Cuong Le

Design Patterns

Implementation in video game programming

Metropolia University of Applied Sciences

Bachelor of Engineering

Degree programme in Information Technology

Thesis

6 May 2016

Author(s) Title Number of Pages Date	Cuong Le Design Patterns, Implementation in video game programming 37 6 May 2016
Degree	Bachelor of Engineering
Degree Programme	Degree Programme in Information Technology
Specialization option	Software Engineering
Instructor(s)	Antti Laiho, Senior Lecturer Toni Nylund, Vulpine Games CEO

The goal of this thesis was to develop Last Planets, a social mobile game for iOS devices. The game development theory and the design patterns are portrayed in the first part of this study. The second part presents how such theories are put into practice during the development of Last Planets.

The completion of the project resulted in the launch of Last Planets during spring 2016. Multiple design patterns were chosen to be implemented within the code base. Patterns such as Observer, Strategy and Model-View-Controller create the foundation architecture and convention for the code base. Meanwhile, various other support systems are implemented with the help of Singleton and Decorator pattern. However, the implementation process did not occur instantaneously nor at the beginning of the project but was rather a gradual realizing and refactoring effort.

Applying these design patterns has helped ease up maintenance work as well as improved the readability of the source code. Additionally, these design patterns have also enabled several major functionalities of both client and server sides. However, in a few cases of overusing design patterns, the game source code has actually become more complex and tangled.

Since the materials of this thesis are limited, readers are highly recommended to study design patterns further. Careful consideration before applying design pattern is also vital for project success. Insightful decisions can both maximize the values of design patterns as well as avoiding the risk of abusing them.

Keywords	programming, design pattern, Vulpine, Games, Last Planets

Metropolia

Contents

1 Introduction

This thesis shows the principles of game programming and how design patterns affects the development of a video game. The goal of this study is to lay the theoretical foundation behind the Last Planets code architecture and provide useful patterns for other programmers to study or implement.

There are three main parts in this thesis. The first part portrays the overview of game development. The second part shows fundamental concepts of design patterns and several pragmatic well-known patterns. In the third part, multiple design pattern implementations are examined. At the end of this thesis, findings and results are concluded.

The first part - game programming theory - serves the purpose of getting background knowledge for improving game architecture. At the beginning of Last Planets game development, prototypes were created in order to test out game ideas and assess the risks related. The codes created during this phase are often dirty and expendable. Therefore, the programming principles are considerably important during the transition from prototypes into actual production phase game development. Such principles should be applied to improve both reusability and maintainability of the code base.

Design patterns illustrate clever reusability of the problem solving strategy in the second part. With the rise of the computer and technology, hardware performance has been increasing at an overwhelming pace. Numerous development tools and programming languages are being changed at a similar fast pace. Despite constant changes in the development environment, similar problems are often encountered by programmers. This has created the concept of assimilating those resembling problems and proven solutions into design patterns. They are documented and read by programmers of each generation to help quickly solve known problems.

After the theoretical parts are considered, Last Planets implementation of the theory are described in the third part. This part includes considerable implementation techniques to deal with the limitation of the Unity3D development environment. Examples are provided to illustrate how maintainability and reusability are improved by applying design patterns. Due to the lengthy nature of source code, only short snippets are included.

The scope of this thesis is to provide specific knowledge about game programming and design patterns. Thus, a great number of related topics will not be discussed. Further study concerning such topics is recommended by reading the literatures mentioned in the references list.

Reading of the first and second parts does not require much technical background in game programming. However, existing knowledge about programming will be considerably helpful for understanding the implementations. Unity3D knowhow and object oriented programming knowledge are also valuable for getting the most of this thesis.

Since the history of the video game is relatively short compared to many other industries, there is an inherent lack of academic material related to this topic. Fortunately, the available materials related to programming in general are relatively bountiful. However, with the current exponential improvement in the hardware technology, the relevance of these materials might quickly become outdated.

2 Game Programming Overview

Since the advent of the video game with Pong in 1972, millions of computer games have been released worldwide. Unlike traditional games where mostly physical mechanism is involved, computer games rely heavily on the use of software technology to deliver enjoyable experiences.

In this chapter, game definitions will be examined from the available literature in the first subsection. Next, multiple game programming disciplines are introduced to help readers get familiarized with game components. In the third subsection, well known programming principles are explained. Lastly, the forth subsection gives out a summary about how development methodology affects game programming practices.

2.1 Definition of a Game

Numerous articles have been written about games and game programming. There are a galore of definitions voiced by numerous scholars. Despite such a quantity, there is still no conclusion drawn on a universally acceptable definition of a game [1,24]. However, by examining certain quantity of definitions, an understanding about game can be reached on an academic level.

Early academics such as Elliot Avedon and Brian Sutton-Smith (1971) have given the following definition: “Games are an exercise of voluntary control systems, in which there is a con-test between powers, confined by rules in order to produce a disequilibrial outcome.” [1,31]. Tracy Fullerton, Chris Swan and Steven Hoffman (2014) produced a slightly different opinion: “A game is a closed, formal system, that engages players in structured conflict, and resolves in an unequal outcome.” [1,43]. The author of the Art of Game Design, a book of lens – Jesse Schell (2015), summarized these definitions into a single condensed one: “A game is a problem-solving activity, approached with a playful attitude.” [1,47].

Even though a universally acceptable definition is lacking, academic researchers have agreed upon the elements of a game. These elements are discussed in the following subsection.

2.2 Game Elements

A modern video game consists of many elements such as graphic, sounds, scripts, story, AI. Most of them require a certain degree of programming. Thus there are many specialized groups of game programmers to cope with each element's specific requirements. This subsection introduces the most common elements and disciplines of programmers that have specialized at each of them.

Engine

In June 1996, Doom was released by id Software on PC and immediately gained huge popularity. The software components behind Doom were so well designed that id Software has gained a sizable additional income from licensing them to other developers. This advent has created the term "game engine" - software packages which allows the creation of multiple games. [1,11.]

The key characteristic of a game engine lies in the "data-driven architecture" that it apprehends to [2,11]. Through the process of generating game assets (art, sounds, etc.) and combining them with the game engine, developers can create a wide variety of games.

There exists however, a tradeoff between versatility and performance/optimization. The more varied types of games an engine can create, the lower the optimization for the games that were created. For example, an engine specifically designed for creating a first person shooter game would not be able to produce a real time top down strategy game. The problem in this case is the Level of Detail (LOD) techniques are very different from one genre to another.

Different genres require different characteristics from a game engine. A modern FPS game needs an efficient render system for a big 3D environment and sensitive control mechanism. Third-person games on the other hand demand a sophisticated camera handling system in order to not clip a player's view. Supporting a great amount of game entities is essential for real time strategy games.

Even though different genres of games have different requirements, the possibility of cross-genre engine development is created by the constant increase in computing power as well as steady breakthroughs in rendering algorithms and techniques [2,12]. There are a number of real time strategy games that are made from a first person shooter engine.

Artificial Intelligence

Game artificial intelligence traditionally constitutes a portion of software that gives an appearance of rational decision making. This term however has been used much more broadly to include most behavior generating codes. In some extreme cases, even movement and collision solving logic have been encompassed. [3,4.] Often, game development teams have to discuss and agree upon the scope of the term.

Various types of game genres require different types of artificial intelligence. Regardless, there are five fundamental characteristics for game artificial intelligence:

- The game artificial intelligence should be smart but also deliberately dumb occasionally.
- There should be no unintentional deficiency.
- The performance of the artificial intelligence should be within an acceptable range.
- Game designers should be able to fine-tune the artificial intelligence.
- The development cost of an artificial intelligence should not put the game at risk.

[4,522.]

Overtime, game programmers have collected a number of frequently used techniques for creating game artificial intelligence. Popular basic techniques include Finite-State Machines, Fuzzy-State Machines and Message-Based Systems. More advanced ones involve using sophisticated methods such as Genetic Algorithms and Neural Networks. Regardless, the choice of the technique being used is decided based on many factors like the game genre, platform, responsiveness, development constraints and entertainment values. [3,30.]

