The Impact of Flyweight and Proxy Design Patterns on Software Efficiency: An Empirical Evaluation

Muhammad Ehsan Rana[1], Wan Nurhayati Wan Ab Rahman[2]
Masrah Azrifah Azmi Murad[3], Rodziah Binti Atan[4]
Faculty of Computer Science and IT, Universiti Putra Malaysia
43400 Serdang, Selangor, Malaysia

***Abstract*—In this era of technology, delivering quality software has become a crucial requirement for the developers. Quality software is able to help an organization to success and gain a competitive edge in the market. There are numerous quality attributes introduced by various quality models. Various researches and studies prove that the quality of the object-oriented software can be improved by using design patterns. The main purpose of this research is to identify the relationships between the design patterns and software efficiency quality attribute. This research is focused on the impact of Flyweight and Proxy Design Patterns on the efficiency of software. An example scenario is used to empirically evaluate the effectiveness of applied design refinements on efficiency of a system. The techniques to measure software efficiency and the results obtained for each solution are elaborated in detail. At the end of this research, comparative analysis is provided to show the relative impact of each selected design pattern on software efficiency.**

***Keywords*—Software efficiency; design patterns; flyweight design pattern; proxy design pattern; measuring software efficiency; empirical evaluation of software**

I. Introduction

Efficiency is an essential quality factor that needs to be considered by every software engineer while designing a software program. A highly efficient software can give its users a more pleasant experience when interacting with the software by being more responsive to the user's actions and commands. To design and develop a highly efficient software, designers will have to minimize the system resources used by the software to accomplish the tasks. Increasing the efficiency of the software through the likes of lowering the process time and memory used by the software will make the users feel that the software is more responsive as time used to process their input and provide an output will be lower. The minimal usage of memory will also decrease the time used to search for data, further lowering the processing time.

Due to the competitive nature of software development, developers would need to find ways to satisfy the users' needs. One of the problems of measuring software quality is that it is often intangible and abstract. Therefore, to combat the problem that developers usually face, Jim McCall and several other software engineers have presented their models to ensure quality of developed software using available industry standards.

There are three perspectives of quality attributes in Jim McCall's model, namely Product Revision, Product Transition, and Product Operations. Product revision is the ability or enhancement of the ability for the software to change in accordance to user needs. Product transition on the other hand, is the ability for the software to adapt itself to changing environments. Finally, product operation, which is the topic of discussion of this research, is defined by the ability of the software to operate in accordance to the user demands and without defects [1].

Efficiency is "the state or quality of being efficient", which is the ability of a system or machine on "achieving maximum productivity with minimum wasted effort or expense" or the ability of a person "working in a well-organised and competent way" [2]. However, in the context of software engineering, efficiency is the capability of a software application to fully utilize the amount of resources that are required to perform a task, the resources are inclusive of CPU time, storage, transmission channels and others [3] [4].

As this research is aimed to evaluate the efficiency of software systems, authors are interested in "the ability of a system on achieving maximum productivity with minimum wasted effort or expenses", which in a simpler form is processing most input into output with least amount of resources. Since resource distribution to run software systems are on average the same, more efficient software can usually finish the same task in less amount of time.

II. Literature Review

A. Importance of Efficiency in Contemporary Software Systems

With the development of increasingly advanced hardware, most software, even inefficient, do not need to take full advantage of hardware resources except for a few types, such as games. Due to the abundance of hardware resources, software development typically has low efficiency standards, since the development of an efficient software is more expensive. Due to the higher cost of developing a more efficient software, the end product would also end up costing more. Moreover because of the fact that the end users prefer cheaper products and would usually not notice the subtle difference in performance, commercial software would choose to enhance other aspects of the software rather than enhancing the efficiency [5].

Efficiency is often traded off with other quality attributes as efficiency always comes last in terms of urgency and priority [6]. However, for some real-time systems such a banking system, efficiency is a critical factor in order for the system to be proven useful and successful. Even though the weakness of software efficiency is compensated by the advancement of hardware, efficiency is still pivotal as many users of software are using various specifications of laptops, tablets and smartphones. On the other hand, battery life is an important factor to be considered for mobile devices. If a mobile device is constantly running inefficient software, it will occupy more resources and the battery will be consumed faster.

A case study by [7], depicts the fact that complex systems such as banking systems that require real time interaction between multiple actors are difficult to implement as issues such as efficiency and reliability arises in the process of designing and actually producing the system. Furthermore, introduction of the system over a large geographical area (inclusive of rural areas) requires even more attention towards efficiency as data has to be transmitted, received, and synced among all nodes within the network in real time. The same could be seen in Healthcare Systems which poses emphasis on efficiency as the equipment which are connected using Internet of Things (IoT) have to provide real-time response for notification purposes when the monitored patient's condition is near or in emergency level [8] and serious implications could happen when the system fails to extend immediate feedback to the target audience (in this case, the nurses and the doctors) for assistance.

Efficiency is becoming more important in software applications that uses emergent technologies as it emphasize on using fewer resources to achieve better results, saving computational time and storage. The importance of efficiency can be seen through telecommunication software applications such as Skype, WhatsApp or Facebook Messenger which provides video or voice calling services through wireless networks. In the context of Skype, a study that was done by [9] have shown that the video quality of Skype calls is greatly affected by the efficiency of Skype's transmission and encoding algorithms as well as utilizing the bandwidth resource provided by the targeted machine itself. Similarly, within the healthcare sector, through integration of IoT, surgeons are able to perform telesurgery on patients without having to be physically present at the operation theatre [10]. In these cases, efficiency of the software application is extremely crucial as defective or inefficient software could potentially put the patient's life to risk, hence the software application must perform at maximum efficiency to ensure there is zero to none disruption. Content management system (CMS) which is a type of data driven system is used to perform CRUD (create, remove, update, delete) functionalities on text data and this stored data is then used for big data analytics [11]. Due to the large amount of data being stored in the database, the software has to utilize the computing power by optimizing insert and retrieval codes to ensure the performance of the system.

