Applied Computer Systems
ISSN 2255-8691 (online)
ISSN 2255-8683 (print)
May 2021, vol. 26, no. 1, pp. 1–11
https://doi.org/10.2478/acss-2021-0001
https://content.sciendo.com

Evaluating the Impact of Design Pattern Usage on Energy Consumption of Applications for Mobile Platform

Awais Qasim[1*], Adeel Munawar[2], Jawad Hassan[3], Adnan Khalid[4]
[1, 4] *Department of Computer Science, Government College University, Lahore, Pakistan*
[1] *School of Science, Engineering and Environment, University of Salford, the UK*
[2, 3] *Department of Computer Science, Lahore Garrison University, Pakistan*

***Abstract* – Energy efficiency in mobile computing is really an important issue these days. Owing to the popularity and prevalence of Android operating system among the people, a great number of Android smartphone applications have been developed and proliferated by the software developers. While developing these applications, developers have to keep energy consumption factor in mind, as the efficiency of an application is largely affected by it. Thus, designers and programmers endeavour to choose the best designing approaches to develop energy-efficient applications. It is imperative to assist the programmers in choosing appropriate techniques and strategies to manage power consumption. In the present research, we have investigated the effect of Android application design on its energy utilisation. For this purpose, we have practically implemented design patterns on two Android applications and evaluated their energy consumption before and after implementing these patterns. We have modelled the high-level design of these two Android applications by using software design patterns in such a way as to abate their energy requirement. We have also checked how the quality, maintainability, and efficiency of code are affected by these design patterns. The outcomes of the research can facilitate programmers to utilise these details while developing energy efficient solutions.**

***Keywords* – Energy efficiency, green computing, green design, software design, software maintenance.**

I. Introduction

The efficiency of a mobile application is a considerable factor during the development of mobile phone applications. We do not regard a developed application as an efficient one if it just performs all of its required functions and tasks. Other nonfunctional quality attributes of software like maintenance and performance are also considered while developing mobile applications [1]. Maintenance specifically means how one can easily retest the application by understanding and modifying it. Sometimes, it is difficult to maintain the applications and, in some cases, the applications are even unmaintainable. Therefore, it is necessary to concentrate on the maintenance aspect when a software application is in the development process. Its benefit is that if we need a modification in a software application, we can achieve it at a small cost, less time and less work.

The performance of an application implies how effectively and efficiently it works and how much it satisfies its users. In the present research, we have specifically focused on application performance in terms of power management. Performance is an important aspect to concentrate on, for effective solution delivery. Poor delivery of solutions is owing to poor performance, which results in a loss of money [2]. Nowadays smart devices lack permanent power having rechargeable batteries only. The effect of our proposed solution on batteries of mobile devices is well under consideration.

We have analysed two different types of open-source Android applications, which are named AppLocker and Gergek applications. The source code of these Android applications is collected from different online repositories of GitHub. AppLocker and Gergek have 7 thousand and 8 thousand lines of code (LOC), respectively. The number of classes (NOC) in the source code of both (AppLocker and Gergek) applications are 36 and 45, which shows that the nature of the source code is very complex. AppLocker application works as a security lock of an application, in which a user can select multiple applications that are already installed in the mobile device to secure them from un-authentic access so that no one can open and steal the data of that application without the permission of the administrator. If someone tries to open it, a pop-up window appears, which takes command for authentication of administrator's rights. This application works continuously in the system background, and it maintains a list of all the selected applications, which have been marked by the device administrator in order to secure these. The role of the application is to confirm the name of the foreground running application from the list of the marked applications, which has been selected by the user. If it is present in that marked list, then it gives an authentication pop-up to authenticate the user. In Gergek, the application gives a reminder to the user when to water the house plants. There are a number of plants in a house, which are different in size. These plants need water on different

* Corresponding author's e-mail: Awais@gcu.edu.pk

©2021 Awais Qasim, Adeel Munawar, Jawad Hassan, Adnan Khalid.
This is an open access article licensed under the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0), in the manner agreed with Sciendo.

days and at different intervals of time accordingly. The user of the application selects the watering schedule for all plants according to the plant type. The user sets the period, due days and overdue days of watering for all the plants according to their types. After scheduling the plan of watering, the application continuously checks when the period becomes over and the first day of due days starts. Then it gives a reminder to the user that this specific plant needs water. If the user does not take notice of these reminders, then it starts to calculate the days, as the overdue day starts it gives warning to the user with a dying percentage of that specific plant according to the counting of due days and overdue days.

To evaluate the maintainability and performance of these Android mobile applications, we have implemented different design patterns, i.e., Observer, Singleton, Facade, Abstract Factory, and Template on them. We have used two copies of each application. One copy is used to check the implementation and impact of design patterns on this application, whereas the other copy remains without the implementation of design patterns. We are going to evaluate and compare the performance and maintainability of these two copies of applications. Model-View-Controller architectural pattern has been used to develop these applications. It is important to go through some key characteristics of software application development. The application source code ought to be simple, well-structured, lucid, proficient, testable and self-reported. Software engineers invest a large portion of time in redesigning, altering and upgrading the software applications. This requires much effort, time and cost that can be decreased simply by using better techniques during the development of software [3]. Design patterns are the generally reusable substitute of normally occurring problems. Design patterns can be utilized in software development as they are effectively created and tried solutions. Programmers implement these pre-tested design patterns easily during the development, rather than reinventing new solutions. Development of software application can get easier by utilising these design patterns. These are even proven to give more reusable and maintainable code actually. Design patterns are the 'basic language' for programmers regardless of the language they use for programming.

Developing a mobile application is a complicated task since there are numerous aspects and factors that need to be considered to achieve the specified quality attributes. Mobile devices are evolving very quickly in terms of hardware. A lot of new sensors and new technologies are launched as part of mobile devices. Various types of mobile devices are available on the market. These differ in length, width, display resolutions, operating systems, processor speed, storage capacity, and battery backup. However, one common problem in all cellular devices is the modicum battery back-up. Mobile devices possess limited computing resources and capability. Hence, complex software, which in turn consumes a large number of resources, cannot run well on cellular devices. As a result, it needs exigency of time to develop such kind of applications, which abate the energy consumption problem as well as give high performance. Our research contribution consists of an evaluation of the design pattern impacts on application performance, i.e., energy consumption and code maintenance.

Based on the problem statement, below are some research questions which we are going to investigate in our research.

- What impact the Design Patterns have on energy consumption/performance of Android applications?
- Can we attain the desired maintainability of applications by the implementation of design patterns?
- How much design patterns would be effective in Android application development?
- Can we ignore the maintenance of application over the performance of the application?