Audio

Audio has been an integral part of the game experience. The game audio source has evolved from MIDI sounds in the 80s to full lossless soundtracks in modern AAA games.

Although the medium of sound delivery has changed dramatically, the primary goal for an audio programmer remains the same: play sound at the correct time and in synchronization with art and animation. [5,36-46.]

The primary goal of correct timing may sound simple, yet it poses numerous challenges for programmers. Performance hiccups in other parts of the engine can cause audio and animation desynchronization. For 3D games and certain 2D games, the audio source might be behind a wall, thus multiple filtering techniques are used to simulate this obstruction [5,130]. Multiple sounds and music can play at the same time, creating a chaotic experience for players if the sounds are not in harmony or the extra sounds are not muffled by the engine [4,522].

User Interface

The game's user interface adheres to the common user interface rules. A well-developed user interface should perform exactly like the user thinks it should [6,8]. This point can be elaborated into a number of essential factors such as intuitiveness and responsiveness.

In order to help improve those two qualities, game programmers are often bound to implement plenty of input mechanisms and user interfaces. Such systems need to be able to allow tweaking and fine-tuning by game designer. At the same time, they must also have reasonable response time to user input.

Network

Most modern games incorporated some level of networking in their structures. There are multiple models for networking in games. Two most common models are peer-to-peer and client/server.

With the peer to peer network model, multiple clients connect to each other and there is no centralized game state manager. Thus the biggest advantage of this model is that there is no single point of failure. Any player can disconnect and the game will still be running. [7,81.] However, this model imposes a limit on the number of concurrent players. Due to the lack of centralized regulating power, the game is also prone to cheating and other security risks.

The client/server model on the other hand provides much improved security features. Since there are many clients connecting to servers, the number of clients can be scalable as much as the infrastructure allows. Maintaining such hardware however can be a considerable cost for running the game.

Development Tools

In order for the whole game development team to collaborate, some forms of development tools are necessary. Usually, the development tools come together with the game engine as a bundle. However, in most cases, the development team finds the need of modifying the tool to include customized or specialized functionalities. Certain engines (such as Unity3D) have taken the concept of re-using such modifications and packaged them into plugins for distribution.

The purpose of game development tool is to help developers ease their workflow [8,13]. Thus the main functions involve automating tedious tasks such as asset management and geometry handling. The tool for the programmer often uses some form of late binding techniques in order to quickly reassemble and provide a quicker feedback loop.

2.3 Programming Principles

Since game programming is a subset of software programming, it shares the common principles behind good software coding practices. The most common ones are code reusability, abstraction, single responsibility and loose coupling. This sub-section will explain each of them in detail.

Code Reusability

The purpose of code reusability is to repeat utilizing code assets in the course of development. There are two basic modes of reuse. Compositional reuse constitutes of recycling pre-used codes into developing new products. On the other hand, generative reuse is defined by using a software generator to create new application from a high abstraction state. [9.] In practice, the most common forms are software libraries, design patterns and application frameworks.

There are numerous benefits derived from having high reusability in software. The main biggest advantage is increase in productivity. Instead of having to develop the solution from scratch, programmers will be able to reuse a considerable number of existing codes, thus speeding up the development process tremendously. The process of changing or debugging would be also quicker because the centralized location of codes. Another sizable boon is that due to the repeat usage of software components, they will have higher quality [9]. This will result in higher product quality and fewer defects or risks to the development process.

Abstraction Principle

The concept of abstraction revolves around hiding implementation details and only exposes certain high-level functionalities [10,165]. The purpose of this is to shield programmers away from implementation details and allow them to focus on a single level of perception.

An example of an abstraction principle in normal daily life would be maps. Maps do not show every single objects in an area nor do them in the ratio of 1:1. Often, maps will rather show a drawing of roads, generalized blocks of buildings and the size of the area is scaled one to thousands. This would allow map readers to focus on direction tracking and route planning.

There are two types of abstraction mechanism: control abstraction and data abstraction s [10,165]. The formal one helps concealing procedural processes. Examples for this include functions and function parameters. Data abstraction on the other hand involves hiding complex data structure implementation away from programmers. This is implemented by the usage of classes, interfaces, abstract classes and many other methods.

Single Responsibility

The Single Responsibility Principle requires that there should be only a single reason to modify a code module [11,138]. This means that each class or function in the software should have only one purpose.

In practice however, this principle is often not followed. Programmers are bound by constraints of development time, thus they focus on getting the application to work rather than cleaning up and organizing the code base. Over stages of development, this neglect of reordering will lead to huge overly complex chunks of codes which in turn will slow down the process of debugging and modifying. Thus, the result of adhering to a schedule is actually increase in the cost of development time.

There are certain arguments against the single responsibility principle. One of the most common critics is that separation of codes into multiple files and functions would disallow programmers to grasp the big picture. This is wholly refuted by the fact that in order to fully understand a complex system, the amount of code to be read is the same regardless of refactoring. [11,149.] On the contrary, dividing responsibility reduce the amount of complexity that needs to be understood in order to modify only a part of the code base. Thus, a clear naming convention and single-responsibility classes would help understanding the code base enormously.

Loose Coupling

In software systems, the degree of how much specific software parts know of other parts is referred to as coupling. Tight coupling refers to systems where implementations are tied directly to implementations. On the contrary, loose coupling means that parts of the code bases only connect through the use of abstract interfaces. Thus they have no implementation knowledge of each other and can be swapped out easily on both compilation and run-time.

The main advantage of loose coupling is its flexibility in responding to changes [11,150]. If a component is directly referenced in numerous places, changing its implementation might cause domino-effect changes to other parts of the systems. On the other hand, if a component is only referenced by using an interface, it can be changed into another behavior entirely without disrupting any part.

3 Design Patterns

While obeying the principles mentioned in chapter 2, programmers have to write code that satisfy the requirements of the client and at the same time adhere to a specific schedule. With time, the development environment might change completely from one platform to another or even a different programming language. Yet, the problem and solution pair can be exactly the same in various project scenarios. Thus design patterns are recorded for the sake of reusing a high-level solution to specific problems [12,1].

In 1994, one of the first books about design pattern, Design Patterns Elements of Reusable Object-Oriented Software, was published. The book has quickly become classic literature about how design patterns are conveyed and also served as a standard catalogue. In this chapter, multiple design patterns based on that book will be discussed, along with a number of useful patterns for game development.

A design pattern consists of four main elements. Firstly, a pattern needs a name. A good name helps conveying the patterns will allow the discussion and improvement suggestion among programmers. Secondly, a problem must be stated for each pattern. This shows the contexts and states that should be recognized before applying the pattern. Next, a solution for this problem must be stated. The solution need not convey specific implementation but rather a generalized idea of how to solve the problem. Lastly, the consequences of the pattern should be described. By comparing the consequences of one pattern to another, a programmer can make an informed decision on which pattern to choose. [12,3.]

3.1 Strategy

The Strategy pattern makes use of a group of algorithms. Each algorithm is encapsulated within the group and made to be substitutable for another. Then in implementation, an algorithm can be selected based on the environment at runtime. [12,315]. The structure of this pattern can be seen from Figure 1 below.