In a typical enterprise environment, development of software applications is usually done on cloud where services are delivered to clients or end-users through powerful virtualized data servers that are equipped with high bandwidth and low latency network speed that are kept in data-centers that are owned by external parties (the cloud service providers such as Microsoft Azure, Amazon Simple Storage Service et cetera.) [12]. Cloud computing is considered as the integration of both Grid Computing and Cluster Computing paradigms. Cloud computing architecture allows the software system to be easily expandable and scalable due to the nature of the services provided [13]. However, efficiency is one of the main reasons why most enterprises begin to shift their services to be hosted through the cloud instead of having the servers hosted and managed manually within their company's premises. Efficiency could be achieved in cloud computing paradigm through shared resource pooling which improve data storage and processing power, for example the service providers dynamically assign computing resources to multiple consumers only when needed, hence maximum utilization of computing resources could be achieved [14]. Also, the amount of computational power or storage could be increased or decreased at any point of time based on the amount of resources that will be needed by the software application; hence it does not waste computing resources and at the same time allows consumers to save cost. Due to the flexibility, consumers can ensure that optimum efficiency is reached in order to provide end-users with a system that runs with high performance. Through cloud computing architecture, consumers do not have to worry with concerns on hardware issues and can pay full focus in development of software.

B. Methods to Improve Software Efficiency

1) Code optimization: In order to achieve higher efficiency in software applications, the developers of the system should be able to fully grasp and understand how codes function at an operating system level and optimize the codes to enhance computing resource utilization as area of slowness could only be identified through knowledge and experience in dealing with system related matters. For instance, declaring cached reference of objects that is frequently used as a local variable could greatly help in reducing the processing power needed to complete a task [15]. The diagram represented by Fig. 1 below shows the actual implementation of cached memory.

In Fig. 1, the code snippet on the left is the initial implementation of the code without using cached memory. The difference in two implementations is only a line of code, however it can make a huge difference when the computer interprets the code. The code on left have to retrieve the draw method every single time the loop is executed, however the code on the right retrieves the draw method once, and for each loop execution the reference to the retrieved method is called. Various modifications such as avoiding recursion, managing threads etc. could also help in enhancing efficiency of a software application [16].

```
for i in xrange(1000):
    my_mesh.draw()
```

```
draw_method = my_mesh.draw
for i in xrange(1000):
    draw_method()
```

Fig. 1. Code Snippets Showing the Variation in Efficiency.

2) Parallel programming: Parallel programming is the method of programming which utilize threads where each thread could process a single line of command, however many threads could be created and kept alive concurrently. In the current real-world scenario, industries are paying more focus on hardware components rather than software quality, hence stronger processing chips are being researched on and produced over the years whereas the efficiency of a software is still neglected and there is no major improvement in terms of efficiently utilizing the computing power which is being provided by the chips [17]. However, in systems which are heavily reliant on efficiency, parallel programming would be among the most suitable to be implemented within the actual source code.

3) Design patterns: Design patterns are tested and reusable solutions to reoccurring problems. Some of the most commonly used software design patterns are the twenty-three object-oriented patterns proposed by the Gang of Four (GoF), consisting Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, in their book "Design Patterns: Elements of Reusable Object-Oriented Software", published in 1994 [18]. Design patterns could be commonly seen within the context of object-oriented programming where coupling and coherence plays a very important role in determining the quality of a software. Design patterns are classified into three categories, creational addressing the process of object creation; structural addressing composition of objects and classes; behavioral addressing how classes and object interact and distribute responsibility [18]. Through the usage of design patterns, a common structure of code implementation can be applied to solve certain type of issues. As design patterns are commonly used, recognized and practiced by experienced object-oriented developers, the solutions can be categorized to be the best possible solution in specific situations. Since they are commonly used by developers, the solutions are practically stable in nature. Most of the time in the real-world environment, design patterns are also the commonly used as a base for most software development projects, this help saves time as reoccurring problems could be solved immediately through implementation of design patterns without the need of rethinking code structure and design for constantly emerging problems. As exploring design patters for improvement in efficiency is the core objective of this research, subsequent sections provide details on how it can impact software efficiency.

C. Impact of Design Patterns on Software Efficiency

Design patterns could help in breaking down tight coupling between classes and objects and with such, the structure of an application can be broken down into few separated parts which is typically implemented through the MVC (model, view and controller) architecture. Efficiency of a software could be improved by implementing heavy loading (functions that require a large amount of processing power) to be done at server-side machines with higher computational power and allowing clients to access the Web APIs service to invoke the functions and features [19]. Within the MVC architecture, the Decorator pattern can be used in controller, the Strategy pattern between the controller and view and the Observer pattern in order to notify the view when the model is changed, whereas Factory pattern can be used for creation of multiple views or controllers.

Design patterns could also help in reducing the computing power consumed by an application but instantiating lesser objects and instead of creating new objects, similar objects are shared in usage. For example, the Flyweight design pattern allow objects to be shared. Whereas the Proxy design pattern can improve the efficiency of an application by avoiding duplication of object especially for objects that is very large in size. For example, in a typical web application environment, end users usually make requests to the server multiple times, instead of responding multiple times, the proxy pattern will check if the existing object exists and try to return the local reference if there is an existing object.

Some of the existing researches also suggest that the design patterns Factory, State, and Proxy can improve the performance of a software system by caching or skipping repetitive procedures [20, 21, 22, 23, 24, 25, 26]. In one of the tests conducted by Erik Jansson, it was shown that flyweight has noticeably improved the memory usage of the software [27].

Following is a brief description of the Flyweight and Proxy design patterns, which are used in this research for studying the impact on software efficiency:

1) Flyweight design pattern: The Flyweight design pattern as defined by GoF [18] is "using sharing to support large numbers of fine-grained objects efficiently". In other words, flyweight is a pattern which aims to reduce the number of objects that are created, and instead of creating a large number of objects with tiny differences in attributes the Flyweight will use the shared pool of objects with intrinsic state and extrinsic state properties [28]. The intrinsic state is the predefined states for the object which is constant and unchangeable whereas the extrinsic state is the attribute which is determined during run time. Fig. 2 below shows the general structure for Flyweight implementation.

