were able to notice that both GoF patterns tend to provide a slightly more energy-efficient solution when used to implement more complex behaviors (i.e., with longer methods and multiple calls to method of external classes). This observation is also intuitive from three perspectives:

1) GoF design patterns are not beneficial in simple/non-complex design problems (even w.r.t. other quality attributes [18][19]), since the extra complexity that they introduce is higher than the one that they resolve;

2) The effect of polymorphism weakens when these patterns are handling complex situations. The longer the method, the lower the ratio of method localization compared to the overall computation and, therefore, the overall overhead caused by the polymorphic mechanism of Template Method or State/Strategy; and

3) It is understandable that patterns promote improved structuring of the source code, which may sometimes lead to a smaller and/or more efficient bytecode (for the JVM), which in turn leads to slightly more energy-efficient software. We observed such cases, e.g., when the pattern-related method comprises a set of external invocations (i.e., to methods that are not owned or inherited by the class being measured). In such cases, the JVM might be applying internal optimizations, which would not be possible in the alternative, as the structure pattern-related method is altered.

Although we have provided evidence that alternative solutions are in most of the cases more energy efficient than pattern solutions (approx. 79% of the cases), there are cases in which the opposite holds. Sahin et al. [30] have also reported on pattern instances that can be more energy efficient compared to alternative solutions. In comparison to Sahin et al., we provide a more fine-grained analysis by relating this differentiation to two metrics (i.e., SLOC and MPC). This finding can also be possibly explained by the overhead caused by polymorphism, as we were able to identify statistically significant differences on the metrics between pattern-efficient (i.e., pattern solution consumed less energy than the alternative solution) cases and alternative-efficient cases. On average, pattern-efficient solutions have 65.83% more source lines of code and 43.37% more method invocations than the alternative-efficient solutions.

Finally, there is a crosscutting observation to all findings in this paper, which deals with differences in energy consumption at method and process levels. The measurements from Jalen were lower than the measurements from PowerAPI (40.42% on average). This observation is intuitively correct since the measurements from Jalen are more localized (focused on only one method). Furthermore, it is interesting to notice that differences between pattern and alternative solution were smaller for Jalen (12.11% in average), a fact that suggests that the remaining parts of the applications (i.e., not the pattern-related methods) were, to some extent, biasing the analysis. Another possible explanation could be that the dynamic binding procedure[11] may not be fully captured by Jalen at times, as it focuses on the pattern-related method being measured. However, we sought to mitigate this threat by: (a) verifying cases of dynamic biding while selecting experimental units (i.e., pattern related methods); (b)

[10]Also known as program counter, instruction address register, instruction counter and instruction sequencer, instruction pointer is a processor register that indicates the current assembly command to be executed.

[11]Dynamic binding procedure refers to the action of resolving a binding (e.g., decide which method or variables with same names to use) at runtime, when it is not possible at compile time.

looking for outlier measurements; and (c) checking the correlation of the measurements against pTop (see Section 5.2).

7.2. Implications to Researchers and Practitioners

The findings of this paper suggest that pattern solutions are less harmful or even beneficial to energy consumption when the responsibility assigned to the pattern instance (i.e., the implemented behavior) is non-trivial. Therefore, we advise practitioners on considering this parameter when deciding whether or not to apply Template Method or State/Strategy patterns. GoF patterns serve several purposes: structuring and organizing source code; supporting quality attributes, such as maintainability and reusability; and improving communication between stakeholders by providing a common language. For these reasons, GoF patterns have become a common practice in software development. Several studies that have investigated only a subset of the GoF patterns report that approx. 30% of the classes of a system may participate in pattern instances [15], [16], [54]. However, as studies have shown, there are also side effects on using GoF patterns [17], and energy efficiency is one of the aspects in which the software is negatively affected. Thus, we also advocate the careful consideration of drivers (e.g., energy efficiency) of the software project, balancing them against the forces (e.g., complexity of the behavior to be implemented) that influence the decision on applying a certain pattern or not.