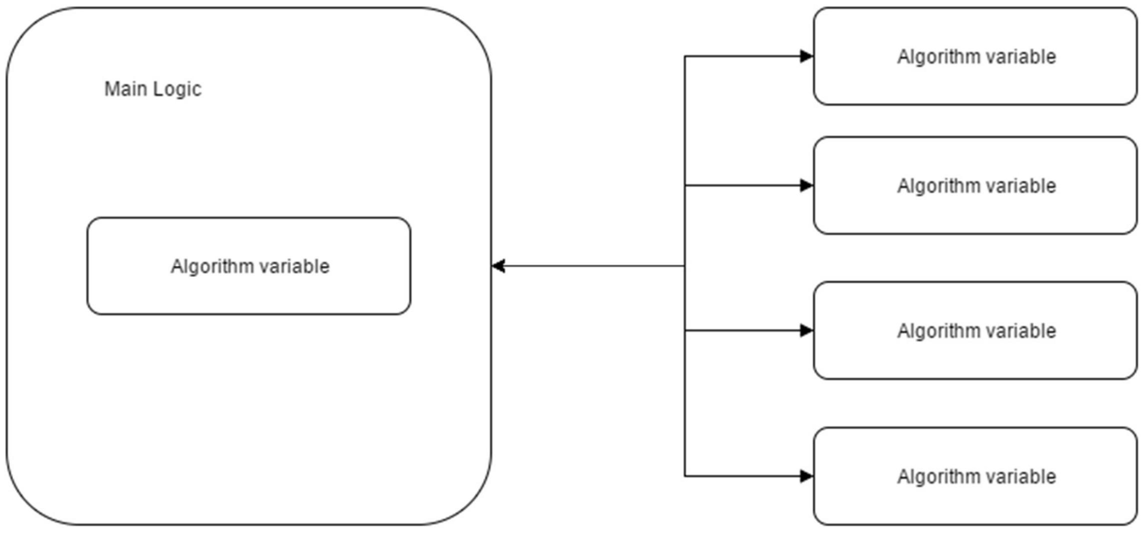


Figure 1. Strategy pattern diagram

The pattern has a number of applicable situations. Firstly, when different classes have a very similar structure except for certain behavior, such behavior can be encapsulated within a Strategy class, thus reducing the number of total classes. In practice, this portrays the favor of using composition instead of inheritance [13,37]. Secondly, if algorithm implementation needs to be abstracted and hidden from the main logic, the Strategy design pattern can be used to avoid uncovering of potentially sophisticated data structures. In another situation, one class might have many branching behaviors that are dependent on conditional switching. Such situation can be helped by using Strategy design pattern to refactor permutable types of logic into their own class.

3.2 Decorator and Abstract Factory

The Decorator pattern allows assigning additional behavior to an object without modifying its own implementation or inherits from its class [12,175]. The Decorator code should only modify or add lightweight functionalities, thus making the core behavior of the object still remaining the same to other components.

Usage of the Decorator pattern is recommended when there is a need to attach responsibilities to objects without involving other components. Such responsibilities are usually transparent enough to be disposable. In other cases, inheritance might be disallowed either because of sealed class or a huge number of additional functionalities. The Decorator pattern in those cases proves considerably desirable by encapsulating the additional functionalities within an extra class.

An example for the Decorator usage would be the case of adding visual effects to text on a graphical user interface. In this example, the text needs to have either a border or a shadow underneath, or both at the same time. The Decorator pattern in this case will suggest two Decorator classes that have reference to the text and add or remove these effects on runtime via the call of public functions. The abstracted implementation is presented in Listing 1 below.

```
static void Main()
{
    Text text = new Text();
    BorderDecorator borderDecorator = new BorderDecorator(text);
    ShadowDecorator shadowDecorator = new ShadowDecorator(text);
    // text with both border and shadow
    borderDecorator.Decorate();
    shadowDecorator.Decorate();
}

// ...

public class BorderDecorator
{
    private Text _text;
    public BorderDecorator(Text text) { _text = text; }
    public void Decorate() { /* Add border logic here */ }
}

public class ShadowDecorator
{
    private Text _text;
    public ShadowDecorator(Text text) { _text = text; }
    public void Decorate() { /* Add shadow logic here */ }
}
```
Listing 1. Example of the text Decorator implementation

Two major benefits of the Decorator pattern are its improved flexibility over subclassing and avoidance of cumbersome classes. Decorator pattern allows adding or removing functionality on run time unlike rigid inherited objects. An attribute can even be added more than once. Complex, lengthy classes can be avoided by adding functionality via the Decorator pattern along the progress of development.

Despite the strong benefits, Decorator pattern also has two major drawbacks. Firstly, the Decorator object might be confused with the actual augmented object. Secondly there might be a situation where numerous little decorators are created, making an overload of complexity. Such situation can make it hard for programmers to understand and debug the system. Furthermore, main purpose of the Decorator pattern is to modify slightly or add minor functionalities. Heavier changes or extensive modification are recommended to be done with the Strategy design pattern instead [12,179].

While the Decorator pattern provides a mean to implement small changes to objects, the Abstract Factory pattern seeks to create a vast number of similar or interlinked objects disregarding their specific implementation [12,87]. The true aim of the Abstract Factory pattern is separation. It promotes decoupling concrete implementation from higher abstract logic.

The Abstract Factory pattern is often used when the main logic is required be separated from how concrete objects are generated and controlled. Another use would be when multiple groups of similar objects are being maintained at the same time. Regardless, the usage commands that the implementation must be hidden and the object only reveal its interface.

By using Abstract Factory pattern, specific implementation is hidden from controlling classes. This results in two advantages. Firstly, switching generation algorithm is easy because of abstraction. Secondly, concrete factory classes have to adhere to the Abstract Factory design, thus maintaining high cohesion among objects created. However, there is one major drawback. New, totally different types of objects are hard to be implemented with Abstract Factory. Changing the set of interface for the Abstract Factory would require changes in all concrete factory implementation.

3.3 Observer and Chain of Responsibility

The Observer pattern provides mechanisms for one object to signal its changes to multiple other objects [12,293]. The object signaling would be called subject and the notified objects are observers. The subject could be considered a publisher of information. The observers therefore are subscribers to the information that the subject propagates. The relationship between the subject and observers is one to many. The model of this pattern is portrayed in Figure 2 below.

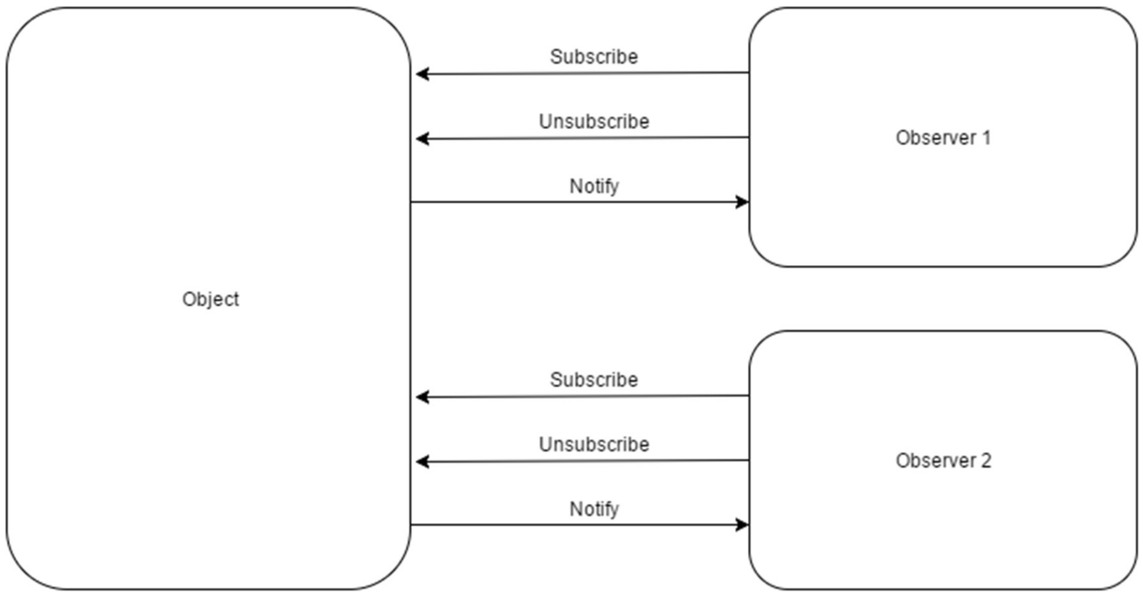


Figure 2. Observer pattern model

The Observer pattern is often used when change propagation among a number of objects is needed. Change happening to one object would cause change to others and the number of affected objects is flexible. In another case, the Observer pattern is used for the flexibility of subscribing. Observers can subscribe or unsubscribe on run time, thus making the implementation highly versatile and loosely coupled.