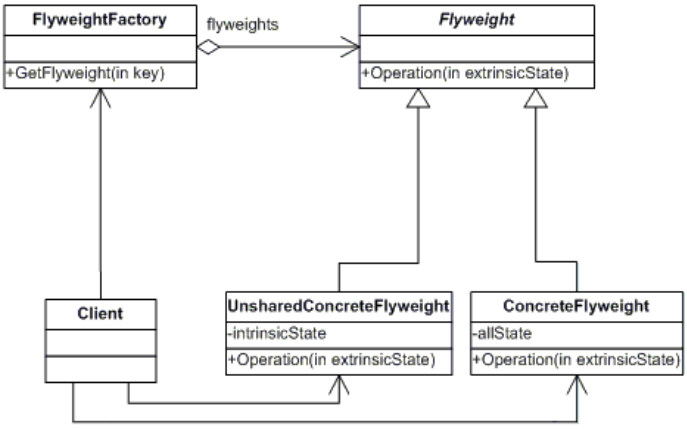

Fig. 2. Flyweight Generic Design.

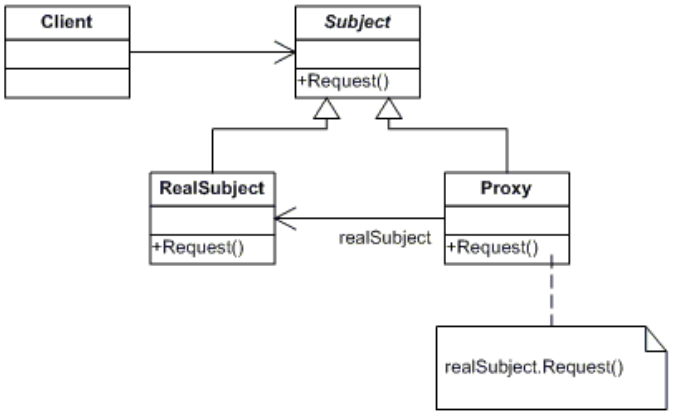

Fig. 3. Proxy Generic Deisgn.

2) Proxy design pattern: The Proxy design pattern as defined by GoF [18], is "providing a surrogate or placeholder for another object to control access to it". Proxy is categorized into four main types, which are the remote proxy, virtual proxy, protection proxy and smart proxy respectively. Remote proxy is used for the purpose of accessing remote services by processing on the request that was sent by the client; Virtual proxy is for delaying the sending of the actual until the moment where the data is required; Protection proxy is to act as a layer of protection when accessing data; Smart proxy is to check the caching of objects. As shown in Fig. 3 above, the Proxy design pattern consists of four main components, the Client which send requests; the Subject (interface) which handles request made by the Client; the Real Subject which is the actual object or data; the Proxy which is a pointer or reference to the actual data or object. The Proxy acts a checker or a control which identifies if similar request is previously sent from the Client, if the result to the request is found within the local context of the Proxy, then the Proxy will redirect the request to the local reference of the result instead of accessing the Real Subject to request for the result again. In cases where the Real Subject is heavyweight, through the implementation of proxy design pattern, greater efficiency could be achieved.

D. System Attributes related to Efficiency

As efficiency is a broad term, there exists many ways in which one can measure a system's efficiency. As proposed by [29], there are a few sub-factors in a system which are related to efficiency. The sub-factors include:

- Time Behavior–The response time, throughput, and capacity to perform of a system.
- Resource Behavior–How much resource is used by the system while performing its tasks. The resources can be random access memory (RAM), read-only memory (ROM) as well as the utilization of Input and Output device.
- Reply Time–The time passed between which an inquiry or demand is given to a system and the beginning of the system's response to the inquiry or demand.
- Processing speed–The amount of time used by a system to complete a task or the actual time spent by the user on the system to generate a result.
- Execution Efficiency–The run time performance of a component in the system.
- Robustness–The ability of a component in the system to execute tasks correctly when given incorrect inputs or while under stressful environmental conditions to give desired results.

With higher efficiency, less hardware resources and more importantly, less time will be consumed, and accomplishing more tasks deemed very important by the user under a given amount of time. Therefore, in order to measure the efficiency of a system, these factors should be considered.

III. Design and Implementation of Simpler and Refined Solutions

To measure the impact of design refinements on system efficiency, a simple imitation of an online shooter game is designed and implemented. Initially a simpler solution is implemented without applying any design refinements. Then the same design is improved by applying appropriate design patterns and the solutions is re-implemented. Finally efficiency of simpler legacy solution as well as the refined design pattern based solutions are calculated by measuring the execution time and memory usage. Comparison of the solutions highlight the impact and effectiveness of the design patterns used and help in identifying the design refinements that should be used to make a software system efficient.

Scenario: In an online shooter game, there are two teams consisting of defenders and attackers. 200 players will be playing in a single map and they can either choose to be an attacker or defender. These players can also choose their desired weapons. Once players have chosen their weapons, the attackers will spawn at a location and the defenders will spawn another. For imitation purpose, the 200 players will randomly choose classes and we assume that the likelihood of them picking each of the weapons will be the same. The attackers are assigned the task to attack the objective while the defenders are assigned to defend it.

Following represents the design of a simpler solution:

As shown in Fig. 4, the basic solution contains four (4) classes. The Players are separated into Attackers and Defenders. The OnlineShooter class (Fig. 5) will then spawn the Attacker and Defender Players. In the OnlineShooter class, there are 2 arrays which are playerTeam, containing the teams the players can join, and Weapon, which is the weapon that the players can pick. For each player, the player will be generated by placing them into a random team with a random weapon selected using the Java Random utility. Once the player is loaded, they will be spawned.

The Player interface (Fig. 6) contains methods that will be implemented in the Attacker and Defender class which are assignWeapon(String weapon), for assigning the randomly chosen weapon to the players, spawn(), which spawns the players, and loadModel() which will load the player models.