To perform our analysis, we have used two different open-source Android applications and applied some design patterns from Gang of Four [4]. We have chosen five most commonly used design patterns based on our personal experience, i.e., Facade, Observer, Abstract Factory, Singleton, and Template pattern. This way we are able to cover all the three classifications of design patterns, i.e., Structural design patterns, Creational design patterns, and Behavioural design patterns. Structural design patterns simplify the structure by identifying the relationships and deal with how classes and objects can be composed, to form larger structures. Our chosen Facade pattern belongs to this category. Creational design patterns discourage hard-coded solution and provide a mechanism to make decision at the time of instantiation of a class (i.e., creating an object of a class) providing a more general and flexible approach. Our chosen Abstract Factory pattern, Singleton pattern belong to this category. Behavioural design patterns provide a mechanism that objects are loosely coupled and are still able to communicate with each other providing an efficient way of interaction and responsibility of objects. Our chosen Observer Pattern, Template pattern belong to this category. The architectural style followed in both Android applications is Model-View-Controller (MVC). The tools used to apply these design patterns and bring changes in these open-source Android applications are android studio and eclipse IDE. The goal of the research is to analyse and verify the impacts of design patterns on Android applications. For this purpose, we have conducted an empirical investigation to examine application maintenance and performance. Maintenance has been measured by quality metrics plug in such as Metrics Reloaded [5]. Performance in terms of energy consumption has been monitored using sophisticated Android application, Power Tutor-App and Power Tutor Pro-App, developed at the University of Michigan. Power Tutor is a power estimating mobile application that informs smartphone users and developers of the energy consumed by different apps. The billed power model in Power Tutor includes six elements: CPU and LCD in addition to GPS, Wi-Fi, audio, and cellular interfaces. For 10-second intervals, it is accurate up to 0.8 % typically with at most 2.5 % error [8]. The performance of AppLocker and Gergek, before and after implementing design patterns, has been measured by using the above-mentioned power monitoring applications. This has been done on a handheld Android device for a fixed interval of time. The

device uses a normal battery for its power. Other hardware specifications of this device are given in Table II.

II. The Proposed Evaluation Strategy and Tool Usage

We have relied on quality metrics plug in for maintenance factor, i.e., Metrics Reloaded [5] for Lines of codes, Cyclomatic complexity and the number of classes. For performance measurement, we have monitored the result of design pattern implemented application after its execution time. We finally affirm that the outcomes produced by these design pattern implemented Android applications are positive, after analysing the results of power profiler applications and quality metrics in terms of Cyclomatic complexity, Lines of codes, object creation and the number of classes. Code quality gets better simply by the use of design patterns. When we evaluate the performance of every single design pattern implemented applications against those which are not using design patterns, we notice that some design patterns produce a negative impact on performance by increasing the energy consumption and some of these design patterns produce a positive impact on the performance of application by curtailing the energy consumption and CPU usage. From the obtained results generated by quality metrics and power profiler applications, we finally maintain that the use of design pattern improves the code quality. We suggest that programmers should be aware of design pattern advantages and disadvantages so that they can use/implement the appropriate design pattern in software development to produce good quality software or application in terms of performance and maintenance. We additionally recommend that a poorly composed code may also be rewritten simply by using design patterns. In order to prevent performance and maintenance issues, we can use design patterns with caution.

Power Tutor is an application for Google Android phones, developed at Michigan State University. It displays the energy utilised by major system parts such as CPU, network interface, screen, GPS navigation receiver, and various running applications. This application enables software programmers to visualise the effect of design adjustments on power consumption. Application end users may also make use of it to regulate how their activities affect the battery life cycle. Power Tutor runs on the energy usage model designed by direct measurements within the careful control of device power administration states. This model generally gives power usage estimates within 5 % of actual cost. A configurable screen for power consumption background is provided. It offers users a text-file based result containing detailed outcomes. Energy utilisation of any application can be checked by Power Tutor application.

III. Preliminaries

In this section, we provide a summarised overview of the structural design patterns that we have used for the analysis of energy consumption. These patterns are useful for building the structure and therefore enhance the maintenance of the system.

A. Facade Pattern

The Facade design pattern provides a standard single interface to multiple interfaces in a subsystem. The interface created by the Facade pattern is a very advanced interface, which makes the subsystems easier to use. The Facade pattern reduces the overall complexity of a system, simply by encapsulating the subsystem with a simple facade class and by decoupling the client from the subsystem. Now if a client wants to connect to the system, then it only communicates with the system through the Facade interface. Without Facade, a client has to be interacting at many places for communication purpose with the subsystem. It separates the subsystem from the client; therefore, the complexity of the subsystem is reduced due to the decoupling. A standard working of the Facade pattern is presented in Fig. 1. A client connects to the subsystem with the help of Facade interface. Subsystem contains methods, which perform some functionality of the application, and clients use Facade interface class to connect with these subsystems.

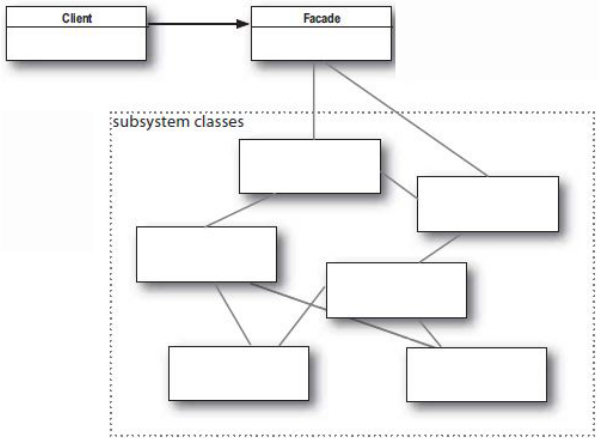

Fig. 1. Facade pattern UML diagram [4].

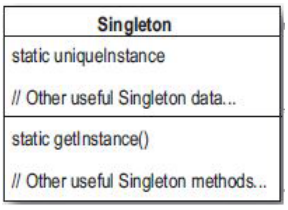

Fig. 2. Facade pattern UML diagram [4].