The primary benefit of the Observer pattern is the high degree of abstraction it provides between the subject and its observers. They each does not know of any implementation detail of the other, thus allowing them to be of any layer of abstraction in the program architecture. Another considerable advantage is the flexibility of subscribing. Observers cannot only freely subscribe or unsubscribe during application runtime but also choose to handle or neglect a change update. However, this allowance of freedom can lead to a major pitfall within the application architecture. The observers often do not have any knowledge of each other and the number of observers is also often unlimited. If the logic of several observers conflicts with each other, disastrous consequences could be hard to track down due to the sheer number of subscribers. In one example case, the observer upon handling an update, would cause the original subject to change, resulting in a never-ending loop.

Unlike the Observer pattern which broadcasts changes to multiple objects at same time, the Chain of Responsibility pattern passes a request through a group of ordered objects until one of them processes it [12,223]. In this pattern, the origin object which creates the request is not aware of the handler of its request. This defines the request handlers as an implicit receiver. The handlers have to implement an interface which allows calling of the successor handler in case it decided not to process the request and pass it along the chain.

The main situation where the Chain of Responsibility pattern would be used is when observers of an object need to have priority or follow a certain order. Inherently, this pattern still allows the same degree of flexibility as the Observer pattern when it comes to dynamically subscribing or unsubscribing to object changes.

Due to its similarity to the Observer pattern, the Chain of Responsibility pattern shares the primary attribute of high abstraction. The request senders and receivers have no knowledge of each other, thus they can be of any layer of abstraction. The receivers can also be detached or attached to the chain on run time, making the code base amply versatile. Compared to the Observer pattern, the chain structure negates the disadvantage of scattering receiver. However, the Chain of Responsibility pattern suffers from another weakness. Each request is only passing through objects in the chain until one of them handles it. Therefore, the rest of chain will not be able to receive the request at all. On the other hand, the request can pass through the whole chain without ever being handled due to misconfigured chain management.

3.4 Singleton

The purpose of the Singleton pattern is to guarantee that one class has only one instance and its access is available everywhere [12,127]. A global access point can be achieved via using a static global variable. However, such implementation does not ensure that there will be one and only one instance. The Singleton pattern will enforce this one instance policy by having the class itself monitoring all instance accessing.

The Singleton pattern is often used when there is a need for a sole instance of a class and a global access point of that instance. The pattern is also useful for situations where the global class needs to be inherited from. Listing 2 below illustrates one example of a usage case where the Singleton pattern is utilized to make database CRUD operation (Create, Read, Update, Delete) available globally.

```
public class Database
{
    private static Database _instance;
    public static Database Instance
    {
        get
        {
            if (_instance == null)
                _instance = new Database();
            return _instance;
        }
    }

    private Database() { /* initialization logic */ }
    public void Create() { }
    public Entry Read(string id) { }
    public void Update(Entry entry) { }
    public void Delete(Entry entry) { }
}
```

Listing 2. Database implementation using Singleton pattern

There are multiple benefits of using the Singleton pattern. Firstly, accessing to the single instance is monitored. Controlling logic can be written to regulate the behavior of accessing the instance. Secondly, the name space would be less encumbered by bloating amount of global variables. Thirdly, since the instance is instantiated from a class, it can be a subclass and modified in the inherited class. This would allow substantial customizability of the single instance behavior and increase reusability. However, the abused usage of the Singleton pattern can cause a similar problem to the usage of the static classes. Bloated and disorganized singletons can cause a heavy burden to code maintenance.

3.5 Adapter

The Adapter pattern helps changing one interface to another by providing middle conversion code [12,139]. This allows incompatible classes or frameworks to cooperate. The adapter class usually provides additional required methods from one interface to another, or encapsulate functionalities altogether. Figure 3 shows an example of how an adapter bridges the connection between two incompatible interfaces.

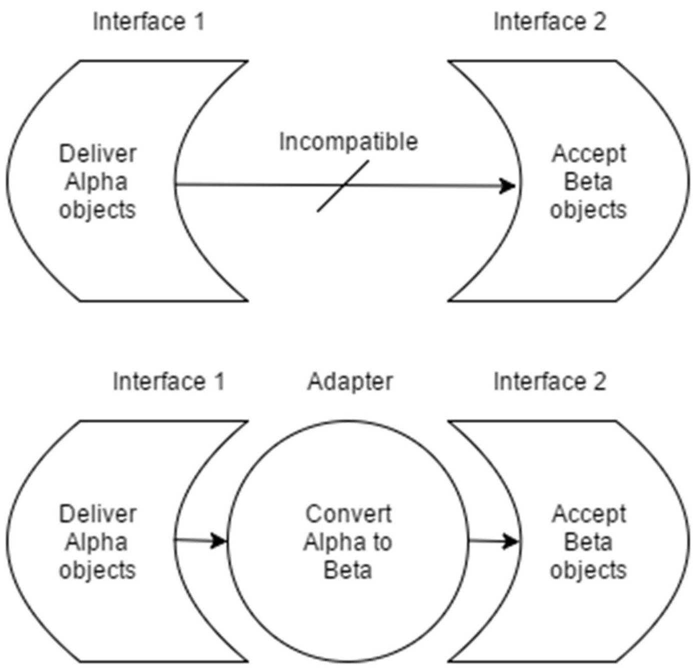


Figure 3. Adapter bridging two incompatible interfaces

The main use case of the Adapter pattern is when a group of incompatible classes needs to work together. A highly reusable class could also be created with this pattern for the purpose of adapting to most future changes. In the case of rigid inheritances, the Adapter pattern can help adapting multiple subclasses instead of changing the interface, which can be much more expensive to implement.

There are two main ways of applying the Adapter pattern: class adapter and object adapter. Depending on the implementation, each type can have different effects on the code architecture. The class adapter is written to adapt one specific class. This hinders the adapter from adapting subclasses but let the adapter override functionality of the adapted class. The object adapter can be configured to adapt multiple objects from different classes at the same time. However, this behavior would only allow the adapter to add functionality, not override existing ones.

3.6 Model-View-Controller and Thin Component

The Model-View-Controller (MVC) pattern refers to its three components. The model component handles the data manipulation and core logic. The view provides a display for the data in the model. Finally, the controller helps both the view and the model to communicate with each other. [14,530.]

The MVC pattern is a compound pattern – a combination of various pattern into a single pattern. The model's relationship with the view and the controller is implemented with the Observer pattern. Both the controller and the view subscribe to the changes published by the model. The controller represents the Strategy pattern. It can be swapped among a number of implementations to fit application details. Lastly, the composite pattern defines the view. Different components containing each other can be handled at the same time by the view. [14,532-533.]

User Interface implementation utilizes the MVC pattern heavily. The graphical presentation is handled with the view component. Numerous types of components like labels, panels, text fields, etc. are handled by its own view as well as a composited view combining them. The controller handles user input and interaction. Button clicks, input form submission and various other interactions are processed and relayed by the controller to the model. In the model, the core data of the application is manipulated by multiple logic functions such as creating, storing, loading and updating. After every change, the model would signal the view and the controller to update the graphical interface to the user.

Another common application of the MVC pattern is the network handling module. The model often refers to data package being sent and received. The controller is the routing service on server and the view is the final web content being displayed to users.

Since the influential inception of the MVC pattern from Smalltalk 80, a hefty amount of inspired and related patterns has merged. Among the most popular ones are the Model View Presenter used by IBM first in the 1990's [15]. It introduces the presenter as a replacement of the controller. The presenter can be considered a supervising controller and the whole architecture revolves around its centralized authority. Another widely used pattern is the Model View ViewModel which employs data binding robustly to handle interactions.

In game development and specifically the Unity3D environment, the Thin Component pattern derived from MVC to allow separation of a game object's presentation from its governing logic and data. It is developed by Alex Boyd and published to the public domain during Unite 2013. [16.]

