International Journal of Engineering & Technology, 7 (4.31) (2018) 511-515

SPC

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

An Automated Code Optimizer of Design Patterns for Reducing Energy Usage in Green Computing

Jamilah Din*, Ooi Chiew Wei and Muhammed Basheer Jasser

Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, 43400 Serdang, Selangor, Malaysia
**Corresponding author E-mail: jamilahd@upm.edu.my*

Abstract

With the trend towards green computing, researchers are looking for more energy efficient software solutions. One of the approaches is optimizing the software design patterns. Studies have shown that adopting certain design patterns (especially Decorator, Observer) leads to a higher energy consumption. Fortunately, there are studies for proposing optimization rules to reduce the energy consumption of such patterns. However, due to the lack of guidance, designers do not adopt these optimization rules. This research aims to identify the factors that cause energy inefficiency during the implementation of software design patterns and to provide a tool to automate the optimization of software design patterns. Based on the experimental results, the adopted automated optimization rules in the proposed tool reduce the energy consumption of the design pattern of interest meeting the research objective in automating the design pattern optimization, which makes it easier to adopt design patterns in green computing.

Keywords: *Green Computing; Code Optimization; Design Patterns; Energy Consumption.*

1. Introduction

In recent years, green computing is getting popular. Green compting is an environmental friendly computing, in the sense that the practice consumes less resources or causes less damage to the environment [9]. As new technologies are introduced, they do not only bring in improvements and conveniences but also damages and cost to the environment. For example, current famous cloud computing environments are expected to consume 47 billion kWh by 2019, which stands for 4.7 billion dollars and 50 million MT carbon dioxide [18].

New non-functional requirements have been introduced to define greenness in computing, since existing attributes are not meant for green computing. In [12, 16], Sustainability has been defined as the new non-functional requirement that represents greenness. It means that the software system or service shall contribute to preserving environment and human well-being. It includes resource consumption, greenhouse gas emission, social sustainability, and recycling. Some other terms are introduced as well. For example, Procaccianti et. al. [14] claims that energy hotspots are elements or properties at any level of abstraction of the system architecture, which has a measurable and significant impact on energy consumption.

There are two general ways to categorize green computing. The first way is green in computing and the second way is green by computing. Existing research studies have been done on architectures [1, 3, 7, 10], while some other studies focus on the hardware issues [11]. In terms of green in computing, many works focus on the design phase [10].

Numerous research works have been done to bring greenness into software engineering. However, it is not easy to practice green computing [15]. Many of the research works require manual implementation. There is a need to bring these works to practitioners. It is hard for practitioners, more specifically the software designers, to adopt "green" design. Thus, there is a need to aid designers through automating the transformation processes of software designs into greener ones.

There are many research studies done on the effect of design pattern from the green perspective [4, 8, 2, 5, 22]. Noureddine and Rajan [5] proposed transformation rules that can be applied onto the design pattern so that energy consumption could be reduced. However, applying these rules is difficult if the approach requires manual transformation. Designers need to learn about the transformation rules before performing them manually [20]. This extra works leads to unwillingness to adopt green computing. Thus, there is a need to improve green computing adoption.

The main aim of this research is to enhance green computing adoption by providing a tool that helps to transform selected design pattern to an optimized (in term of green) design pattern. Our work automates the transformation rules as an alternative way to the manual transformation. Our work considers the design of the software and then translates it to code. Then, transformation rules are implemented on the code. Three main objectives of this work are as follows:

- Identify the energy consumption of design patterns
- Identify existing transformation rules.
- Develop a tool to automate the transformation rules

The paper is structured as follows. Section 2 presents the background and literature review including the studies on energy consumption of design patterns, the causing factors for that, some optimized design patterns and some tools for development assistance. Section 3 introduces the design and implementation of the targeted design pattern and its optimized one. Section 4 presents the tool and its evaluation. Section 4.2 discusses the results. Section 5 concludes the work.