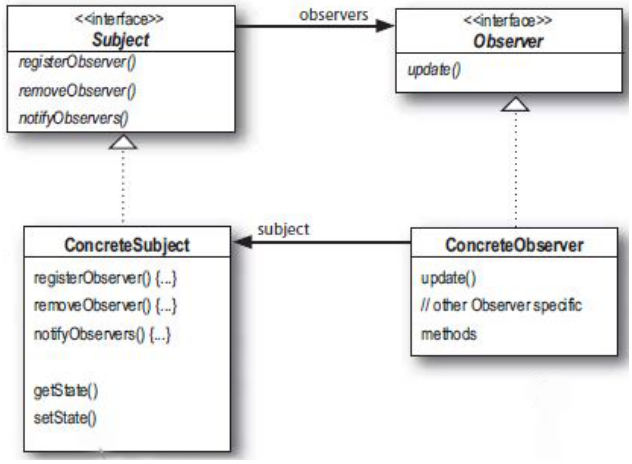

Fig. 3. Observer pattern UML diagram [4].

B. Singleton Pattern

Singleton design pattern makes sure that a class does not have more than one instance and provides a single point of access to it at every point in the source code. It is utilised as a language approach in object building. Usually, an object requires being instantiated a lot of times in the application lifecycle. In an application, the same type of object is created multiple times, which is a resource demanding procedure. Singleton makes sure that only a single object instance is created in the application lifecycle and gives a way to access it. When someone wants the object of the singleton class, it is created for the first time and then the created object is used multiple times in the application. The overall working of Singleton design is presented in UML form in Fig. 2.

C. Observer Pattern

In some situations, it is important to inform a lot of objects regarding some situation within the application. By using the Observer pattern, we can handle these types of situations, in which it is important to transmit notices regarding a specific thing to the several object instances in the application software. Observer pattern describes one-to-many relationships among several objects to ensure that if the state of an object is changed, then all of the objects which are dependent on it change themselves instantly. The standard UML diagram of the Observer design pattern is presented in Fig. 3. The most important objects employed in the Observer pattern are observer and subject. Subject contains a group of observers. All the groups of observers which are in a subject are notified when the state of the subject is changed. The observer is coordinated with the subject state.

D. Abstract Factory Pattern

The Abstract Factory design pattern is the Creational design pattern. It gives a framework, which enables everyone to create objects that adopt an overall pattern. Therefore, at execution time, the Abstract Factory is certainly combined with any preferred concrete factory that may produce items of a preferred type. UML representation of the Abstract Factory design pattern is presented in Fig. 4. Abstract Factory gives interfaces for creating groups of identical or reliant objects without mentioning their concrete implementation detail. Client software makes a concrete implementation of the Abstract Factory and then utilizes the general interfaces to produce the concrete objects that are composed of the group of objects. The clients do not have any knowledge that they get which object from concrete factories as it only utilises the general interface of the product.

E. Template Pattern

In some cases, we need to fix the sequence of procedures that a function uses, but let subclasses give their own personal implementations by applying the same sequence of procedures. This Template pattern gives a sequence of procedures in a function, and defines the implementation of this function in subclasses. Representation of the UML diagram of the Template pattern is given in Fig. 5. In the Template pattern, there is an abstract class which contains the Template function. Template function should contain specific actions whose sequence is usually fixed and for a few of the functions, the implementation of the Template method is different from the bottom class to top class. In the Template method technique, the template function should be declared final to fix the sequence of actions. The majority of the time, subclasses call functions from superclass; however, in Template design pattern, superclass calls the template function from subclasses. Functions in the base class with standard implementation are known as Hooks, plus they are meant to be overridden by subclasses.

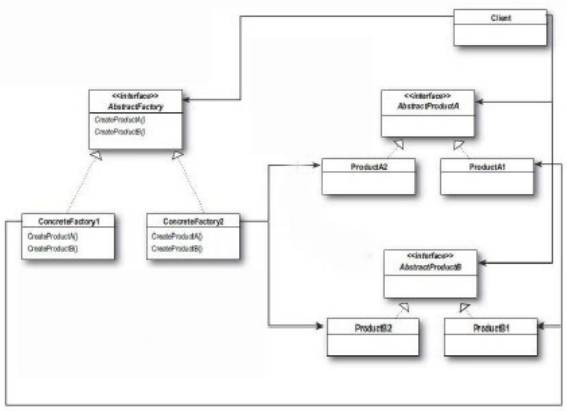

Fig. 4. Abstract Factory pattern UML diagram [4].

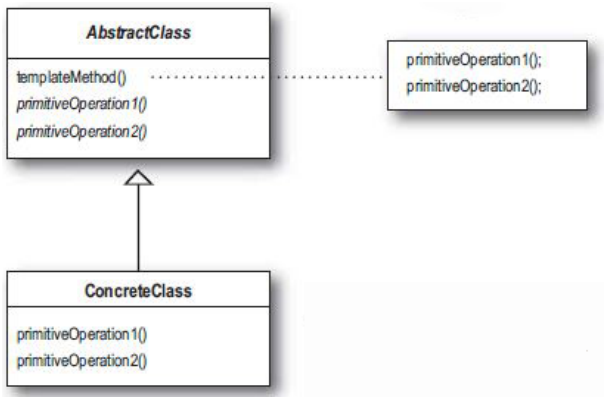

Fig. 5. Template Pattern UML diagram [4].

IV. Related Work

The conservation of energy by making energy saving mobile applications is a hot area of research. Many energy conscious techniques make an effort to reduce energy consumption by replacing hardware components [6], and some other techniques use the handling of energy requirements. It is described in [7] that a basic replacement of the resources between CPU and memory space helps decrease the quantity of energy needed. The authors in [8] say that battery operated mobiles and inlay system devices highly depend upon the energy usage of the involved components. They have also pointed out the fact that processing less information requires less energy, and also presented a setup for measuring the energy requirements of the core (CPU) and memory space of a micro-controller structured system by working on sorting algorithms. Energy statistics for software application depend on hardware based or software based methods [9]. An approach to producing energy source models for mobile devices utilising the smart battery software accompanied by methods to achieve reliability has been presented in [10]. A power producing scheme with execution to enable fine-grained energy processing has been presented in [11]. The effect of applying design patterns onto performance is investigated in [12], also a procedure for choosing design patterns is provided. A strategy for mapping software design to power utilisation is provided in [13], as well as the results of using this strategy on diverse software implementations are presented.