Most classes in Unity3D are implemented by inheriting from the MonoBehaviour class, where logic is written onto specific hook functions like Start or Update. This allows a dynamic callback and runtime editing and tweaking in the editor. This feature is very useful for all members of the development team to see the result of their tweaking immediately and have a very short feedback loop. However, due to this imposing factor of MonoBehaviour, the logic can quickly become cumbersome for large-scale projects.

In order to counter this limitation, the Thin Component pattern tries to abstract the game logic from default functionalities of MonoBehaviour. This is achieved by writing the logic within non-Monobehaviour classes. These classes represent the traditional model in the MVC pattern. The next piece is the controller component, which inherits MonoBehaviour but have links to the model class. The controller component acts as a bridge for relaying the model's data information towards other Unity3D components such as Animator, SpriteRenderer and AudioSource. All these other components combined act as the view from the MVC pattern.

Direct benefit of this design pattern is that the logic is separated between the controller and the model. This in turn creates great flexibility within the code base. The standard controller component can be substituted with a tester component which implement the same interface. Such tester component will allow robust testing independently of model implementation. The logic model can also be substituted with a simulated logic model. This is useful in case of a client needing testing without a constant server connection. On the other hand, because the controller is hidden from the model, the same model logic can be run on both server and client without modification. The only changes will be in the controller.

The separation of the controller component and the model in the Thin Component creates many more classes and lines of code than a normal MonoBehaviour does. Thus, implementing this design pattern requires extra efforts at the beginning to design and execute. However, it can be argued that such efforts pay for themselves as the code base is much cleaner and easier to maintain and debug. Another disadvantage can be that the runtime variable viewing is no longer possible. Tweaking at runtime requires variables to be public in MonoBehaviour inherited class. Due to those logic variables being encapsulated within the model, they are no longer exposed to MonoBehaviour class, thus hiding them away from the editor. However, this limitation can be worked around by separating data variables into separated serializable classes.

4 Implementation of Last Planets

This section describes how the Last Planets project is developed and how the design patterns are applied to the source code of the project. There have been many components implementing various design patterns in this project. However, this section only discusses the most prominent ones.

4.1 Project Overview

Last Planets is a social mobile game developed by Vulpine Games studio from the beginning of 2014. The game has the typical feature of a social base defense game. Players have to defend their planets (bases) and improve their facilities in order to progress. They are also strongly encouraged to interact with other players in the same universe to form alliance, start raid and participate in events together.

The game was originally developed for both Android and iOS operating system. However due to development resource constraints as well as financial strategies, the team decided to focus only on an iOS version. The Android version of the game will be released later.

The game architecture revolves around a client-server connection. The need for a centralized server arises from the security need for the nature of social competitiveness of the game. Each part of the game structure, client and server will be discussed in the following sections. Figure 4 below displays a screenshot of the game.

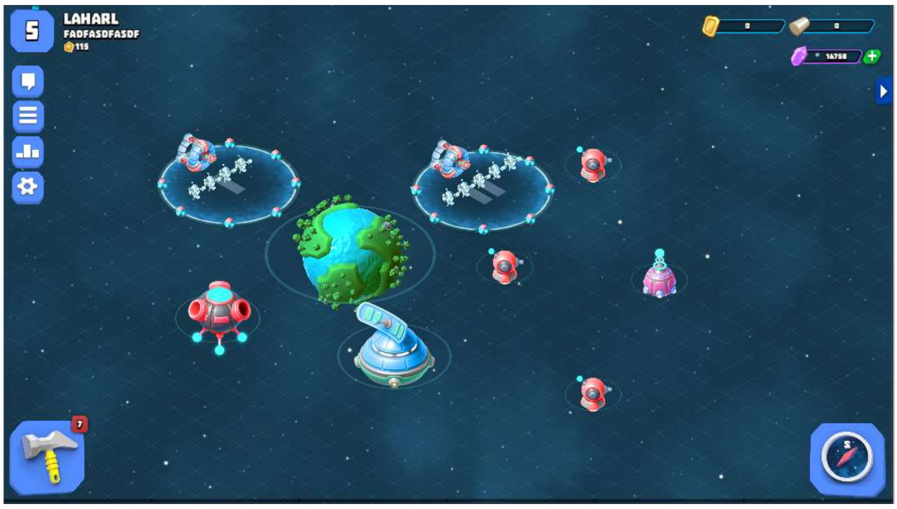


Figure 4. Last Planets screenshot

Client

The client side of Last Planets is developed using Unity3D game engine. Unity3D is chosen mainly due to its versatility in building for multiple platforms from the same code base. Another advantage of using Unity3D is that the engine allows an instant feedback loop, thus reducing game tweaking time. Lastly, the engine also features an asset store, allowing the purchase and usage of 3rd party plugins. These plugins help speeding up game development tremendously by reducing the need to write the same code from scratch as well as maintaining them.

Last Planets client is structured in two scenes: the start scene and the main scene. The start scene mainly focuses on initialization logic. Network connecting, logging player in, configuration checking and game asset loading are among the most crucial tasks of the start scene.

The responsibility of the main scene is to display all aspects of the game as well as handling the interaction of the player. The main scene is further abstracted into a number of states and has a dedicated system for switching among states to reduce the loading time between switching.

Aside from these scenes, there are numerous systems that persist across scenes and game states. These systems serve a number of global functionalities such as caching data and event listening. Such systems are configured with scripts preventing them from being destroyed when a scene loads or reloads.

Common

The common part of the project refers to the code base portion that is used by both the client and server. Contained within are the data structures, common logic and utility classes. These are components are vital to the game because they allow the communication between the client and server. They also reduce the amount of code by avoiding to write the same logic twice. Figure 5 below describes the relationships among these three parts of Last Planets’ code base.

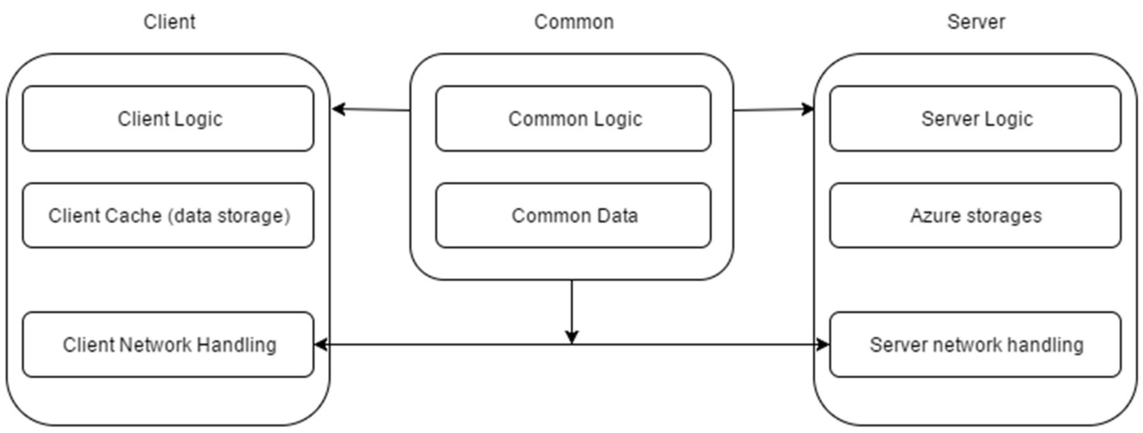


Figure 5. Client, common and server relationships

In current repository practice, the common is contained within its own repository. The client and the server repositories both include the common as a submodule within each. This submodule is only tracked via a commit hash, thus allow changing the version of common without knowing the exact details.

Server

The server side of Last Planets is organized into three major parts. The first part is the controlling and routing logic. These are deployed onto a login server and a proxy server. The goal of the login server is to handle simultaneously a huge number of concurrent connections. The proxy server on the other hand is responsible for routing the incoming transmission to the correct processing server.