2. Background and Literature Review

Design patterns have been used to optimize the system performance and promote re-usability of the system design. Research works in [2, 4, 5, 8, 22] have been carried out to investigate effect of adopting design pattern on energy consumption. The results show that different design patterns affect the energy consumption differently. Some design patterns reduce the energy consumption, while some other design patterns increase the energy consumption. This section is divided into four subsections. Section 2.1 discusses the existing studies regarding energy consumption of design patterns. Section 2.2 reviews the factors that cause energy consumption in design patterns. Section 2.3 exhibits existing work on alternative design patterns. Section 2.4 presents some tools that assist developers in adopting green computing

2.1. Energy Consumption of Design Pattern

Litke, A. et. al. [4] have selected three design pattern from the categories (creational, behavioral, and structural). The authors found out that the observer pattern increases the power consumption dramatically.
Sahin et. al. [8] investigates the power profiles of software application using design patterns and without using design patterns. In this research, the authors selected fifteen design patterns as subjects of research, five from each category (creational, structural, and behavioral) proposed by Gamma et. al. They found out that design patterns have different power profile regardless of their categories.
Noureddine and Rajan [5] have proposed a compiler optimization approach to reduce energy consumption without sacrificing design patterns in software engineering. The authors aim to show that it is possible to further reduce energy consumption of design pattern implementation. In this research, they have studied fourteen patterns (Mediator, Observer, Strategy, Template, Visitor, Abstract, Builder, Factory, Prototype, Singleton, Bridge, Decorator, Flyweight and Proxy) written in C++ and other seven patterns (Chain, Command, Interpreter, Iterator, State, Adapter and Composite) in Java regarding the energy consumption of design pattern code against non-design pattern code. The authors have compared the source code with design pattern and source code without design pattern in their experiment. The effect of adopting design pattern in source code varies. While some design patterns reduce the energy consumption, some others increase the energy consumption. The results show that the Mediator design pattern, Observer design pattern, and deco- rator design pattern have the highest increase in energy consumption.
Bunse et. al. [2] have conducted their research in mobile platform. They have used six design patterns (Facade, Abstract factory, Observer, Decorator, Prototype) and Template method in Java language in the experiment. The experiment subjects are separated into two categories, with design pattern and without design pattern. Similar to Noureddine's experiment, Bunse compares the energy consumption and execution time of Java application with design patterns and Java application without design pattern. The results show that Prototype and Decorator topped the lists with the highest energy consumption and execution time difference, while other design patterns (Facade, Abstract Factory, Observer) and Template Method show only a little difference. In this experiment, they have used PowerTutor App to measure the energy consumption.
Feitosa et. al. [22] investigates the effect of design patterns in more sophisticated systems, which contain the investigated design pattern and additional functions. In [22], three design patterns have been selected: Strategy, State, and Template patterns and their alternatives. The alternatives are codes that provide the same functionality but have different implementation. The results show that three of the selected design pattern increase energy consumption. In their work, they use the following software to measure the energy consumption: PowerAPI, pTop and Jalen. PowerAPI and pTop are used to measure the energy consumption at the operating system level, while Jalen is used to measure the energy consumption at the method level.

2.2. Causing Factors of the Increase of Energy Consumption

The works in [2, 4, 5, 8, 22] further infer the underlying factors that cause the increase of the energy consumption. Result are displayed in Table 1. Table 2 shows the factors caused by design pattern individually and the alternatives used to do the comparison.
In Litke's research [4], Observer pattern introduces additional classes and methods. This leads to a longer code, more method calls, and more number of memory access, as compared to a non-patterned subject. Thus, the authors infer that these are the reasons behind the increase of energy consumption.
Meanwhile, Sahin et. al. [8] stated in their research that the general factors that cause the increase in energy is due to more object instantiation, method calls and number of parameter passing. However, the authors believe that what causes the 700% increase in energy usage of Decorator pattern is due to creating complex objects without inheritance.
Bunse et. al. [2] and Noureddine and Rajan [5] both agree that more object instantiations and method calls cause the hike in energy consumption.
Lastly, Feitosa et. al. [22] have investigated the energy consumption of design patterns with two criteria which are line of code and number of method invocations/calls [22]. The authors believe that these are primary factors that cause the energy inefficiency. Based on their experiments, they notice that design patterns are not beneficial for less complex design problems. In addition, they notice that when dealing with complex situations, the effect of polymorphism weakens.