Based on the aforementioned negative relationship between GoF patterns and energy efficiency, one may wonder why using GoF patterns in systems that have energy efficiency as a main concern. Nevertheless, GoF patterns are widely adopted and, therefore, we expect that even systems that have energy as a concern may have a non-negligible amount of GoF pattern instances, either intentionally (to promote other quality attributes) or unintentionally. Therefore, the results of our study can be used to help control a system's efficiency in different situations. On the one hand, while developing software, our findings may support the management of unintentional harm to energy efficiency (via not necessary use of GoF patterns), as well as intentional use to balance various quality attributes. On the other hand, when refactoring a system for a new purpose, the findings of this study may support the decision making process on what parts of the system to refactor and how.

This study has three main implications to researchers. First, the usage of non-trivial systems for investigating patterns energy consumption is a challenging task, since researchers need to a) deal with pattern variants, b) decide which variants have to be investigated, c) incorporate these variations into the alternative solution, and d) measure the energy consumption of the pattern instance by executing the same scenario for which the pattern instance was intended to. However, the obtained evidence can be very insightful as shown by this study. Therefore, we do suggest that when investigating the energy consumption of GoF patterns, non-trivial systems should be used. Second, the use of method level energy measurements has proven to provide extra information for investigating the hypotheses both visually and statistically. It also contributed to the reliability of our findings by triangulating results of process and method level measurements. Third, when investigating the energy efficiency of GoF patterns, exploring design parameters (e.g., SLOC and MPC) proved to be highly relevant. By investigating the parameters, we were able to not only suggest whether or not the pattern solution is worse than the alternative solution, but also, and most importantly, we were able to interpret this phenomenon. By further investigating this hypothesis and observing the magnitude of the influence of these parameters, we were able to highlight the circumstances under which the patterns are more efficient than the alternative. Therefore, we suggest exploring similar parameters and other design and source code properties when investigating the influence of GoF design patterns to energy consumption.

8. THREATS TO VALIDITY

In this section, threats to construct validity, internal validity, reliability, and external validity of this study are discussed. Construct validity reflects how far the studied phenomenon is connected to the intended studied objectives. Internal validity expresses to what extent the observed results are attributed to the performed intervention, and not to other factors. Reliability is linked to whether the experiment is conducted and presented in such a way that others can replicate it with the same results. Finally, external validity deals with possible threats when generalizing the findings derived from sample to the entire population.

Concerning construct validity, one threat is that the transformation of non-trivial systems may be risky since, due to their complexity, it is more error-prone. Although "synthetic" programs could facilitate the control over external factors, we believe that non-trivial programs were imperative to investigate pattern-related methods. Thus, to mitigate this bias, we took several measures while selecting experimental units (see Sections 4.2 and 4.3). The collected energy measurements pose another threat, as we consider consumption only by the CPU. If we included energy consumed by other

resources, such as hard drive and network, the results might change. First, by only looking at CPU consumption, it enabled us to use three different measurement tools to increase the confidence on the obtained measurements. To mitigate this threat further, we verified that the energy consumed by the memory was negligible and do not represent a considerable bias (see Section 5.2), as well as restricted the selection of pattern instances to those that do not require operations such as writing to or reading from files and communicating through network. Another threat concerns the level of measurement (i.e., process or method level), which can be a source of bias to the study as different perspectives could lead to different results. For that reason, we performed the analysis at both levels (process and method), and checked their correlation. Additionally, some lack of precision could have been introduced by a limitation of the used energy measurement tools. To mitigate this threat, we selected tools that have been validated in different studies. In addition to that, we performed data triangulation for all measurements by using three different tools. Moreover, the measured data may also be slightly biased, since small environmental changes might exist between different executions, leading to different values. To mitigate this threat, we used a basic OS, installing only strictly required dependencies, and every measurement was performed multiple times, using the average value for the analysis.