The second part is the core game logic handling. This is separated into two servers: the synchronous logic server and the asynchronous logic server. Each of these two servers will handle the data and update the corresponding database. The asynchronous server does not require up to date information, thus no information is stored on this server. The synchronous server on the other hand requires real time processing of player events. Hence, it is implemented with dedicated thread and holds states about the game.

The system supports and monitoring attributes to the third part of server architecture. This includes the Chef server for deployment and configuration management and the monitoring server for checking the database and performance of servers. All three parts are presented in Figure 6 below.

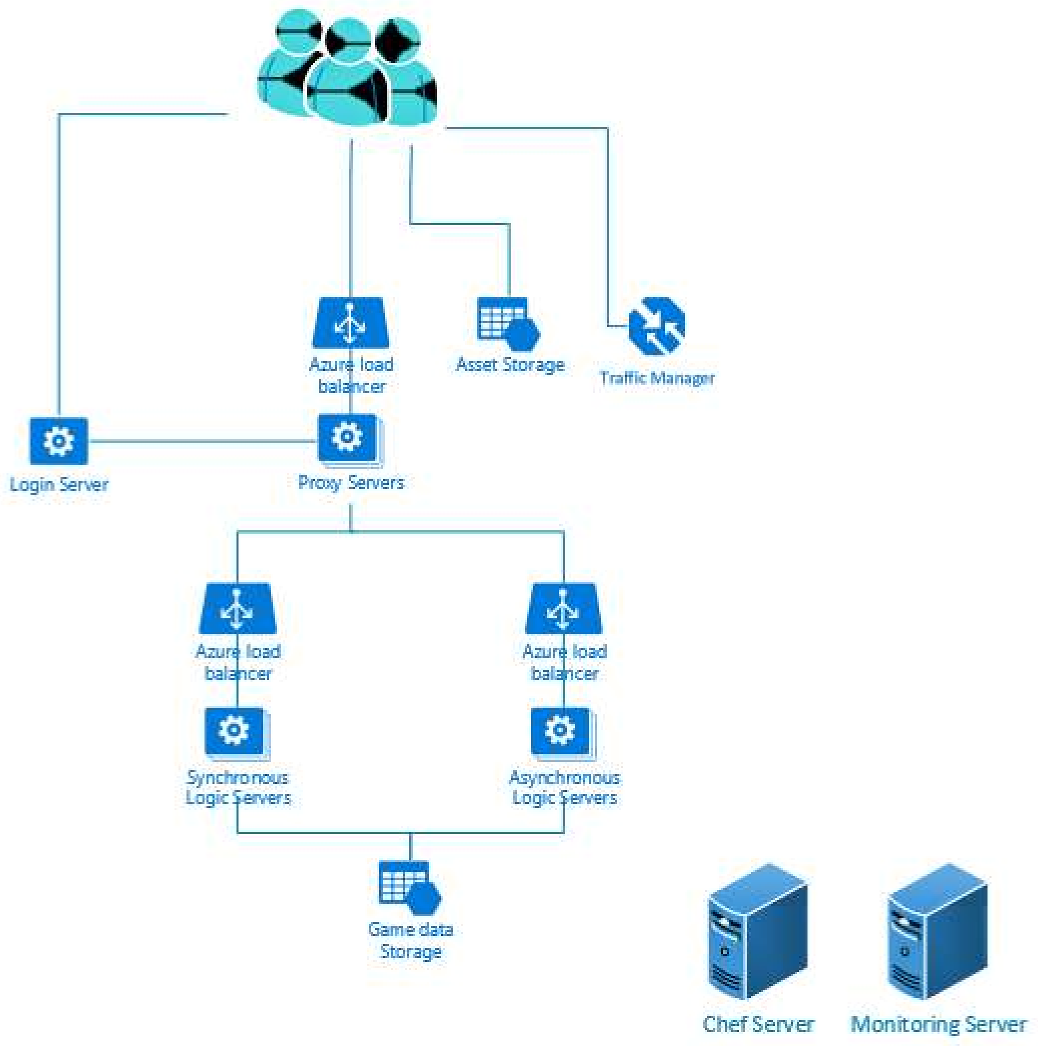


Figure 6. Last Planets server architecture

All servers in the architecture are deployed onto Windows Server virtual machine running in Azure Cloud. There are multiple Azure services to help with server maintenance such as load balancing, automatic scaling, automated software update and security policy. Furthermore, if any of the instanced virtual machines failed, it would be automatically recovered and replaced by a fresh instance.

Photon Server is chosen as the communication middleware framework for handling net-working transmission between the server and client. The framework has an extensive set of features and documentation. It is also easy to implement and it supports all popular mobile platforms including Android and iOS.

4.2 Design Pattern Application

This section presents how design patterns are applied into the code base of Last Planets. In some cases, the design pattern is found right in the beginning, before writing down the solution. However, in most cases, the design patter is found and applied later after a makeshift solution has already been applied. Thus, refactoring is essential during the course of development.

Unity3D Component

Unity3D's Component is one of the base classes in the architecture of the game engine. However, the Unity3D component pattern refers to how this Component-based architecture allows adding, removing and modifying behavior on runtime and on editor.

Unity3D organizes the game into scenes. Each scene has multiple game objects which can be modified by the editor. Each game object can attach multiple components or edit on runtime. This is possible because components implement serialization, allowing displaying properties on the editor. This also contributes to saving such properties to assets for hard disk storage.

There are three main ways of communicating among game objects with the Unity3D Component pattern. The first method is direct referencing using a public variable in a MonoBehaviour class. With this method, another component can be dragged and dropped to the specific slot on the referencing component. The target component can be an attached component in the current scene or an existing prefab component stored in assets. This referencing method is intuitive for a graphical user interface user. However, it needs access to the editor interface, hence it cannot be done during the time when the actual user plays the game.

The second way overcomes that limitation of play-time referencing. It utilizes Component built-in methods such as GetComponent and FindObjectOfType in order to get the referenced component. It can be made to run during editing time by using "ExecuteInEditMode" tag. On the other side, this referencing scheme suffers from null reference handling and performance drop when the scene has a huge number of objects.

Lastly, broadcasting messages is made possible using the default SendMessage and SendMessageUpwards. These functions take a method name in the parameter to invoke during runtime. This allows complete freedom of referencing because the sender does not need to know where the receiver is. Despite that, because the method name is a string, there is no guarantee that the receiver will actually call the method. Furthermore, this broadcasting scheme can be susceptible to errors and poor performance if not implemented properly.

Singleton

The Singleton pattern is used widely for the purpose of providing an access point to the global systems in Last Planets. There are three main types of Singleton implemented in the game: pure C# singleton, SingletonMonoBehaviour and SingletonMonoBehaviourAwake.

The pure C# singleton implementation adheres strictly to the requirements of the Singleton design pattern. It only allows a single instance at a time and provides one global access point to it. In order to improve reusability, a Singleton class was created with the type base approach for all pure C# singleton classes to inherits from. This reusability mechanism is also applied to the other two methods.

The SingletonMonoBehaviour class is vastly similar to the pure C# singleton implementation. It also ensures one instance at a time and one global access point to it. However, this class inherits MonoBehaviour, thus having all the member methods of it. Therefore, the constructor of this class has to be different. Instead of a normal constructor, it has to use the AddComponent method and reference the component gotten from it.

Lastly, the SingletonMonoBehaviour class is largely the same as SingletonMonoBehaviour. The only big difference is that the class no longer imposes the requirement of one single instance only. As a result, multiple instances can exist at the same time. The inherited class use OnEnable method to register itself as the current active instance. This behavior is particular useful in case of switching among game states. Different instances of the same class can be manipulated with different states while not modifying the other.

```
public class GameStateManager
    : SingletonMonobehaviourAwake<GameStateManager>
{
    public void SwitchState(GameState state)
    {
        /* switch state logic (enable objects, raise event etc.)
*/
    }
}

public class GalaxyMapButton
{
    void OnClick()
    {
        GameStateManager.Instance.SwitchState
                 (GameState.GalaxyMap);
    }
}

public class DoDamageAllianceAttackErrorHandler
      : DefaultErrorHandler
{
    protected override ErrorCode HandleOperationDenied
                                   (ErrorCode errorCode)
    {
        /* error logging logic */
        GameStateManager.Instance.SwitchState
                           (GameState.GalaxyMap);
    }
}
```

Listing 3. Singleton pattern implementation in GameStateManager

In Listing 3 above, the usage of GameStateManager with the help of the Singleton pattern is presented. The pattern permits a single instance of GameStateManager and calling its SwitchState method from anywhere in the code base. In this example, two separate classes of different functionalities and abstraction layers are calling this same method.