Power analysis of embedded software is given in [14], according to this research, it is the software that plays a key role in power utilisation. In [15], it is identified that energy can be reduced to 40 % just by rewriting the code on an Intel-based system. It is also identified that the CPU and memory system energy can be reduced by code compilation. In [16], it is expressed that a system is composed of both software and hardware components where software controls the hardware. Thus, steps taken during program design have a critical effect on the energy usage of the processor. The same concept is given in [17], which states that by changing software design and quality using software engineering techniques we can optimise energy consumption. The researchers in [18] are of the view that it is important to provide a correct set of tools to programmers and designers so that they are able to make better the process of energy conscious programming. SEEP presents a programming framework that helps programmers and designers in energy conscious programming [19]. The research conducted in [20] suggests that for developing high-standard and high-quality software, it is important to create a test case scenario in order to compare different version or release for energy consumption. The factors like warming, greenhouse gas footprint or energy usage have to be viewed as for the durability of the process of software production. In [21], the authors have presented standard techniques for the measurement of energy consumption of application running on mobile devices. These standard techniques include a set of requisites, an implementation and an API, which give three totally different measurement methods resulting in accuracy. In [22], the authors have given energy profiler; the first energy profiler for mobile phone applications examines the profiling of applications running on smartphones. Energy profiler finds that 65 % to 70 % energy of applications is wasted on a third-party advertising campaign. They also have found multiple bugs of “wakelock”, which are part of the bugs belonging to “energy” in smartphone applications. They have also realised the appropriate position of these bugs in the source code of applications. In [23], a strategy is given by which an application can find its energy consumption according to the pairing of energy-related software and hardware constraints. Also in [24], it is argued that the design process of the embedded system can be helpful in estimating the energy consumption. In [25], the researchers have presented a model-based architectural approach for the analysis of energy efficiency. This model can be run with architecture models and it performs an analysis at the architecture level to calculate the power consumption of the model software system. The authors of [26] have stated that energy efficiency can conflict with the quality of design. It is imperative to consider energy efficiency when improving or developing the design of a mobile application. They have performed an analysis on different mobile applications by an anti-pattern approach. They have also found that the applications containing anti-patterns can be refracted and gave positive results. The works by [27] and [41] have revealed that any change in the system call of an application profile can affect the energy consumption of that application. The authors have also discussed a tool, which systematically checks and analyses the system call of an application and detects whether there is a change in energy consumption of the application.

The authors of [28] have told us that a number of users nowadays increasingly demand high-performance computing solutions being able to address sustainability and limit power consumption. They have also given an HDFS approach, a hybrid storage mechanism, which uses hard disk and solid-state disk combination to get better performance and save energy. The researchers in [29] have identified that the developers have less information regarding energy efficiency and lack information about the best techniques to decrease software energy utilisation, and are typically unclear about how software utilises energy. In [30], [39], [40], the authors have delineated the challenges that end users face and the greatest

responsibilities they have for their mobile phone energy utilisation. Using many situations, the writers have presented that performing an activity can use pretty much energy depending upon the user's preferences and software options. In the work presented in [31], the authors have given a demonstration and implementation of Green-Miner to measure the energy utilisation of smartphones. It simplifies the application assessment, and the results of this assessment are reported back again to experts and developers. In [32], the researchers have recommended a programming tool eLens to calculate the power utilisation of Android applications. This tool is in a position to calculate the billed power consumption of actual applications up to 10 % of ground-truth measurements.

The work of [33] has recommended a refactoring strategy to make improvements in energy utilisation of similar software systems. They have used this refactoring strategy on 15 open resource projects and experienced an energy reduction of 12 %. Moreover, [34] has explored the effect of six generally used refactoring on about 197 applications. The overall results of their research have verified that refactoring effects energy utilisation and it can either boost or reduce the quantity of energy utilised by an application. Their results have also displayed the necessity for energy-conscious refactoring techniques, which can be included in IDEs. Similarly, the researchers in [35] have studied the way in which the inline method refactoring affects the energy consumption of three embedded software applications developed in Java. The results of their research present that inline methods can maximise energy utilisation occasionally while reducing it in all others. In [36], a scientific study has been carried out concentrating on the singular and joint performance effects of three Android performance anti-patterns, on two open-source Android applications. The writers have considered the overall performance of the original and fixed programs on a prevalent user case test.

V. The Proposed Approach and Implementation Details

In the preceding section, we briefly discussed the design patterns that we are going to use in our research. In this section, we now elaborate the implementation details of these design patterns on our chosen open-source Android applications. These design patterns are Singleton, Facade, Observer, Template and Abstract Factory. We will discuss the individual and combined results of each of the design patterns after implementing them on both the AppLocker and Gergek Android applications. Application energy consumption, i.e., an indication of its efficiency, has been measured by Android power profiler applications named Power Tutor and Power Tutor Pro [37]. Both android applications (AppLocker and Gergek) have been tested on a physical Android device and the hardware details of the device are given in Table I. For measuring the maintainability metrics of both Android application source codes, we have used Metrics Reloaded and Metrics. Overall, we have used the metrics of Energy Consumption (EC), Cyclomatic Complexity (CC), Line of Code (LOC) and also the number of classes (NOC).

TABLE I
Samsung Galaxy Note 4 Hardware Details

Component Name	Related Detail
O.S	6.0.1 (Marshmallow)
Chipset	Qualcomm APQ8084 Snapdragon 805 Exynos 5433
CPU	Quad-Core 2.7 GHz Krait 450-Snapdragon 805 Octa-Core (4 × 1.3 GHz CortexA53 and 4 × 1.9 GHz Cortex-A57) – Exynos 5433
Battery	Removable Li-Ion 3220 mAH battery
RAM	3GB

Our selected Android applications have been developed in the Java programming language by following the MVC Structural pattern. We have implemented Singleton, Facade, Observer, Template and Abstract Factory design patterns on (AppLocker and Gergek) Android applications.

A. Results before the Implementation of Design Patterns

Before the implementation of any design pattern, we executed both applications for a fixed time (fifteen minutes) on the Android device and measured the energy consumption with the help of Power Tutor and Power Tutor Pro applications. We then measured the CC, NOC, and LOC as a maintainability factor of both applications. The values of positive maintainability are given below.

- The standard value of Cyclomatic Complexity given by McCabe is 10 for low-risk applications and between 11 and 30 for moderate risk applications [38].
- An increase in the Line of Code may affect the testability, understandability and reduce the maintainability of the code.
- An increase in the number of classes also increases the complexity of the classes and source code, but it decreases the dependency of classes.

Before implementing the decided design patterns, we noted the results of power consumption of applications using Power profiler application. The obtained values along with the values of maintainability metrics are given in Tables II and III. Average Cyclomatic Complexity of both applications is greater than 10, which means that the code is very complex and testability is very low. In the long run, very high cost and effort are required for the maintenance of code.