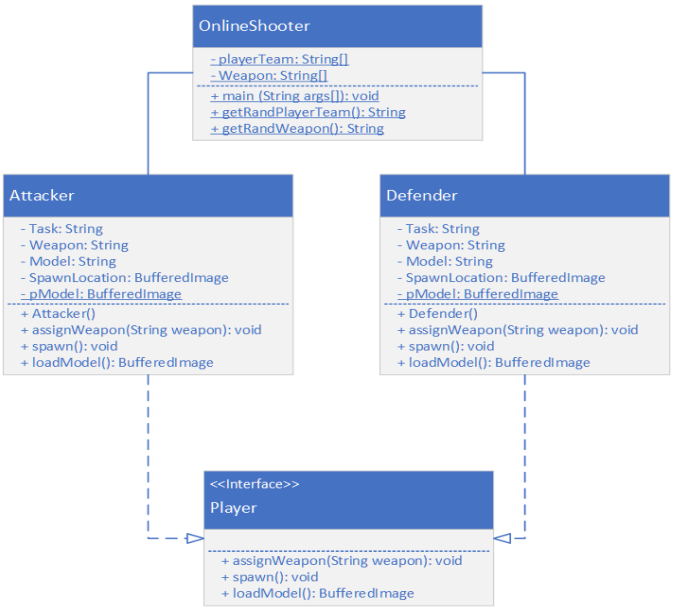

Fig. 4. UML Class Diagram for Simpler Solution.

```
package dpatproj.basic;
import java.util.Random;
public class DPATProjBasic {
    private static String[] playerTeam = {"Attacker", "Defender"};
    private static String[] Weapon = {"Bow","Sword","Axe","Shield",
                "Dagger","Sabre","Spear","Club","Hammer","Glaive"};

    public static void main(String[] args) {
        for(int i=0;i<200;i++){
            Player p = null;
            String PlayerTeam = getRandPlayerTeam();
            if (PlayerTeam == "Attacker"){
                p = new Attacker();
                System.out.println("Attacker Created");
            }
            else if (PlayerTeam == "Defender"){
                p = new Defender();
                System.out.println("Defender Created");
            }
            else{
                System.out.println("No such player");
            }
            if (p != null){
                p.assignWeapon(getRandWeapon());

                p.spawn();
            }
        }
    }
    public static String getRandPlayerTeam(){
        Random r = new Random();
        int randInt = r.nextInt(playerTeam.length);
        return playerTeam[randInt];
    }
    public static String getRandWeapon(){
        Random r = new Random();
        int randInt = r.nextInt(Weapon.length);
        return Weapon[randInt];
    }

}
```

Fig. 5. Online Shooter Class.

```
package dpatproj.basic;

import java.awt.image.BufferedImage;

public interface Player {
    public void assignWeapon(String weapon);
    public void spawn();
    public BufferedImage loadModel();
}
```

Fig. 6. Player Interface.

```
package dpatproj.basic;
import java.awt.image.BufferedImage;
import java.io.File;
import javax.imageio.ImageIO;
public class Attacker implements Player{
    private final String Task;
    private final String Model;
    private String Weapon;
    private BufferedImage SpawnLocation = null;
    private static BufferedImage pModel;

    Attacker(){
        Task = "ATTACK THE OBJECTIVE";
        Model = "Example.png";
        try{
            SpawnLocation = ImageIO.read(new File(Model));
        }catch (Exception e){}
    }

    public void assignWeapon(String weapon){
        this.Weapon=weapon;
    }

    public void spawn(){
        pModel = loadModel();
        System.out.println("Attacker with weapon " + Weapon +
        " spawned |" + " Task is " + Task);
    }
    public BufferedImage loadModel(){
        System.out.println("Loading Model " + Model + "...");
        BufferedImage img = null;
        try{
            img = ImageIO.read(new File(Model));
        }catch (Exception e){
            System.out.println("Player model not found!");
        }
        System.out.println("Model "+ Model + " loaded");
        return img;
    }
}
```

Fig. 7. Attacker Class.

As Attacker and Defender classes are similar, only the Attacker class is shown in Fig. 7 above. In both Attacker and Defender class, the task, file location of the player model, and the weapon assigned are stored as Strings, while the spawn location and actual player models are stored as buffered images to simulate the actual process of loading the players, which needs a considerable amount of time and memory.

A. First Design Refinement using Flyweight Design Pattern

In above design and implementation, there is a problem with the system needing to create a completely new Player object for every single player loaded, even though there are only slight differences between the different Player objects. For each Attacker or Defender, the Task, Model, and Spawn Location are the same, with only their weapon being different.

This creates a situation where the system needs to create about 99 player objects for each type of player which can be avoided.

To solve this issue, Flyweight design pattern is implemented. By using Flyweight, each Player object will only have to be instantiated once, which can then be reused repeatedly, only needing to reassign their weapons each time instead of reassigning everything. The refined UML Class Diagram (Fig. 8) is depicted below.

As can be seen from Fig. 8 below, most of the classes remained same as the basic implementation, except that a PlayerFactory class is added, which the OnlineShooter class will use for the generation of Players. The implementation of the Attacker and Defender classes as well as the Player interface is the same. The DPATProjFlyweight in the implementation however, has a minor change, which is the generation of Player object, instead of directly making a new Player, the player is gotten from the PlayerFactory class.

The PlayerFactory class, as shown in Fig. 10, is a new class that was not implemented in the simpler solution of the scenario. The class contains a HashMap, which will be used as a key that is associated with created objects. When the getPlayer() method is called, the system will first check if the player type is already created by searching for the associated key. Then return the player object if it is already created, not needing to recreate a new object. If the associated key cannot be found, only then the system will create a new object and store it with its associated key before returning it.