2.3. Design Pattern Alternatives Based on Energy Consumption

This section focuses on the optimization of design patterns in term of energy consumption. From the literature review, there are two research studies [5, 22] in which optimization rules are proposed.
Noureddine and Rajan [5] in their work apply modifications on specific design patterns. In their work, decorator transformation is done through replacing multiple object construction (an object and its decorators) with a single object creation. As for observer, optimization is done through transforming multiple function calls and memory accesses into a single call.
In [22], the alternative design patterns are selected from popular research studies. There are two alternatives design pattern in [22]: State/Strategy and Template. State/Strategy's alternative adopts replacing polymorphism with the use of conditional statement. This alternative omits some cases like abstract class or interfaces. As for the Template design pattern, the authors focus on reducing method calls.

2.4. Existing Works on Assisting Developers in Adopting Green Computing

There are not many models or tools that are available to support developers in adopting greenness.
Gamez et. al. [13, 19] focus on addressing energy consumption at the architectural level. The authors propose HADAS that can identify the energy consumption at different hotspot, store, communication, compression, security, data access, notification, synchronization, user interface, code migration, and fault tolerance. HADAS displays the expected energy consumption to the user. HADAS automates the green configuration chosen by the user. The optimization requires the predefined optimization rules, without which the optimization cannot be performed.

Table 1: Factors Causing Higher Energy Consumption

Reason causing more energy consumption	[4]	[8]	[2]	[5]	[22]
Object Instantiation		Y	Y	Y	
Method Call	Y	Y	Y	Y	Y
Code Size/ Line of Code	Y				Y
Memory Access	Y				
Message/Parameter Passing		Y			
Pattern that caused more energy consumption	Observer	Decorator, Observer	Decorator, Prototype	Decorator, Observer	Template, state/strategy

Table 2: Factors Causing Higher Energy Consumption Based on Design Pattern and Comparison with Alternatives

	[4]	[8]	[2]	[5]	[22]
Factor based on pattern	Observer: Code size increase, more method call, more frequent memory access	Decorator: More object instantiation, Complex object creation without inheritance	Decorator: More object instantiation, more method call	Decorator: More object instantiation, more method calls	Strategy/State: More object instantiation, Code size increase
		Abstract Factory: More object instantiation, more message/parameter passing	Prototype: More object instantiation, more method call	Observer: More function calls	Template: More object instantiation, Code size increase
Alternatives	-	-	-	Reducing object creation and function calls	State/Strategy: replace the use of polymorphism with the use of conditional statements (situational)
Difference in energy consumption	Observer>Non-pattern ~40%	Decorator>Non-pattern 712%	Decorator>Non-pattern 132%	Decorator>alternative 12.23%	Template> alternative 17.4%(PowerAPI) 24.34%(Jalen)
		Observer>Non-pattern 62.2%	Prototype>Non-pattern 33.2%	Observer>alternative 6.95%	SS>Alternative 53.68%(PowerAPI) 55.51%(Jalen)

Manotas et. al. [21] investigate how software practitioners think of energy in software development processes. Manotas et. al. conclude that software practitioners do care about green computing; however, they do not really practice it because there are no guidelines or assistance to practice green computing.

3. Design and Implementation

This section discusses the targeted design pattern, the concept behind the modification onto it and how it is implemented as a prototype. This section is divided into two subsections. Section 3.1 presents the Decorator design pattern. Section 3.2 presents the optimized Decorator design pattern.