The main threat to the internal validity of our experiment is related to whether the observed differences in the energy consumption were caused by the implemented alternatives, and not by other factors. To mitigate this threat, we acted from measurement and implementation perspectives. On the one hand, we used Jalen, which is able to measure only the energy consumed by the experimental unit (i.e., pattern related-method), discarding the energy consumed by the rest of the application, JVM, and OS. In addition, the procedure to measure the energy consumed by pattern and alternative solutions was identical. On the other hand, while implementing the alternatives, we assured that only the design changes proposed for the alternatives (see Sections 3.2 and 3.4) were implemented, not altering the behavior of the pattern-related method. Another threat to this category is the fact that the set of parameters that we investigated for answering RQ3 is not exhaustive, and we cannot guarantee that differences in energy consumption have been comprehensively explained, since there might be other parameters that influence the energy consumption of design patterns.

In order to mitigate reliability threats, two different researchers were involved in the data collection, double-checking all outputs. In addition to the two researchers, a third one was involved in the analysis procedure. To implement the alternative solutions, the provided guidelines are sufficient and any replication should lead to the same results. To complement that, all scripts and source code are available on-line[12] and, therefore, all raw data can be reproduced with small variations by using the same energy measurement tools and environment setup. Finally, data analysis bias is limited in this study, since no subjectivity was involved.

Finally, concerning external validity, we have identified four possible threats. First, we investigated a limited number of OSS projects. However, the two selected projects are very different, both in terms of domains and characteristics (e.g., Joda Time has more than the double of SLOC per class when compared to JHotDraw); this partially alleviates this threat. Second, we investigate a limited number of pattern instances, as well as a limited range of pattern variants. However, we evaluate a fair number of pattern-related methods (i.e., units of analysis), what partially alleviates this threat. Nevertheless, a larger sample could strengthen the results, and increase our confidence on generalizing our findings. Next, the presented results are dependent on the used alternatives and pattern solutions. Thus, different alternatives or pattern variations could lead to altered results. For example, alternative and/or pattern solutions optimized for energy efficiency may increase the observed difference between the solutions, or even invert it. However, the focus of our study was to analyze representatives of existing and commonly used non-trivial software, in terms of both pattern and alternative design solutions, as such investigation would impact a plethora of software. Therefore, we selected the alternatives that we believe to be the most common, as well as considered the original definition of the studied patterns (also with small and similarly common variations), so as to have a more representative sample of solutions that exist in practice. Finally, the results of this study cannot be directly generalized to other GoF patterns, especially those that do not use polymorphism as their main mechanism.

9. CONCLUSIONS

In this paper we investigated the effect of Template Method, and State/Strategy GoF design patterns on energy consumption. In particular, we conducted an experiment on two non-trivial OSSs, JHotDraw and Joda Time, from which we identified 21 pattern instances and 169 pattern-related methods (i.e., methods that use the pattern structure), implemented an alternative (non-pattern) solution for each

[12]http://www.cs.rug.nl/search/uploads/Resources/JSEP_Feitosa_etal_resources.zip

instance (which contained the alternative implementation of the pattern-related methods), and measured the energy consumption of both solutions using tools at both process and method levels. Based on the collected data, we identified which solution was more energy-efficient and what parameters affect the efficiency of the pattern solution. To this end, we collected two metrics from every pattern-related method, SLOC and MPC, and correlated them to the efficiency of the pattern solution. The results of the study suggest that the alternative solution excels the pattern solution in most cases. However, in some cases the pattern solution had similar or even slightly lower energy consumption than the alternative solution. Since these cases were identified in large pattern-related methods and/or methods with high number of method invocations, it is suggested that these patterns are more suitable when more complex behaviors have to be implemented. We clarify that some factors, such as the considered design pattern alternatives, may have influence on the aforementioned observations, and that altering these factors may change the aforementioned observations (for more details, see Section 8).

The findings of this study have value for both practitioners and researchers. On the one hand, practitioners can reuse this knowledge to perform more informed decision-making when applying GoF patterns. On the other hand, researchers can learn from the reported experiences and reproduce aspects of this study when investigating GoF design patterns and/or energy consumption. Finally, there are several opportunities of future work. This study can be replicated with more experimental units or more source code metrics. Different tools, especially hardware tools, can be used to not only triangulate the results but also investigate other effects (e.g., from the remainder of the OS or the computer itself). In addition, the same or improved setup can be used to investigate other GoF design patterns, specially focusing on the other two pillars of object-orientation that have not been investigated in depth by this study (i.e., encapsulation and inheritance). Lastly, a case study can be carried out on systems that have energy efficiency among their main concerns, investigating how GoF patterns and alternatives are used within such context and comparing these systems with other kinds of systems.