TABLE II
AppLocker Application Measurement before Design Pattern Implementation

Power Profiler	EC	LOC	Avg CC	NOC
Power Tutor	107.9	7035	33	36
Power Tutor Pro	101	7035	33	36

TABLE III
Gergek Application Measurement before Design Pattern Implementation

Power Profiler	EC	LOC	Avg CC	NOC
Power Tutor	35.4	8392	35	45
Power Tutor Pro	86.0	8392	35	45

VI. Design Pattern Implementation

In this section, we elaborate the impact of each design pattern implementation one by one on both applications.

A. Implementation of Singleton Pattern

We performed a deep analysis of the source codes of both applications and found out that there were two classes in the source code on which we could apply the Singleton pattern. Methods in these classes perform multiple tasks and save the state of application activities, boolean flags and some other temporary variable values in shared preferences. We observed that there was no need for more than one object instance initiation of these two classes for both applications. The access modifier of default constructor has been turned to private so that no other class can access it from outside the Singleton class. After the implementation of Singleton pattern and execution of both applications for 15 minutes, the results obtained are presented in Tables IV and V. From table values, we learn that after the implementation of Singleton pattern the value of energy consumption of both applications has increased. The number of classes remained consistent for both applications and there is no major increase in the lines of code.

TABLE IV
Results after Singleton Pattern Implementation on AppLocker Application

Power Profiler	EC	LOC	Avg CC	NOC
Power Tutor	118.8	7082	34	36
Power Tutor Pro	110.1	7082	34	36

TABLE V
Results after Singleton Pattern Implementation on Gergek Application

Power Profiler	EC	LOC	Avg CC	NOC
Power Tutor	84.8	8569	33	45
Power Tutor Pro	176.6	8569	33	45

However, the average CC of AppLocker application has increased but decreased for Gergek application, which means that after the implementation of Singleton pattern the overall code has become more complex in AppLocker, but it is reverse in Gergek.

B. Implementation of Observer Pattern

In Gergek application, there are three asynchronous services which are running in the background continuously to monitor a specific variation in time. These services contain extra methods that are triggered after a specific time and these methods are executed during the A-sync service execution. These methods put a huge load on the service execution cycle and we cannot write these methods outside of the service class as these are the most important part of the service. We have applied the Observer pattern on these service classes and, for this purpose, three additional classes have been created and registered as an observer in the Observer pattern. The main function of these classes is just to monitor the specific change in time when all the a-sync service classes are interested. By discontinuing the execution of these services we can allow the Observer pattern to monitor the change in time. When a change occurs, the update method of observer resumes the paused service and lets it carry out its basic tasks again. By using Observer pattern, we reduce the complexity of service class and also reduce the huge load on the service execution cycle. In AppLocker application, there is only a single background service to allow and disallow the restricted application access. This service maintains a list of applications that are restricted in the device and we have applied the Observer pattern on this service. For this purpose, we have created a class that registers itself as an observer and monitors the change in the state of the mobile device. When the device is in a locked or idle state, this observer stops the service execution and restarts it again when the device comes in the unlocked state.

Results obtained after the Observer pattern implementation, and execution of both applications for 15 minutes are presented in Tables VI and VII. From these table values, it clear that the Observer pattern has a good impact on the energy consumption of both applications as it is reduced. However, after the implementation of the design pattern in Applocker application, the number of classes has dropped. On the contrary, in Gergek application, the number of classes has increased. This reduction in classes has caused a reduction in the Cyclomatic Complexity of AppLocker application. For Gergek application, however, the Cyclomatic Complexity of the source code is not changed. LOC is decreased for AppLocker application, which is good from the maintenance point of view, but it is increased for Gergek application. The increase in LOC is due to an increase in the number of classes, which shows that in Gergek application classes are more complex compared to AppLocker.

TABLE VI
Results after Observer Pattern Implementation on AppLocker Application

Power Profiler	EC	LOC	Avg CC	NOC
Power Tutor	11.5	2011	31.5	29
Power Tutor Pro	8.4	2011	31.5	29

TABLE VII
Results after Observer Pattern Implementation on Gergek Application

Power Profiler	EC	LOC	Avg CC	NOC
Power Tutor	5	8603	35	51
Power Tutor Pro	7	8603	35	51

C. Implementation of Facade Pattern

In Gergek application, we have attempted to implement the Facade pattern on the activity class. For proper execution of services, multiple objects are created and a number of parameters are initialised before the initiation of these services. These objects and parameters play an important role in performing the tasks for which these services are initiated. Facade pattern manages the initiation of these objects and parameters, as well as manages the execution of these services without affecting the structure of an activity class. In AppLocker application, we have implemented the Facade pattern on the classes, which manage the application log event

detail and activity log event detail for tracking the proper log detail of the whole application. We created a Facade pattern interface and made both classes implement that interface. The activity class that wants to save the log event detail can use a single log event controller class object and can perform the functionality of both classes more conveniently.

Facade pattern has imparted a positive impact on both applications. Results obtained from the two power profiler applications (Power Tutor, Power Tutor Pro) after Facade pattern implementation are given in Tables VIII and IX. The results show a clear reduction in energy consumption of both applications. Average CC of AppLocker application code has also been decreased showing that after the implementation coding errors in the source code are reduced. However, for Gergek application the source code average CC remains the same. Thus, it may be predicted that Gergek source code contains more coding errors than the AppLocker source code. The NOC and LOC have increased for both applications though.

TABLE VIII
RESULTS AFTER FACADE PATTERN IMPLEMENTATION ON APPLOCKER APPLICATION

Power Profiler	EC	LOC	Avg CC	NOC
Power Tutor	83	7138	30.25	43
Power Tutor Pro	71.8	7138	30.25	43

TABLE IX
RESULTS AFTER FACADE PATTERN IMPLEMENTATION ON GERGEK APPLICATION

Power Profiler	EC	LOC	Avg CC	NOC
Power Tutor	16.3	8491	35	50
Power Tutor Pro	33	8491	35	50

D. Implementation of Template Pattern

In the source code of both AppLocker and Gergek applications, the activity classes responsible for the initiation of background services do a lot of work for service initiation. By using the Template pattern, we can overcome the complexity and responsibility of an activity class and can also reduce the load of an activity class life cycle. Since an activity class provides an interface to interact with the application, by putting a huge load on activity lifecycle, the application can go to a not-responding-state. This will result in poor usability from the user's perspective.

The results obtained after the implementation of the Template pattern are presented in Tables X and XI. After execution of the AppLocker application for 15 minutes, the results show that Template pattern has a good impact on energy consumption as it is reduced. On the contrary, in Gergek application it has increased. The NOC in AppLocker application has slightly increased showing that there is no major change in the NOC. We have obtained similar results for the Gergek application as there is no significant change in the NOC. Average CC of AppLocker application has reduced, but in Gergek app there is no change in it. However, LOC has increased in both applications.