For the Decorator pattern, Noureddine's optimization rules [5] have reduced the object instantiation by placing all the methods of multiple objects into one single object. While the authors in [5] do this manually converting the source code, this research applies the optimization in a different way. The proposed tool uses the similar optimization concept, which is reducing object instantiation by putting all the methods in multiple objects into one object. In this case, the tool implements the optimization rules automatically onto the source code.

The proposed automation tool takes the source code as input and changes the source code to fit the optimization. In this case, the automation relies on the naming convention of the method inside the class file.

3.1. Decorator Design Pattern

Decorator design pattern is mainly used for designing a flexible and reusable object-oriented software. The design pattern allows behaviors to be added to an individual object, without affecting the behaviors of other objects from the same class [23].

A basic Decorator design pattern class diagram is shown in Figure 1. In Decorator design pattern, the *Window* is" decorated" through multiple object instantiation. *HorizontalScrollBarDecorator* and *VerticalScrollBarDecorator* contain additional functions that are to be added or decorated onto the *SimpleWindow*.

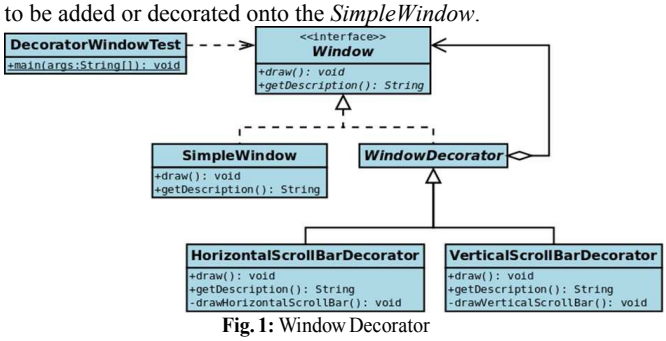

Fig. 1: Window Decorator

3.2. Optimized Decorator Design Pattern

The goal is to reduce the object instantiation. While there are many ways to modify the code, this research follows the optimization rules proposed by Noureddine and Rajan [5]. The employed concept is creating a new abstract class and moving existing functions from all the other decorator classes into this new abstract class. The alternative for the *Windows Decorator* is shown in Figure 2.

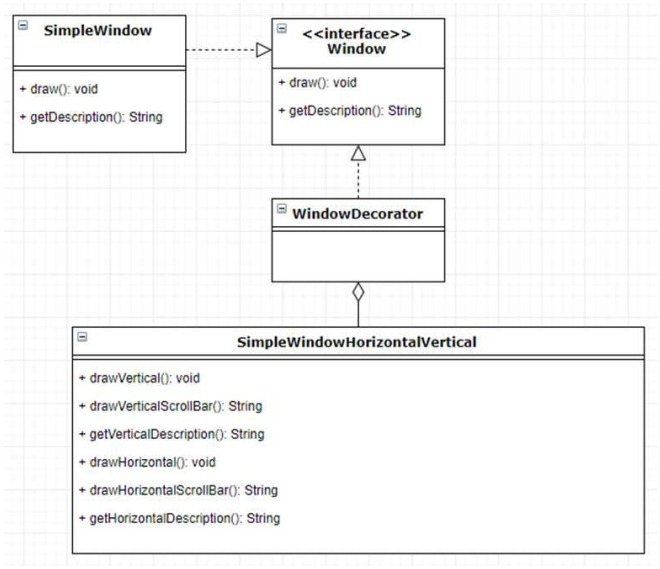

Fig. 2: Alternative Design Pattern for Window Decorator Design Pattern

This alternative design pattern reduces multiple object instantiation to one simple object instantiation. Consider the number of objects in original source code as N*original* the optimized code will have N*original* -1 less objects, as compared to original source code. The final form of the optimized design pattern has less object instantiation while having similar number of function calls. This allows the number of function calls to be a constant in the experiment.

4. The Tool Evaluation and Results

This section discusses how the experiment is carried out in this study and the evaluation of the data collected from the experiment. This section is divided into two subsections. Section 4.1 presents setup of experiment. Section 4.2 presents the data evaluation and discusses the results.