Observer

In order to counter the heavy performance cost and unregulated nature of Unity3D's SendMessage, a comprehensive Event raising system is developed for Last Planets with the Observer pattern in mind. The solution will allow registering to any event and raising events from any layer of abstraction within the code base.

The core classes of this solution are the Events and GameEvent classes. The former serves as the centralized API for raising, subscribing and unsubscribing to events, while the other acts as the template for all other events to inherit from. Underneath, the Event system utilizes dynamic invoking from the C# framework to raise the event handler based on the class type signature.

```
public class BuildingManager
{
    void CreateBuilding(BuildingType type)
    {
        Events.Fire(new CreateBuildingEvent(type));
    }
}

public class BuildingManagementSubscriber
{
    void Subscribe()
    {
        Events.AddListener<CreateBuildingEvent>(HandleEvent);
    }

    void HandleEvent(CreateBuildingEvent e)
    {
        /* handle creating new building logic */
    }
}

public class MetricsSubscriber
{
    void Subscribe()
    {
        Events.AddListener<CreateBuildingEvent>(HandleEvent);
    }

    void HandleEvent(CreateBuildingEvent e)
    {
        /* handle metric logging */
    }
}
```
Listing 4. Building creation event and subscribers

Listing 4 above illustrates one scenario in Last Planets where the Observer pattern is utilized. There are two subscribers registering to the same event, yet handling entirely different logics. BuildingManagementSubscriber is used in this case for submitting the creation event to the server, while MetricsSubcriber logs the event down with the purpose of aggregating these data later. This pattern allows limitless subscribers to hook to that same event. Furthermore, these subscribers can be unhooked on runtime. For example, the MetricsSubscriber can be disabled during the tutorial gameplay because no actual building is created.

Adapter

There are several external 3rd party Software Development Kits (SDKs) being used within the Last Planets code base. Each of them provides one or several functionalities and helps tremendously in reducing the effort to code such functions from scratch. However, they come with a risk of maintainability. The SDK can be outdated or discontinued by the original author. Thus replacing the SDK with either another SDK or internal code is needed. Additionally, the SDK might change its Application Program Interface (API) partially or entirely after a version update. All these potential changes pose a surmounted amount of risk to the project, especially the parts that are dependent on those SDKs.

In order to counter and mitigate these risks, the Adapter pattern is implemented to wrap each of the SDKs. The implementation of the SDK is abstracted and hidden away from functionality dependent classes. Thus, even if the SDK is changed or SDK's API is altered, the change will only happen at one place, which is within the adapter.

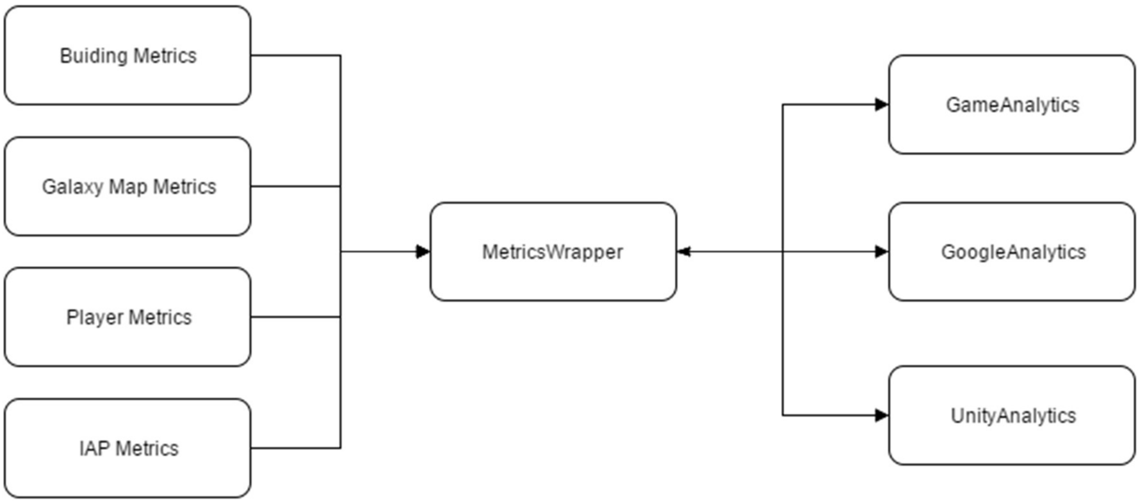


Figure 7. Metrics Wrapper Interaction.

The implementation of the MetricsWrapper class is done with the help of the Adapter pattern and can be seen in Figure 7 above. The MetricsWrapper class isolates the concrete implementation of metric SDK vendors from the actual logging logic in the main code base (Building Metrics, Player Metrics, etc.) During the project, multiple SDK changes were carried out effortlessly by just replacing the API within the MetricsWrapper class.

Decorator

Most Unity3D classes and components are packaged into binary files. Even though there is no possibility to modify or add functionality to them directly with this kind of complied library, the decorator pattern still allows adding extra behaviors indirectly. One example case is modifying the sealed class Vector2 of Unity3D. Listing 5 below presents how rotation behavior can be added to this class even though it is already compiled into binary.

```
public static class Vector2Extension
{
    public static Vector2 Rotate(this Vector2 v, float degrees)
    {
        float sin = Mathf.Sin(degrees * Mathf.Deg2Rad);
        float cos = Mathf.Cos(degrees * Mathf.Deg2Rad);

        float tx = v.x;
        float ty = v.y;
        v.x = (cos * tx) - (sin * ty);
        v.y = (sin * tx) + (cos * ty);

        return v;
    }
}
```
Listing 5. Vector2 extension method

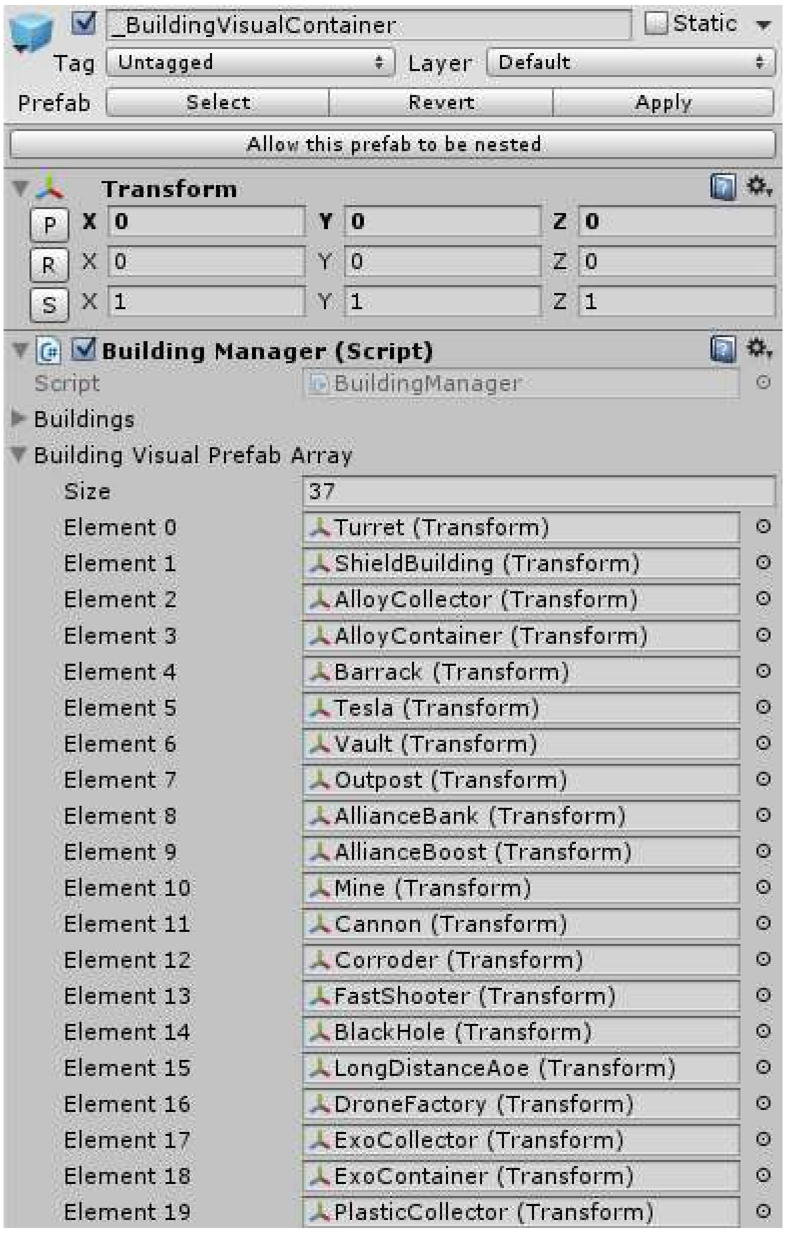
C# extension methods are implemented with the decorator pattern in mind. For another example, the extra method SetState in Last Planets allows the state toggling of MonoBehaviour and game objects with a null checking logic. These extension methods are implemented with syntactical disguise as a member method in order to help readability.