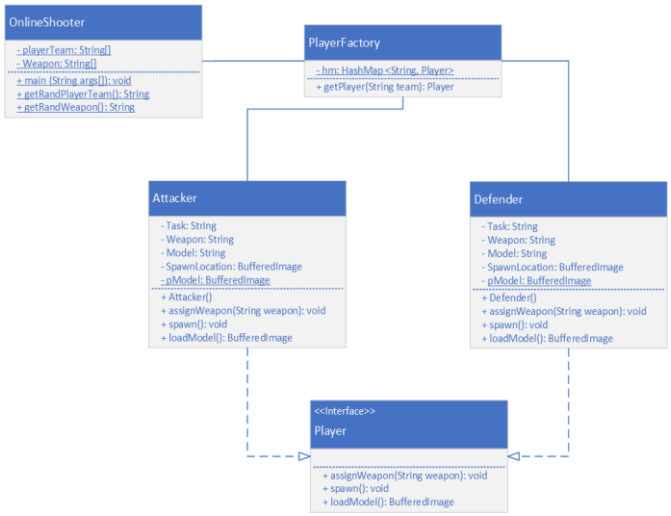

Fig. 8. UML Class Diagram for Flyweight Pattern based Refinement

```
public static void main(String[] args) {
    for(int i=0;i<200;i++)
    {
        Player p = PlayerFactory.getPlayer(getRandPlayerTeam());

        p.assignWeapon(getRandWeapon());

        p.spawn();
    }
}
```

Fig. 9. Online Shooter Class tweak

```
package dpatproj.flyweight;

import java.util.HashMap;

public class PlayerFactory {

    private static HashMap <String, Player> hm = new
    HashMap<String, Player>();

    public static Player getPlayer(String team)
    {
        Player p = null;

        if (hm.containsKey(team))
                p = hm.get(team);
        else
        {
            switch(team)
            {
            case "Attacker":
                System.out.println("Attacker Created");
                p = new Attacker();
                break;
            case "Defender":
                System.out.println("Defender Created");
                p = new Defender();
                break;
            default :
                System.out.println("No Such Player Type");
            }

            hm.put(team, p);
        }
        return p;
    }
}
```

Fig. 10. Player Factory Class.

B. Second Design Refinement using Proxy Design Pattern

There is still an issue in the refined solution, whereby before each player is spawned; their player models need to be loaded from a specific file. In this case, spawning 200 players would mean that the system would have to repeatedly read the player model from a remote file for 200 times, which 99 times can be avoided for each player type (as in Fig. 9).

To resolve this issue, Proxy design pattern is implemented alongside Flyweight, where each player type's model file would only have to be accessed and read once, and then the system will store the remote file's information as an object, which can then be reused repeatedly without needing to access the remote file again. Fig. 11 shows the UML class diagram designed for the implementation with Proxy design pattern.

In the refined UML class diagram (Fig. 11) depicted above, no changes are made to the already existing classes except for a few variables in the Attacker and Defender classes. However, two interfaces AttackerModel and DefenderModel, which are the player models for each of their respective player types, are added. Proxy player model classes ProxyAtkM and

ProxyDefM as well as real player model classes RealAtkM and RealDefM are shown for their respective player types. As in the first refinement, classes that are unchanged are not shown here, that include OnlineShooter (renamed to DPATProjFProxy), PlayerFactory, and the Player interface. There are slight changes to the Attacker and Defender classes. The String Model variables are changed to a type of their respective model classes and the loadModel() method is replaced by the getModel() method, which gets the player model from their respective classes as a BufferedImage as shown in Fig. 12 below.

The AttackerModel and DefenderModel interfaces in Fig. 13 and Fig. 14 contain a getModel() method, which is used for the Attacker and Defender classes to get player model.

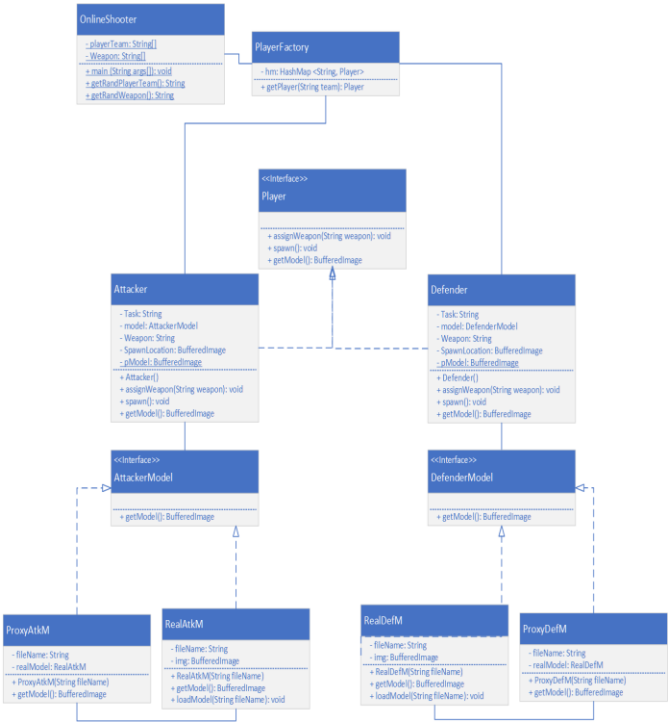

Fig. 11. UML Class Diagram for Proxy Pattern based Refinement.

```
private final String Task;
AttackerModel model;
private String Weapon;
private BufferedImage SpawnLocation;
private static BufferedImage pModel;

public BufferedImage getModel(){
    BufferedImage pImg = model.getModel();
    return pImg;
}
private final String Task;
DefenderModel model;
private String Weapon;
private BufferedImage SpawnLocation;
private static BufferedImage pModel;

public BufferedImage getModel(){
    BufferedImage pImg = model.getModel();
    return pImg;
}
```

Fig. 12. Player getModel().

```
public interface AttackerModel {
    public BufferedImage getModel();
}
```

Fig. 13. AttackerModel interface.

```
public interface DefenderModel {
    public BufferedImage getModel();
}
```

Fig. 14. DefenderModel Interface.