4.1. The Proposed Tool and Experiment

The experiment is done on a conventional computer. The specification is shown on Table 3. The objective of the experiment is to compare the original source code and optimized source code, and to measure which one consumes more energy during execution time. In this case, the test subjects (source codes) and energy measuring tools (PowerAPI) are tested on the same computer, which means the data collection and analysis for this experiment are not affected by the computer specification.

Table 3: Computer Specification Used

CPU	Intel®,Pentium® CPU G620 @ 2.60GHz
Memory	1.8gb
Operating,System	LinuxMint,18.3 cinnamon

There are many power measuring tools used in the literature, such as pTop, Jalen, PowerAPI and Intel Power Gadget. However, many of these tools are no longer supported by the development team. Due to this difficulty, this experiment uses PowerAPI which still have a good support. On top of that, PowerAPI allows user to selectively measure the power consumption of application, which definitely fits the purpose of this experiment and allows to identify the power consumption of a single application at one time.

Subjects of experiment are the source code with the decorator design pattern and also optimized source code with the optimized design pattern. The Decorator source code is a general source code collected from an online source [23]. As for the optimized source code, it is generated with the help of the proposed tool.

The flowchart of the conducted experiment is shown in Figure 3. The main objective is to identify the energy consumption of two test subjects and compare both readings. The first step is to get the test subject for the experiment. Next, is to setup the testing environment. Due to the constraint posted by PowerAPI, which requires Linux environment, a virtual machine is installed on the computer to host the operating system. Afterwards, the power monitoring tool is tested.

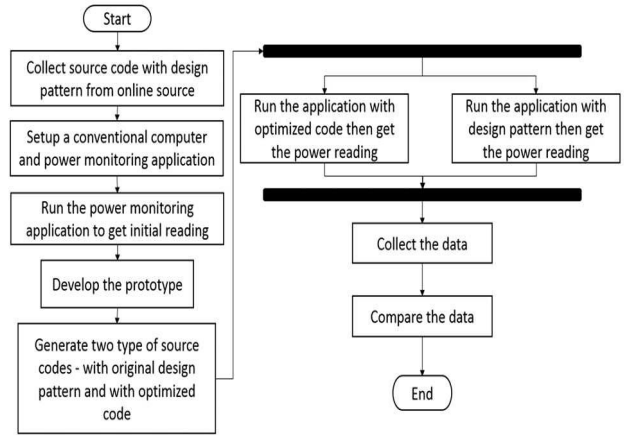

Fig. 3: Flow of Experiment

The prototype of the proposed tool is developed in Java programming language, and it is used to generate the optimized source code. Figure 4 shows the flowchart of the prototype. This early prototype only provides the optimized design pattern structure to the designer, so that designer still need to apply changes to the function naming and calling part in the source code in order to make it executable.

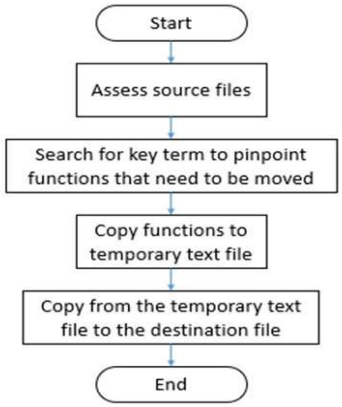

Fig. 4: Flow Chart of Prototype

The original and optimized source codes are then tested for the energy consumption during execution and data is collected. Lastly comparison is made and results are discussed.

4.2. Results and Discussion

Five readings have been obtained during the experiment. Figure 5 and Table 4 clearly show that the energy consumption of original source code (Eoriginal) is more than the energy consumption of optimized source code (Eoptimized). Although the reading of Eoriginal varies, the average reading is higher than Eoptimized, which is 193% higher. This shows that the proposed tool does help to create a less energy consuming application.