ACKNOLEDGEMENTS

The authors would like to thank the financial support from the Brazilian and Dutch agencies CAPES/Nuffic (Grant N.: 034/12), CNPq (Grant N.: 204607/2013-2), as well as the INCT-SEC (Grant N.: 573963/2008-8 and 2008/57870-9).

REFERENCES

1. Procaccianti G, Lago P, Bevini S. A systematic literature review on energy efficiency in cloud software architectures. *Sustainable Computing: Informatics and Systems* 2015; 7:2–10, doi:10.1016/j.suscom.2014.11.004.
2. Hammadi A, Mhamdi L. A survey on architectures and energy efficiency in data center networks. *Computer Communications* 2014; **40**:1–21, doi:10.1016/j.comcom.2013.11.005.
3. Zhang Z, Cai YY, Zhang Y, Gu DJ, Liu YF. A distributed architecture based on microbank modules with self-reconfiguration control to improve the energy efficiency in the battery energy storage system. *IEEE Transactions on Power Electronics* 2016; **31**(1):304–317, doi:10.1109/TPEL.2015.2406773.
4. Pinto G, Castor F, Liu YD. Understanding energy behaviors of thread management constructs. *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ACM, 2014; 345–360, doi:10.1145/2714064.2660235.
5. Liu YD. Energy-efficient synchronization through program patterns. *Proceedings of the First International Workshop on Green and Sustainable Software*, IEEE, 2012; 35–40, doi:10.1109/GREENS.2012.6224253.
6. Peréz-Castillo R, Piattini M. Analyzing the harmful effect of god class refactoring on power consumption. *IEEE Software* 2014; **31**(3):48–54, doi:10.1109/MS.2014.23.
7. Sahin C, Pollock L, Clause J. How do code refactorings affect energy usage? *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ACM, 2014; 1–10, doi:10.1145/2652524.2652538.
8. Johann T, Dick M, Naumann S, Kern E. How to measure energy-efficiency of software: Metrics and measurement results. *Proceedings of the First International Workshop on Green and Sustainable Software*, 2012; 51–54.
9. Tiwari V, Malik S, Wolfe A, Lee MTC. Instruction level power analysis and optimization of software. *Technologies for Wireless Computing*. Springer US, 1996; 139–154, doi:10.1007/978-1-4613-1453-0\ 9.
10. Noureddine A, Bourdon A, Rouvoy R, Seinturier L. A preliminary study of the impact of software engineering on GreenIT. *Proceedings of the First International Workshop on Green and Sustainable Software*, IEEE, 2012; 21–27.
11. Noureddine A, Rouvoy R, Seinturier L. Monitoring energy hotspots in software. *Automated Software Engineering* 2015; **22**(3):291–332, doi:10.1007/s10515-014-0171-1.
12. Noureddine A, Bourdon A, Rouvoy R, Seinturier L. Runtime monitoring of software energy hotspots. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ACM, 2012; 160–169, doi:10.1145/2351676.2351699.
13. Jain R, Molnar D, Ramzan Z. Towards understanding algorithmic factors affecting energy consumption: switching complexity, randomness, and preliminary experiments. *Proceedings of the 2005 Joint Workshop on Foundations of*