The ProxyAtkM and ProxyDefM classes in Fig. 15 and Fig. 16 are the proxy classes for the player models. There are String variables for storing the file name of the player model and RealAtkM and RealDefM type classes which is used to refer to the real model classes. When the getModel() method is called, it will first check if the real model classes are instantiated, and get the player model from the real model class and return it.

```
package dpatproj.fproxy;

import java.awt.image.BufferedImage;

public class ProxyAtkM implements AttackerModel{
    private String fileName;
    private RealAtkM realModel;

    ProxyAtkM(String fileName){
        this.fileName = fileName;
    }

    public BufferedImage getModel(){
        if (realModel == null){
            realModel = new RealAtkM(fileName);
        }
        return realModel.getModel();
    }
}
```

Fig. 15. ProxyAtkM Class.

```
package dpatproj.fproxy;

import java.awt.image.BufferedImage;

public class ProxyDefM implements DefenderModel{
    private String fileName;
    private RealDefM realModel;

    ProxyDefM(String fileName){
        this.fileName = fileName;
    }

    public BufferedImage getModel(){
        if (realModel == null){
            realModel = new RealDefM(fileName);
        }
        return realModel.getModel();
    }
}
```

Fig. 16. ProxyDefM Class.

The RealAtkM and RealDefM classes in Fig. 17 and Fig. 18 are responsible for loading and storing the player model from the remote file. Similar to the proxy classes, there is a String variable in each of the real model classes to store the file name of the model. The BufferedImage is first instantiated as null but when the class is first instantiated, the model will be loaded into the BufferedImage. When the getModel is called by the proxy classes every next time, the BufferedImage will be returned without needing to be loaded again.

```
package dpatproj.fproxy;

import java.awt.image.BufferedImage;
import java.io.File;
import javax.imageio.ImageIO;

public class RealAtkM implements AttackerModel{
    private String fileName;
    BufferedImage img = null;

    RealAtkM(String fileName){
        this.fileName = fileName;
        loadModel(fileName);
    }

    public BufferedImage getModel(){
        System.out.println("Model " + fileName + " Loaded");
        return img;
    }

    public void loadModel(String fileName){
        System.out.println("Loading Model " + fileName + "...");
        try{
            img = ImageIO.read(new File(fileName));
        }catch (Exception e){
            System.out.println("Player model not found!");
        }
    }
}
```

Fig. 17. RealAtkM Class.

```
package dpatproj.fproxy;

import java.awt.image.BufferedImage;
import java.io.File;
import javax.imageio.ImageIO;

public class RealDefM implements DefenderModel{
    private String fileName;
    BufferedImage img = null;

    RealDefM(String fileName){
        this.fileName = fileName;
        loadModel(fileName);
    }

    public BufferedImage getModel(){
        System.out.println("Model " + fileName + " Loaded");
        return img;
    }

    public void loadModel(String fileName){
        System.out.println("Loading Model " + fileName + "...");
        try{
            img = ImageIO.read(new File(fileName));
        }catch (Exception e){
            System.out.println("Player model not found!");
        }
    }
}
```

Fig. 18. RealDefM Class.

IV. Empirical Evaluation and Analysis of Results

Efficiency of a system can be calculated by measuring its execution time and its memory usage when executing tasks. The efficiency of a system is inversely proportional to both memory usage and execution time. Therefore, in order for a system to be considered as efficient, its memory usage as well as execution time needs to be as low as possible. In order to show the impact of design patterns on system's efficiency, the memory usage will be measured in Mega Bytes (MB) and its execution time will be measured in Seconds.

To measure these attributes of the system, the above designs are implemented in Java programming language using NetBeans IDE. NetBeans is chosen as it has an internal profiler which automatically tracks the project's memory usage, CPU usage and Garbage Collection, etc. To obtain system's memory usage, one would only need to run the project in the profiler's telemetry tab, and the memory usage at each stage of the project's execution will be shown in a graphical form. For each execution of a project, NetBeans' internal output console will also display the project's completion time after it has finished executing the project. The completion time can then be directly translated to the execution time of the system. To simulate large amount of data being processed by a real system, BufferedImage (a Java library) is used. The large file size of an image, when loaded into the system using BufferedImage, can simulate a large object in a real system. Without the existence of a large object, the execution time and memory usage of the implementation will be too low to have a noticeable difference.

Each variant of differently designed implementation was run multiple times using NetBean's built-in profiler to obtain the memory usage and execution time so that the time needed to start the profiler is not calculated into the execution time. Following are the results for each of the three variant implementations discussed above:

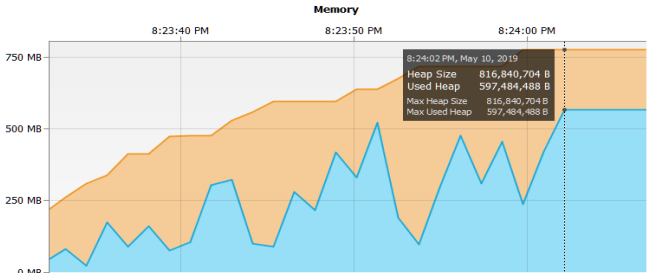

Fig. 19. Memory usage by Simpler Legacy Solution.

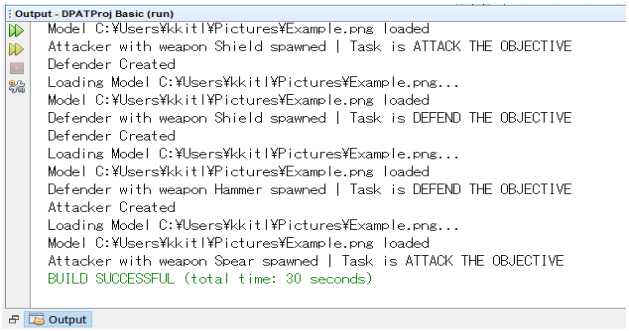

Fig. 20. Execution Time Taken by Simpler Legacy Solution.

The results for the simpler implementation are shown in Fig. 19 and Fig. 20 above. The memory needed to complete running the system was 817MB and the execution time was 30 seconds. As one would expect, the memory and time would increase proportionally to the number of players generated as well as the file size of the Buffered Image.

Fig. 21 and Fig. 22 shows the results for refined implementation with Flyweight pattern. Here the memory needed to complete the execution of the project was 648MB, which is significantly lower than the simpler solution, while the execution time was 15 seconds, which is half of what the simpler implementation needed.