TABLE X
RESULTS AFTER TEMPLATE PATTERN IMPLEMENTATION ON APPLOCKER APPLICATION.

Power Profiler	EC	LOC	Avg CC	NOC
Power Tutor	32.6	7128	30.25	38
Power Tutor Pro	58.1	7128	30.25	38

TABLE XI
RESULTS AFTER TEMPLATE PATTERN IMPLEMENTATION ON GERGEK APPLICATION

Power Profiler	EC	LOC	Avg CC	NOC
Power Tutor	109.6	8509	35	49
Power Tutor Pro	162.7	8503	35	49

E. Implementation of Abstract Factory Pattern

Abstract Factory pattern provides an interface, which manages the creation of appropriate objects of a factory without indicating their classes clearly. A class, which starts the background services, decides at runtime which type of service it should initiate, and what type of objects and data is required by this particular type of service for its execution. We have implemented the Abstract Factory pattern on this class; now this class only uses the appropriate object created by a particular factory for a particular service. All the decisions that are taken before the initiation of a particular service are now handled in a particular factory. The only responsibility of the service starter class is just to call an appropriate factory for service initiation and it starts the particular service without creating a different type of object and parameter initialisation.

After the Abstract Factory pattern implementation, the obtained results are provided in Tables XII and XIII. Abstract Factory pattern has produced a good impact on the energy consumption of both Android applications as it is dramatically reduced. The NOC has been increased in both applications but not that much. The Average CC has reduced in the AppLocker application, but in Gergek application there is neither positive nor negative change in it. It may be predicted that the AppLocker source code has fewer coding errors than the Gergek source code. There is a noticeable increase in LOC of both applications. Although there is a decrease in average energy consumption for both applications, but an increase in NOC and LOC makes the source code very complex, which is not good for the maintainability factor of the source code.

TABLE XII
RESULTS AFTER ABSTRACT FACTORY PATTERN IMPLEMENTATION ON APPLOCKER

Power Profiler	EC	LOC	Avg CC	NOC
Power Tutor	40.7	7275	31.5	43
Power Tutor Pro	29	7275	31.5	43

TABLE XIII
RESULTS AFTER ABSTRACT FACTORY PATTERN IMPLEMENTATION ON GERGEK APPLICATION

Power Profiler	EC	LOC	Avg CC	NOC
Power Tutor	19.1	8620	35	55
Power Tutor Pro	29.3	8620	35	55

F. Implementation of All the Combined Patterns

After implementing each design pattern separately, we have implemented multiple design patterns simultaneously on the AppLocker application to check the effect of combined patterns. This approach will help us to find if any of these patterns has a contradictory effect on energy consumption or code complexity. It is rare that in industry we use so many patterns in a single application. However, we did not find any attempt in previous work in which these patterns were all combined in a single application. This is very important for a large code base, in which multiple patterns can be implemented. We need results that show how combined pattern implementation will have a positive impact. Since a lot of effort is required to implement all of these pattern combined, there is a need to make sure that the amount of effort required is less than the benefit obtained. Therefore, we chose Facade, Template and Singleton design patterns. After the implementation of these patterns and running the application for 15 minutes, we monitored the energy consumption using power Tutor and Power Tutor Pro applications. The results of our observation are presented in Table XIV.

TABLE XIV
RESULTS AFTER COMBINED PATTERN IMPLEMENTATION ON APPLOCKER APPLICATION

Power Profiler	EC	LOC	Avg CC	NOC
Power Tutor	105.7	14441	27.56	41
Power Tutor Pro	92.8	14441	27.56	41

From these results, it is clear that after the implementation of the combined pattern on AppLocker application the energy consumption has decreased. The number of classes has increased and it is good for maintainability of the source code, as when the number of classes increases, the responsibility of all the classes decreases. Average CC of the source code has also decreased, which is also good for maintainability as a low value indicates that the source code has fewer coding errors. Noticeably, LOC has increased and it is almost double the original value. This increase in LOC can create a problem in the long run for source code maintenance.

VII. DISCUSSION AND FUTURE WORK

In the present research, we have investigated the effects of different design patterns on the energy consumption of two different android applications. We have also investigated whether the maintainability related issues are curtailed or not in the source code of these applications. From the obtained values, we see that the values of energy consumption have increased after the implementation of the Singleton pattern. We can say that the Singleton pattern puts a negative effect on the energy consumption of the application. This could be because in the Singleton pattern an object remains present in memory during the lifecycle of an application. This is opposed to lazy instantiation where an object is created on demand. In the same way, the results obtained after the template pattern implementation are not consistent. The energy consumption has increased in Gergek application, while in AppLocker app it has reduced. The increase in energy consumption for the Gergk application may be due to an increase in the number of classes. The increase in the number of classes before template pattern implementation is due to reducing the responsibilities of classes, so that the Template pattern can be implemented. Facade, Abstract Factory and Observer pattern implementation has reduced the energy consumption of both applications though. In case of Abstract Factory this is because rather than passing object references among classes, now we have a single place to get the required objects. The Observer pattern has the highest level of reduction in energy consumption. This is due to the nature of the changes made for the implementation of pattern. Now if any objects need to get notified of any event of interest there is no need to continuously enquire other objects. In case that event occurs the object is notified by the Observer. For those classes in which a lot of events related to background tasks are performed, it is specially recommended to implement Observer pattern. The use of Facade pattern has lower energy consumption because the interaction among classes is reduced significantly. Facade contains the business logic of application and becomes the center point of interaction. All the business logic related activities are now performed in Facade resulting in reduced interaction among classes. This means a fewer number of temporary objects are created that are required for pass by value.

Tables XV and XVI shows the percentage of energy saved after the implementation of these design patterns.

TABLE XV
PERCENTAGE OF ENERGY SAVED IN APPLOCKER APPLICATION

Pattern Name	Energy Consumption before Pattern	Energy Consumption after Pattern	Percentage
Observer Pattern	107	11.5	↓89.26
Facade Pattern	107	83	↓22
Singleton Pattern	107	118	↑10
Template Pattern	107	62.6	↓41
Abstract Factory Pattern	107	40.7	↓61

TABLE XVI
PERCENTAGE OF ENERGY SAVED IN GERGEK APPLICATION

Pattern Name	Energy Consumption before Pattern	Energy Consumption after Pattern	Percentage
Observer Pattern	86	7	↓91
Façade Pattern	86	33	↓61
Singleton Pattern	86	176.6	↑105
Template Pattern	86	162.7	↑89
Abstract Factory Pattern	86	29.3	↓65

Among all these three patterns, the Observer pattern has given the best results. We can clearly see a noticeable decrease in energy consumption for both applications. Between Facade and Abstract Factory pattern results, Abstract Factory has given better results than the Facade pattern.