Mobile Computing, 2005; 70–79, doi:10.1145/1080810.1080823.
14. Gamma E, Helm R, Johnson RE, Vlissides J. *Design patterns: elements of reusable object-oriented software*. 1995.
15. Khomh F, Gueheneuc YG, Antoniol G. Playing roles in design patterns: An empirical descriptive and analytic study. *Proceedings of the IEEE International Conference on Software Maintenance*, IEEE, 2009; 83–92, doi:10.1109/ICSM.2009.5306327.
16. Ampatzoglou A, Chatzigeorgiou A, Charalampidou S, Avgeriou P. The effect of GoF design patterns on stability: A case study. *IEEE Transactions on Software Engineering* 2015; **41**(8):781–802, doi:10.1109/TSE.2015.2414917.
17. Ampatzoglou A, Charalampidou S, Stamelos I. Research state of the art on GoF design patterns: A mapping study. *Journal of Systems and Software* 2013; **86**(7):1945–1964, doi:10.1016/j.jss.2013.03.063.
18. Huston B. The effects of design pattern application on metric scores. *Journal of Systems and Software* 2001; **58**(3):261–269, doi:10.1016/S0164-1212(01)00043-7.
19. Hsueh NL, Chu PH, Chu W. A quantitative approach for evaluating the quality of design patterns. *Journal of Systems and Software* 2008; **81**(8):1430–1439, doi:10.1016/j.jss.2007.11.724.
20. Fowler M, Beck K, Brant J, Opdyke W, Roberts D. *Refactoring: Improving the Design of Existing Code*. Object technology series, Addison-Wesley, 1999.
21. Adamczyk P. Selected patterns for implementing finite state machines. *Proceedings of the 11th Conference on Pattern Languages of Programs*, 2004; 1–41.
22. Saúde AV, Victório RASS, Coutinho GCA. Persistent state pattern. *Proceedings of the 17th Conference on Pattern Languages of Programs*, ACM, 2010; 1–16, doi:10.1145/2493288.2493293.
23. Lyardet FD. The dynamic template pattern. *Proceedings of the Conference on Pattern Languages of Design*, 1997; 1–8.
24. Ampatzoglou A, Charalampidou S, Stamelos I. Design pattern alternatives. *Proceedings of the 17th Panhellenic Conference on Informatics*, ACM, 2013; 122–127, doi:10.1145/2491845.2491857.
25. Tsantalis N, Chatzigeorgiou A, Stephanides G, Halkidis ST. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering* 2006; **32**(11):896–909, doi:10.1109/TSE.2006.112.
26. Ampatzoglou A, Charalampidou S, Stamelos I. Investigating the use of object-oriented design patterns in open-source software: A case study. *Proceedings of the International Conference on Evaluation of Novel Approaches to Software Engineering*. Springer Berlin Heidelberg, 2011; 106–120, doi:10.1007/978-3-642-23391-3\ 8.
27. Weisfeld M. *The Object-Oriented Thought Process*. 4th edn., Addison-Wesley Professional, 2013.
28. Harper R, Morrisett G. Compiling polymorphism using intensional type analysis. *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 1995; 130–141, doi:10.1145/199448.199475.
29. Bunse SSC, Schwedenschanze Z, Stiemer S. On the energy consumption of design patterns. *Proceedings of the 2nd Workshop EASED@ BUIS Energy Aware Software-Engineering and Development*, 2013; 7–8.
30. Sahin C, Cayci F, Gutierrez ILM, Clause J, Kiamilev F, Pollock L, Winbladh K. Initial explorations on design pattern energy usage. *Proceedings of the First International Workshop on Green and Sustainable Software*, IEEE, 2012; 55–61, doi:10.1109/GREENS.2012.6224257.
31. Litke A, Zotos K, Chatzigeorgiou A, Stephanides G. Energy consumption analysis of design patterns. *Proceedings of the International Conference on Machine Learning and Software Engineering*, 2005; 86–90.
32. Noureddine A, Rajan A. Optimising energy consumption of design patterns. *Proceedings of the 37th International Conference on Software Engineering*, IEEE, 2015; 623–626.
33. Adamczyk P. The anthology of the finite state machine design patterns. *Proceedings of the 10th Conference on Pattern Languages of Programs*, 2003; 1–25.
34. Ferreira LL, Rubira CMF. The reflective state pattern. *Proceedings of the Pattern Languages of Program Design*, 1998; 1–18.
35. Henney K. Collections for states. *Proceedings of the European Conference on Pattern Languages of Programs*, 1999; 57–64.
36. Henney K. Methods for states. *Proceedings of the First Nordic Conference on Pattern Languages of Programming*, 2002; 1–13.
37. Sobajic O, Moussavi M, Far B. Extending the strategy pattern for parameterized algorithms. *Proceedings of the 17th Conference on Pattern Languages of Programs*, 2010; 1–11.
38. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, 2012, doi:10.1007/978-3-642-29044-2.
39. Jedlitschka A, Ciolkowski M, Pfahl D. Reporting experiments in software engineering. *Guide to Advanced Empirical Software Engineering*. Springer London, 2008; 201–228, doi:10.1007/978-1-84800-044-5 8.
40. Basili VR, Caldiera G, Rombach HD. Goal question metric paradigm. *Encyclopedia of Software Engineering*. Wiley & Sons, 1994; 528–532.
41. Seng O, Stammel J, Burkhart D. Search-based determination of refactorings for improving the class structure of object-oriented systems. *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, ACM, 2006; 1909–1916, doi:10.1145/1143997.1144315.
42. Aversano L, Canfora G, Cerulo L, Del Grosso C, Di Penta M. An empirical study on the evolution of design patterns. *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM, 2007; 385–394, doi:10.1145/1287624.1287680.
43. Manotas I, Pollock L, Clause J. SEEDS: a software engineer's energy-optimization decision support framework. *Proceedings of the 36th International Conference on Software Engineering*, ACM, 2014; 503–514, doi:10.1145/2568225.2568297.
44. Kniesel G, Binun A. Standing on the shoulders of giants-a data fusion approach to design pattern detection. *Proceedings of the 17th IEEE International Conference on Program Comprehension*, IEEE, 2009; 208–217, doi:10.1109/ICPC.2009.5090044.
45. Pettersson N, Lowe W, Nivre J. Evaluation of accuracy in design pattern occurrence detection. *IEEE Transactions on Software Engineering* 2010; **36**(4):575–590, doi:10.1109/TSE.2009.92.