Finally, the results of running the second refinement using both Flyweight and Proxy design pattern are shown in Fig. 23 and Fig. 24. The memory usage is even lower than the first refinement, only needing 129MB. The execution time was surprising reduced to less than 1 second.

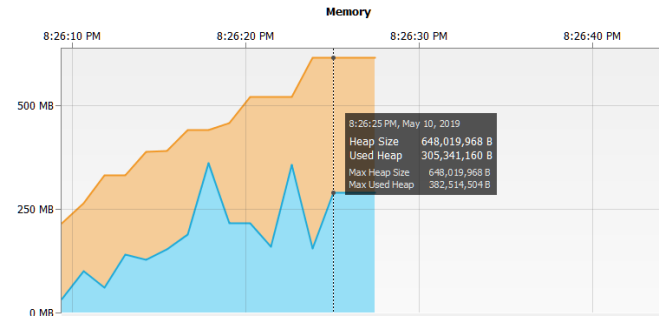

Fig. 21. Memory usage by Flyweight Pattern based Solution.

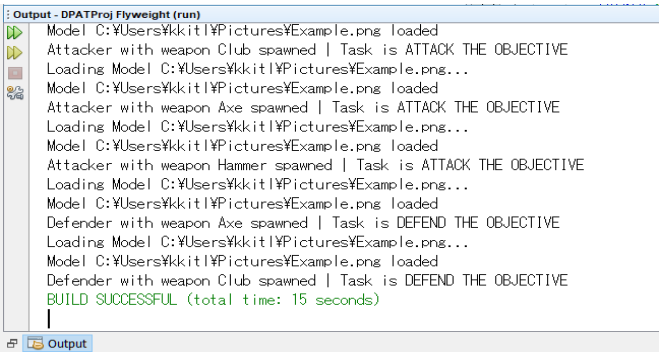

Fig. 22. Execution Time Taken by Flyweight Pattern based Solution.

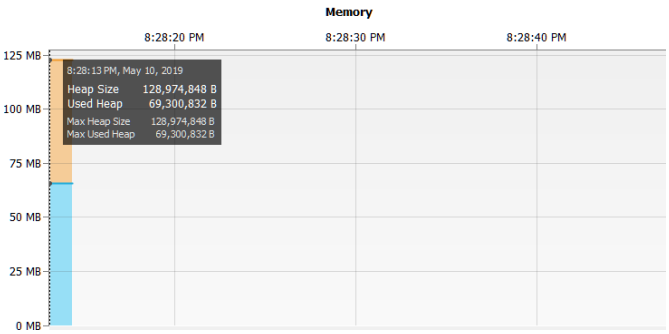

Fig. 23. Memory usage by Proxy Pattern based Solution.

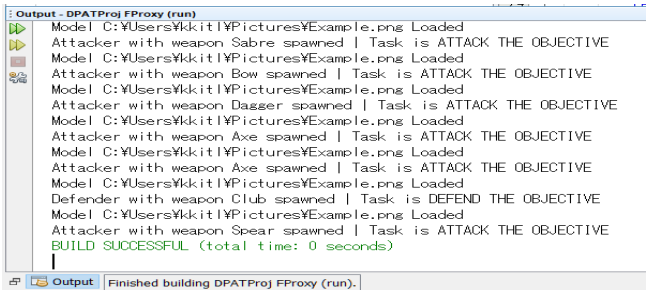

Fig. 24. Execution Time Taken by Proxy Pattern based Solution.

Fig. 25 and Fig. 26 show the comparison of the three solutions using a bar chart in order to better illustrate the impact of the design patterns.

From the results above, it is obvious that both design patterns have significantly lowered the execution time as well as the memory usage. Applying the flyweight pattern has lowered the memory usage by 349MB while applying proxy pattern has further lowered the memory usage by 519MB. Both design patterns have also shown a significant decrease in the execution time which is 15 seconds.

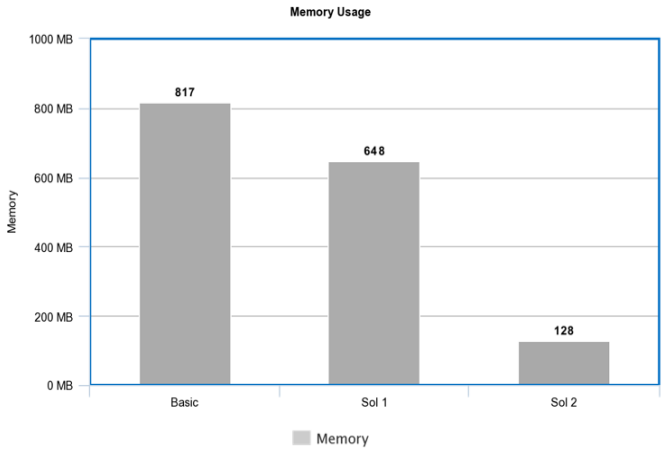

Fig. 25. Memory usage Comparison.

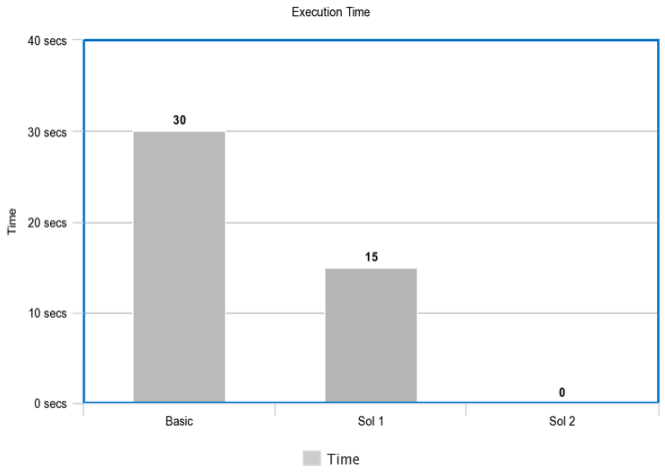

Fig. 26. Execution Time Comparison.