VIII. Conclusion

To reduce the energy consumption of Android applications, we have proposed a new approach by implementing different design patterns on two Android applications. After implementation, we see that the Observer pattern implementation has saved 89.26 % energy in case of AppLocker application and 91 % energy in case of Gergek application. Facade pattern has saved 22 % and 61 % energy for the AppLocker and Gergek applications. The amount of energy required by both Android applications after Singleton pattern implementation has shown a 10 % increase in AppLocker application and 105 % increase in Gergek application, which shows that Singleton pattern has not reduced the amount of energy required by both applications. Template pattern has conserved 41 % energy of AppLocker application but in Gergek application, the Template pattern has not conserved the energy. The amount of energy conserved by the Factory pattern implementation is 61 and 65 % in AppLocker and Gergek, respectively. In our proposed approach, the maximum amount of energy has been conserved by Observer and Factory patterns, followed by the Facade pattern. Template and Singleton patterns have not saved energy, rather these two patterns have caused an increase in energy consumption.

References

[1] ISO & IEC, *Software Engineering – Product Quality: Quality Model*, 2001, vol. 1.

[2] C. U. Smith and L. G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, vol. 23. Addison-Wesley Reading, 2002.

[3] R. Leitch and E. Stroulia, "Assessing the Maintainability Benefits of Design Restructuring Using Dependency Analysis," in *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No. 03EX717)*. 2004, pp. 309–322.

[4] E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education India, 1995.

[5] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2010, pp. 105–114. https://doi.org/10.1145/1878961.1878982

[6] C. Bunse and H. Höpfner, "Resource Substitution with Components-Optimizing Energy Consumption," in *Proceedings of the Third International Conference on Software and Data Technologies*, 2008, pp. 28–35. https://doi.org/10.5220/0001879000280035

[7] C. Bunse, H. Höpfner, S. Roychoudhury, and E. Mansour, "Energy Efficient Data Sorting Using Standard Sorting Algorithms," in *Software and Data Technologies*. Springer, 2009, pp. 247–260. https://doi.org/10.1007/978-3-642-20116-5_19

[8] H. Höpfner and C. Bunse, "Energy Aware Data Management on AVR Micro Controller Based Systems," *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 3, pp. 1–8, 2010. https://doi.org/10.1145/1764810.1764820

[9] C. Bunse and H. Höpfner, "OCEMES: Measuring Overall and Component-Based Energy Demands of Mobile and Embedded Systems," in *Proceedings of the 42. Annual Conference of the German Computer Society (Gesellschaft für Informatik e.V. (GI))*, 2012.

[10] M. Dong and L. Zhong, "Self-Constructive High-Rate System Energy Modeling for Battery-Powered Mobile Systems," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, 2011, pp. 335–348. https://doi.org/10.1145/1999995.2000027

[11] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-Grained Power Modeling for Smartphones Using System Call Tracing," in *Proceedings of the Sixth Conference on Computer systems*, 2011, pp. 153–168. https://doi.org/10.1145/1966445.1966460

[12] D. Gross and E. Yu, "From Non-Functional Requirements to Design Through Patterns," *Requirements Engineering*, vol. 6, no. 1, pp. 18–36, 2001. https://doi.org/10.1007/s007660170013

[13] N. Mani, D. C. Petriu, and M. Woodside, "Towards Studying the Performance Effects of Design Patterns for Service Oriented Architecture," in *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*, 2011, pp. 499–504. https://doi.org/10.1145/1958746.1958822

[14] V. Tiwari, S. Malik, and A. Wolfe, "Power Analysis of Embedded Software: A First Step Towards Software Power Minimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 4, pp. 437–445, 1994. https://doi.org/10.1109/92.335012

[15] V. Tiwari, S. Malik, and A. Wolfe, "Compilation Techniques for Low Energy: An Overview," in *Proceedings of 1994 IEEE Symposium on Low Power Electronics*, 1994, pp. 38–39. https://doi.org/10.1109/lpe.1994.573195

[16] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh, "Techniques for Low Energy Software," in *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, 1997, pp. 72–75. https://doi.org/10.1145/263272.263286

[17] C. Bunse, M. Gottschalk, C. von Ossietzky, S. Naumann, and A. Winter, "2nd Workshop Energy Aware Software-Engineering and Development (EASED@BUIS)," *Softwaretechnik-Trends*, vol. 33, no. 2, pp. 2–3, May 2013. https://doi.org/10.1007/s40568-013-0019-z

[18] T. Hönig, C. Eibel, W. Schröder-Preikschat, B. Cassens, and R. Kapitza, "Proactive Energy-Aware System Software Design with SEEP," *Softwaretechnik-Trends*, vol. 33, no. 2, pp. 6–7, May 2013. https://doi.org/10.1007/s40568-013-0021-5

[19] T. Hönig, C. Eibel, R. Kapitza, and W. Schröder-Preikschat, "SEEP: Exploiting Symbolic Execution for Energy-Aware Programming," *ACM SIGOPS Operating Systems Review*, vol. 45, no. 3, pp. 58–62, 2012. https://doi.org/10.1145/2094091.2094106

[20] S. Naumann, E. Kern, and M. Dick, "Classifying Green Software Engineering – The GREENSOFT Model," *Softwaretechnik-Trends*, vol. 33, no. 2, pp. 18–19, May 2013. https://doi.org/10.1007/s40568-013-0027-z

[21] M. Josefiok, M. Schröder, A. Winter, "An Energy Abstraction Layer for Mobile Computing Devices," *Softwaretechnik-Trends*, vol. 33, no. 2, pp. 12–13, May 2013. https://doi.org/10.1007/s40568-013-0024-2

[22] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the Energy Spent Inside My App? Fine Grained Energy Accounting on Smartphones with Eprof," in *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012, pp. 29–42. https://doi.org/10.1145/2168836.2168841

[23] P. Heinrich and C. Prehofer, "Network-Wide Energy Optimization for Adaptive Embedded Systems," *ACM SIGBED Review*, vol. 10, no. 1, pp. 33–36, 2013. https://doi.org/10.1145/2492385.2492391

[24] D. Shorin and A. Zimmermann, "Evaluation of Embedded System Energy Usage with Extended UML Models," *Softwaretechnik-Trends*, vol. 33, no. 2, pp. 16–17, May 2013. https://doi.org/10.1007/s40568-013-0026-0

[25] C. Stier, A. Koziolek, H. Groenda, and R. Reussner, "Model-Based Energy Efficiency Analysis of Software Architectures," in *European Conference on Software Architecture*. Springer, 2015, pp. 221–238. https://doi.org/10.1007/978-3-319-23727-5_18

[26] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "Anti-Patterns and the Energy Efficiency of Android Applications," arXiv preprint arXiv:1610.05711, 2016.