46. Balmas F, Bergel A, Denier S, Ducasse S, Laval J, Mordal-Manet K, Abdeen H, Bellingard F. SQualE - software metric for Java and C++ practices. *Technical Report*, INRIA 2010.
47. Noureddine A, Rouvoy R, Seinturier L. A review of energy measurement approaches. *ACM SIGOPS Operating Systems Review* 2013; **47**(3):42–49, doi:10.1145/2553070.2553077.
48. Diouri MEM, Dolz MF, Glück O, Lefèvre L, Alonso P, Catalán S, Mayo R, Quintana-Ortí ES. Assessing power monitoring approaches for energy and power analysis of computers. *Sustainable Computing: Informatics and Systems* 2014; **4**(2):68–82, doi:10.1016/j.suscom.2014.03.006.
49. Chen H, Li Y, Shi W. Fine-grained power management using process-level profiling. *Sustainable Computing: Informatics and Systems* 2012; **2**(1):33–42, doi:10.1016/j.suscom.2012.01.002.
50. Noureddine A, Rouvoy R, Seinturier L. Unit testing of energy consumption of software libraries. *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ACM, 2014; 1200–1205, doi:10.1145/2554850.2554932.
51. Do T, Rawshdeh S, Shi W. pTop: a process-level power profiling tool. *Proceedings of the 2nd Workshop on Power Aware Computing and Systems*, 2009; 1–5.
52. Field A. *Discovering Statistics Using SPSS*. 3rd edn., SAGE Publications Ltd, 2009.
53. Hastie T, Tibshirani R, Friedman J. *The Elements of Statistical Learning*. Springer Series in Statistics, Springer New York, 2009, doi:10.1007/978-0-387-84858-7.
54. Ampatzoglou A, Kritikos A, Arvanitou EM, Gortzis A, Chatziasimidis F, Stamelos I. An empirical investigation on the impact of design pattern application on computer game defects. *Proceedings of the 15th International Academic MindTrek Conference on Envisioning Future Media Environments*, ACM, 2011; 214–221, doi:10.1145/2181037.2181074.