Table 4: Experimental Results

	Original Source Code	Optimized Source Code
Reading,1	8272.7272mW	4550mW
Reading,2	8272.7272mW	4550mW
Reading,3	4550mW	4550mW
Reading,4	13650mW	4550mW
Reading,5	9100mW	4550mW

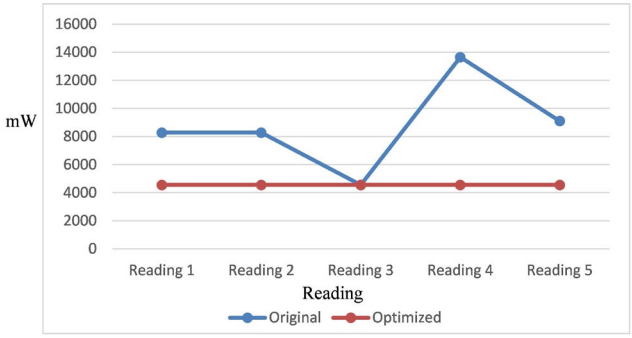

Fig. 5: Experimental Results

Internal Validity: One of the biggest concerns is the sensitivity of the tool. Due to the scale of the test subjects, the execution time is very fast. It is hard for the tool to get the reading for every execution of lines of code. Furthermore, the tool can only measure in miliWatt (mW), and any smaller scale is not captured.
However, in this experiment, the need is to show that optimized code consumes less energy than the original code. For that, this issue can be neglected.
External Validity: Different environments might gain different results. This is because different types of processor consume different amount of energy during runtime. However, this shall not affect the result where optimized source code consumes less energy than the original source code.

5. Conclusion

More energy efficient software solutions are required in the context of green computing. Optimizing the software design patterns is a way of achieving this in software that employ design patterns, especially those which increase the energy consumption in the software that employ them. Studies have shown that adopting certain design patterns (especially Decorator, Observer) leads to a higher energy consumption. Optimizing rules to reduce the energy consumption of such patterns have been introduced. However, these optimization rules still lack the necessary guidance and automation making these rules not being adopted by designers.
This research aims to identify the factors that cause energy inefficiency during the implementation of software design patterns and to provide a tool to automate the optimization of software design patterns.
Based on the experimental results, the adopted automated optimization rules reduce the energy consumption of Decorator design pattern meeting the research objectives.
While full automation is yet to be achieved, further works could be done in order to achieve that. In addition, there are other types of design patterns that require attention in order to be greener. Also, there are more to be discovered in green computing in the aspect of design patterns. Future works can be done with bigger size of test subjects and systems.

Acknowledgement

Thank you to the Ministry of Higher Education (MOHE) and Research Management Center, Universiti Putra Malaysia (UPM) for the financial supports through FRGS Vote No: 08-01-15-1726FR