[27] K. Aggarwal, A. Hindle, and E. Stroulia, "Greenadvisor: A Tool for Analyzing the Impact of Software Evolution on Energy Consumption," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*, 2015, pp. 311–320. https://doi.org/10.1109/ICSM.2015.7332477

[28] I. Polato, D. Barbosa, A. Hindle, and F. Kon, "Hybrid HDFS: Decreasing energy consumption and speeding up hadoop using SSDs," *PeerJ PrePrints*, vol. 3, pp. e1320v1, 2015. https://doi.org/10.7287/peerj.preprints.1320

[29] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What Do Programmers Know About the Energy Consumption of Software?," *PeerJ PrePrints*, vol. 3, p. e886v2, 2015. https://doi.org/10.7287/peerj.preprints.886

[30] C. Zhang, A. Hindle, and D. M. German, "The Impact of User Choice on Energy Consumption," *IEEE software*, vol. 31, no. 3, pp. 69–75, 2014. https://doi.org/10.1109/MS.2014.27

[31] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky, "GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 12–21. https://doi.org/10.1145/2597073.2597097

[32] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating Mobile Application Energy Consumption Using Program Analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 92–101. https://doi.org/10.1109/ICSE.2013.6606555

[33] G. Pinto, F. Soares-Neto, and F. Castor, "Refactoring for Energy Efficiency: A Reflection on the State of the Art," in *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*, 2015, pp. 29–35. https://doi.org/10.1109/GREENS.2015.12

[34] C. Sahin, L. Pollock, and J. Clause, "How Do Code Refactorings Affect Energy Usage?," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2014, pp. 1–10. https://doi.org/10.1145/2652524.2652538

[35] W. G. da Silva, L. Brisolara, U. B. Corrêa, and L. Carro, "Evaluation of the Impact of Code Refactoring on Embedded Software Efficiency," in *Proceedings of the 1st Workshop de Sistemas Embarcados*, 2010, pp. 145–150.

[36] G. Hecht, N. Moha, and R. Rouvoy, "An Empirical Study of the Performance Impacts of Android Code Smells," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, 2016, pp. 59–69. https://doi.org/10.1145/2897073.2897100

[37] B. Tiwana, R. Dick, M. Gordon, L. Zhang, and Z. M. Mao, *A Power Monitor for Android-Based Mobile Platforms*. [Online]. Available: http://ziyang.eecs.umich.edu/projects/powertutor/ [Accessed: 2020].

[38] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, 1976. https://doi.org/10.1109/TSE.1976.233837

[39] A. Qasim, S. Iqbal, Z. Aziz, S. A. R. Kazmi, A. Munawar, B. A. Gilani, and N. Qasim, "Handling Temporal Constraints in Interaction Protocols for Intelligent Multi-Agent Systems," *International Journal on Smart Sensing and Intelligent Systems*, vol. 13, no. 1, pp. 1–15, 2020. https://doi.org/10.21307/ijssis-2020-020

[40] D. M. Vistro, A. Munawar, A. Iftikhar, A. Qasim, and A. U. Rehman, A.U., "Tertiary Care Hospital Monitoring System Using Wireless Sensors," *Journal of Critical Reviews*, vol. 7, no. 10, pp. 1504–1511, 2020.

[41] A. Qasim, H. M. B. Ameen, Z. Aziz, and A. Khalid, "Efficient Performative Actions for E-Commerce Agents," *Applied Computer Systems*, vol. 25, no. 1, pp. 19–32, May 2020. https://doi.org/10.2478/acss-2020-0003

Awais Qasim received the B. sc. degree in Computer Science from the Punjab University College of Information Technology (PUCIT), Lahore, Pakistan, in 2009 and the M. sc. degree in Computer Science from Lahore University of Management Sciences (LUMS), Lahore, Pakistan, in 2011. After that he worked as a Software Engineer in the industry and developed a number of iPhone and Android applications. He works as an Assistant Professor at the Computer Science Department, Government College University. Currently he also works as a Postdoc researcher at the School of Science, Engineering and Environment, University of Salford, the UK. His current research includes developing robotic solutions for real-time disaster management activities. He has published 14 research papers in peer-reviewed ISI indexed journals. Overall, he has conducted research in the areas of code cloning, code smells, model checking, multi-agent systems, real-time systems, and self-adaptive systems.
E-mail: Awais@gcu.edu.pk
ORCID iD: https://orcid.org/0000-0001-8677-9569

Adeel Munawar received the B. sc. degree from Gujranwala Institute of Future Technology University (GIFT), Gujranwala, Pakistan, in 2015 and the M. sc. degree in Computer Science from the Government College University (GCU), Lahore, Pakistan, in 2018. His research interest includes the multi-agent system, artificial intelligence, real-time systems, and natural processing languages. Currently, he works as a Lecturer at Lahore Garrison University, Lahore, Pakistan. From 2019 he is associated with research-related activities to propagate the trends of research at Lahore Garrison University. His research interest includes code smells and design patterns.
E-mail: Adeel.munawar@lgu.edu.pk
ORCID iD: https://orcid.org/0000-0003-3315-3348

Jawad Hassan received the B. sc. degree in Computer Science from the Government College University, Lahore, Pakistan, in 2017 and the M. sc. degree in Computer Science from the Government College University, Lahore, Pakistan in 2019. He has one and a half years of experience as visiting faculty at the Government College University. He joined the Computer Science Department, Lahore Garrison University in 2019 as a Lecturer. He has authored two research papers in peer-reviewed ISI indexed journals. His research interests include multi-agent systems and code cloning.
E-mail: Jawad.hassan@lgu.edu.pk

Adnan Khalid received the PhD degree in Cloud Computing from the University of Engineering & Technology, Lahore, under the supervision of Prof. Dr. M. Shahbaz. He is currently an Assistant Professor with the prestigious Government College University, Lahore. He teaches research methods and software engineering at the undergraduate and postgraduate levels. His area of research is fog computing. He intends to highlight the benefits of this relatively novel field of research. He has 12 scholarly publications in international HEC recognised journals.
E-mail: Adnan.Khalid@gcu.edu.pk