V. Conclusion

Throughout the context of this research, the impact of efficiency on software systems is explored and examined and later evaluated using a simple imitation of an online shooter game. This research also analyze the effectiveness of applied design refinements on efficiency of a system. The design refinements were carried out by applying Flyweight and Proxy design patterns on a simpler solution. In conclusion, Flyweight and Proxy design patterns can both be very effective at increasing the efficiency of a system by decreasing the execution time and memory usage provided they are implemented in the right context.

References

[1] ProQA, "ProfessionalQA," 2016. [Online]. Available: http://www.professionalqa.com/mc-call-software-quality-model.[Accessed 5 4 2019].

[2] Oxford, Oxford English Dictionary, 7 ed., Oxford: Oxford University Press, 2012.

[3] J. P. Cavano and J. A. McCall, "A Framework for the Measurement of Software Quality," ACM SIGSOFT Software Engineering Notes, vol. 3, no. 5, pp. 133-139, 1978.

[4] M.-C. Lee, "Software Quality Factors and Software Quality Metrics to Enhance Software Quality Assurance," British Journal of Applied Science & Technology, vol. 4, no. 21, pp. 3069-3095, 2014.

[5] D. H. M. Kabutz, "JavaSpecialists," 2019. [Online]. Available: https://www.javaspecialists.eu/archive/Issue267.html. [Accessed 5 4 2019].

[6] H.-W. Jung, S.-G. Kim and C.-S. Chung, "Measuring Software Product Quality: A Survey of ISO/IEC 9126," IEEE Software , vol. 21, no. 5, pp. 88-92, 2004.

[7] P. M. Leonardi, D. E. Bailey, E. H. Diniz, D. Sholler and B. Nardi, "Multiplex Appropriation in Complex Systems Implementation: The Case of Brazil's Correspondent Banking System", MIS Quarterly, vol. 40, no. 2, pp. 461-473, 2016.

[8] T. N. Gia, M. Jiang, A.-M. Rahmani, T. Westerlund, P. Liljeburg and H. Tenhunen, "Fog Computing in Healthcare Internet of Things: A Case Study on ECG Feature Extraction," 2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing, pp. 356-363, 2015.

[9] L. Li, K. Xu, D. Wang, C. Y. Peng, K. Zheng, H. Y. Wang, R. Mijumbi and X. X. Wang, "A Measurement Study on Skype Voice and Video Calls in LTE Networks on High Speed Rails," 2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS), pp. 1-10, 2017.

[10] Y. Yin, Y. Zeng, X. Chen and Y. Fan, "The internet of things in healthcare: An overview," Journal of Industrial Information Integration, vol. 1, pp. 3-13, 2016.

[11] J. Thrivani, K. Venugopal and B. Thomas, "An Efficient Cloud Based Architecture for Integrating Content Management Systems," 2017 IEEE International Conference on Innovative Mechanisms for Industry Applications, pp. 337-342, 2017.

[12] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg and I. Brandic, "Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility," Future Generation Computer Systems, vol. 25, pp. 599-616, 2015.

[13] Q. Zhang, L. Cheng and R. Boutaba, "Cloud Computing: State-of-the-Art and Research Challenges," IEEE Internet Computing , vol. 13, no. 5, pp. 10-13, 2009.

[14] H. T. Dinh, C. Lee, D. Niyato and P. Wang, "A Survey of Mobile Cloud Computing: Architecture, Applications, and Approaches," Wireless Communications and Mobile Computing, vol. 13, pp. 1587-1611, 2013.

[15] K. Busch, "The Rules of Optimization: Why So Many Performance Efforts Fail," 2016. [Online]. Available: https://hackernoon.com/the-rules-of-optimization-why-so-many-performance-efforts-fail-cf06aad89099. [Accessed 7 August 2018].

[16] E. Paraschiv, "How to Improve the Performance of a Java Application," 2018. [Online]. Available: https://dzone.com/articles/how-to-improve-the-performance-of-a-java-applicati. [Accessed 7 August 2018].

[17] S. H. Fuller and L. I. Millett, "Computing Performance: Game Over or Next Level?," Computer, vol. 44, no. 1, pp. 31-38, 2011.

[18] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesle, 1995.

[19] F. E. Shahbudin and F.-F. Chua, "Design Patterns for Developing High Efficiency Mobile Application," Journal of Information Technology & Software Engineering, vol. 3, no. 3, 2013.

[20] M. Ayata, "Effects of Some Software Design Patterns on Real Time Software Performance," Middle East Technical University, Ankara, Turkey, 2010.

[21] C. Clifton, "Some GoF Design Patterns:," Rose-Hulman Institute of Technology, Terre Haute, Indiana.

[22] Devlob, "Medium - Proxy Design Pattern to Speed up Your Applications!," 2018. [Online]. Available: https://medium.com/@devlob/proxy-design-pattern-to-speed-up-your-applications-2416816493d. [Accessed 16 4 2019].

[23] B. M. Dotte and D. C. Julson, "Using Design Patterns to Improve the Run-Time Efficiency of Real-Time," University of Wisconsin-Eau Claire , Eau Claire, Wisconsin.

[24] K. Khosravi and Y.-G. Gu´eh´eneuc, "A Quality Model for Design Patterns," German Industry Standard, 2004.

[25] R. Carr, "BlackWasp - Proxy Design Pattern," [Online]. Available: http://www.blackwasp.co.uk/Proxy.aspx. [Accessed 21 4 2019].

[26] M. E. Rana and W. N. W. A. Rahman, "The Effect of Applying Software Design Patterns on Real Time Software Efficiency," in Future Technologies Conference (FTC) 29-30 November 2017, Vancouver, 2017.

[27] E. S. Jansson, "Performance Effect of Flyweight in," Linköping University, Sweden, Linköping, Sweden, 2015.

[28] D. Geary, "Make your apps fly | JavaWorld," 2003. [Online]. Available: https://www.javaworld.com/article/2073632/build-ci-sdlc/make-your-apps-fly.html. [Accessed 8 August 2018].

[29] A. K. Sharma, "Component Based Systems: A Quality Assurance Framework," Himachal Pradesh University, Shimla, 2013.