References

[1] Beik, R.: Green cloud computing: An energy-aware layer in software architecture. In Engineering and Technology (S-CET), Spring Congress on (pp. 1-4). IEEE, (2012).
[2] Bunse, C., Schwedenschanze, Z., and Stiemer, S.:On the energy con- sumption of design patterns. In Proceedings of the 2nd Workshop EASED@ BUIS Energy Aware Software-Engineering and Development (pp. 7-8), (2013).
[3] Jagroep, E., van der Werf, J. M., Brinkkemper, S., Blom, L., and van Vliet, R.: Extending software architecture views with an energy con- sumption perspective. Computing, 1-21, (2016).
[4] Litke, A., Zotos, K., Chatzigeorgiou, A., and Stephanides, G.: Energy consumption analysis of design patterns. In Proceedings of the Interna- tional Conference on Machine Learning and Software Engineering (pp. 86-90), (2005).
[5] Noureddine, A., and Rajan, A.: Optimizing energy consumption of design patterns. In Proceedings of the 37th International Conference on Software Engineering-Volume 2 (pp. 623-626). IEEE Press, (2015).
[6] Ramirez, R. I., Rubio, E. H., Viveros, A. M., & Herna´ndez, I. M. T.: Differences of energetic consumption between Java and JNI Android apps. In Integrated Circuits (ISIC), 2014 14th International Symposium on (pp. 348-351). IEEE, (2014).
[7] Rangaraj, G., and Bahsoon, R.: Green software architectures: A market-based approach. In The Second International Workshop on Software Research and Climate Change (WSRCC), (2010).
[8] Sahin, C., Cayci, F., Gutie´rrez, I. L. M., Clause, J., Kiamilev, F., Pollock, L., and Winbladh, K.: Initial explorations on design pattern energy usage. In Green and Sustainable Software (GREENS), 2012 First International Workshop on (pp. 55-61). IEEE, (2012).
[9] Wang, D.: Meeting green computing challenges. In Electronics Pack- aging Technology Conference, 2008. EPTC 2008. 10th (pp. 121-126). IEEE, (2008).
[10] Williams, J., and Curtis, L.: Green: The new computing coat of arms? IT Professional Magazine, 10(1), 12, (2008).
[11] Zhong, B., Feng, M., and Lung, C. H.: A green computing based architecture comparison and analysis. In Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing (pp. 386-391). IEEE Computer Society, (2010, December).
[12] Lago, P., Kazman, R., Meyer, N., Morisio, M., Mu¨ ller, H.A. and Paulisch, F., . Exploring initial challenges for green software engineering: summary of the first GREENS workshop, at ICSE 2012. ACM SIGSOFT Software Engineering Notes, 38(1), pp.31-33, (2013).
[13] Gamez, N., Pinto, M., and Fuentes, L.: HADAS Green Assistant: de- signing energy-efficient applications. arXiv preprint arXiv:1612.08095, (2016).
[14] Procaccianti, G., Lago, P., Vetro`, A., Ferna´ndez, D. M., and Wieringa, R.: The green lab: Experimentation in software energy efficiency. In Proceedings of the 37th International Conference on Software Engineering Volume 2 (pp. 941-942). IEEE Press, (2015).
[15] Kern, E., Dick, M., Naumann, S., Guldner, A., & Johann, T.: Green software and green software engineering–definitions, measurements, and quality aspects., 87-94, (2013).
[16] Shenoy, S. S., and Eeratta, R.: Green software development model: An approach towards sustainable software development. In India Conference (INDICON), Annual IEEE (pp. 1-6). IEEE, (2011).
[17] Stier, C., Koziolek, A., Groenda, H., and Reussner, R.: Model-Based Energy Efficiency Analysis of Software Architectures. In European Con- ference on Software Architecture (pp. 221-238). Springer International Publishing, (2015).
[18] Rubyga, G., and SathiaBhama, P. R.: A survey of computing strategies for green cloud. In Science Technology Engineering and Management (ICONSTEM), Second International Conference on (pp. 141-145). IEEE, (2016).
[19] Gamez, N., Horcas, J. M., Pinto, M., and Fuentes, L.: A green program lifecycle supporting energy-efficient applications. arXiv preprint arXiv:1612.08073, (2016).
[20] Becker, C., Betz, S., Chitchyan, R., Duboc, L., Easterbrook, S. M., Penzenstadler, B. and Venters, C. C.: Requirements: The key to sustainability. IEEE Software, 33(1), 56-65, (2016).
[21] Manotas, I., Bird, C., Zhang, R., Shepherd, D., Jaspan, C., Sadowski, C. and Clause, J.: An empirical study of practitioners' perspectives on green software engineering. In Proceedings of the 38th International Conference on Software Engineering (pp. 237-248). ACM, (2016).
[22] Feitosa, D., Alders, R., Ampatzoglou, A., Avgeriou, P., and Nakagawa, E. Y.: Investigating the effect of design patterns on energy consumption. Journal of Software: Evolution and Process, 2(29), (2017).
[23] Decorator pattern. Retrieved from https://en.wikipedia.org/wiki/Decorator_pattern, July 2018.