Another benefited situation is when the pure client and pure server logic need to be separated. Both logics cannot be included in the common part because each does not need to be known to the other and compiling these logics would require additional libraries. The extension methods allow these logics remain purely either client or server and depend solely on the corresponding repository.

Factory

The Factory pattern is used in a number of places in order to generate objects. Each factory is managed in a factory class and a manager class. The factory class holds the generation logic, which instantiates objects based on the referenced asset prefabs. The manager class is a singleton that has a link to the factory class. It serves as a layer of abstraction away from generation logic as well as a single point of access.

Pooling is also implemented for most factory classes. It allows recycling of unused objects and avoiding of calling the heavy Instantiate method again, thus reducing a considerable amount of performance load. In most cases, factories use a generic implementation of the Pool class. However, in several special cases, a custom reset logic needs to be applied.



```
_BuildingVisualContainer                Static
Tag    Untagged            Layer  Default
Prefab  Select   Revert   Apply
Allow this prefab to be nested

Transform
P  X 0   Y 0   Z 0
R  X 0   Y 0   Z 0
S  X 1   Y 1   Z 1

Building Manager (Script)
Script                 BuildingManager
Buildings
Building Visual Prefab Array
  Size         37
  Element 0    Turret (Transform)
  Element 1    ShieldBuilding (Transform)
  Element 2    AlloyCollector (Transform)
  Element 3    AlloyContainer (Transform)
  Element 4    Barrack (Transform)
  Element 5    Tesla (Transform)
  Element 6    Vault (Transform)
  Element 7    Outpost (Transform)
  Element 8    AllianceBank (Transform)
  Element 9    AllianceBoost (Transform)
  Element 10   Mine (Transform)
  Element 11   Cannon (Transform)
  Element 12   Corroder (Transform)
  Element 13   FastShooter (Transform)
  Element 14   BlackHole (Transform)
  Element 15   LongDistanceAoe (Transform)
  Element 16   DroneFactory (Transform)
  Element 17   ExoCollector (Transform)
  Element 18   ExoContainer (Transform)
  Element 19   PlasticCollector (Transform)
```

Figure 8 Building Manager inspector view.

In Figure 8 above, the snapshot of Unity3D inspector view is taken for a BuildingManager class. The class represents how Factory pattern is implemented. The building visual prefab array contains blueprints objects for instantiating at runtime. Since the place of storing all these blueprints is centralized within one class, the actual generation logic can confidently instantiate a new object by type instead of having direct reference to the building of that type.

MVC

MVC is a particularly useful for solving user interface architecture. Last Planets' user interface system thus takes advantage of this design pattern heavily. The implementation of MVC is done in two levels: panel and entry.

Most user interfaces in Last Planets are displayed in panels – the section container of a whole functionality often takes the role of a menu or a popup. The panel presentation often constitutes a title, a background and content including buttons, icons, labels, and possible entries. The model in this case is often data common classes taken from Common portion of the code base. A separated controller is coded as a pure C# class to handle storing and getting the common data fetched from the server or generated from configuration classes. A MonoBehaviour class is attached to each panel prefab to act as the view. It often accesses the data from the controller when OnEnable is called and displays this data correspondingly.

On the other hand, the MVC on the entry level deals with the subcomponent of the user interface. Each entry usually represents one small repeating part such as one column in a list or one cell in a grid. Because entries usually do not exist separately, the controllers in most cases are the also the views of the panels. These views when displaying data pass those data either directly or wrapped within the container class – the model – to each entry's view. The view of each entry is a MonoBehaviour class that displays the received data to each referenced user interface components.

Figure 9 below presents a typical view of one panel and multiple entries. This Daily Reward panel consists of a title, a background frame, a grid of entries and a confirm button. There is also a transparent collider in order to block out interaction with other user interfaces or game elements while this panel is active.

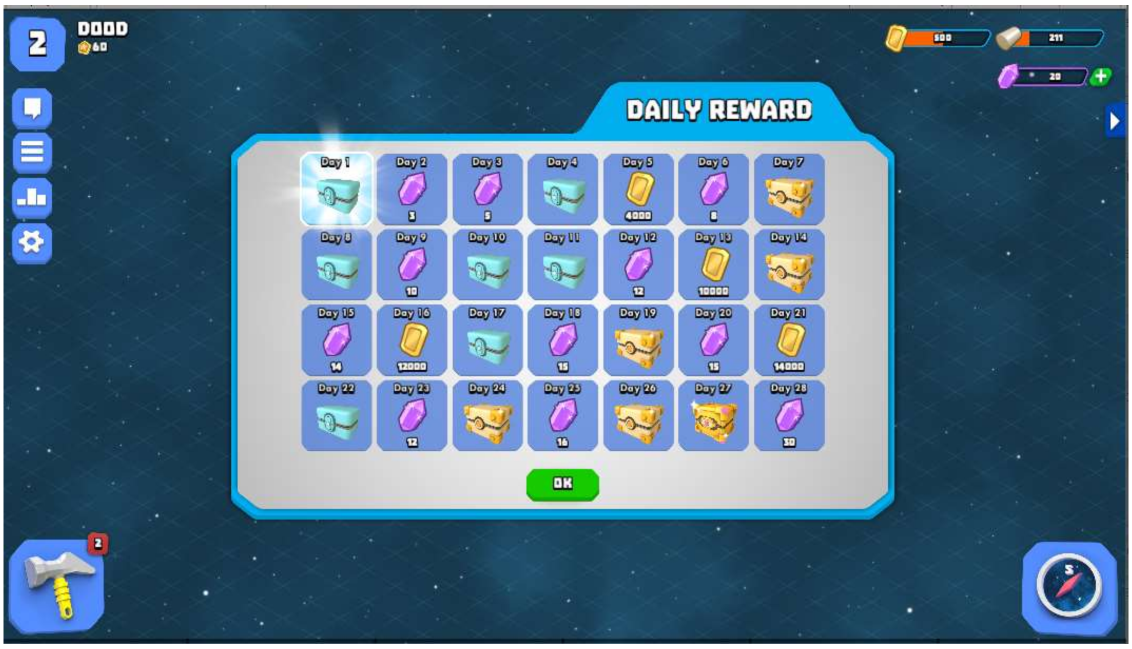


Figure 9 Daily Reward panel and entries.

Thin Component

The Thin Component pattern is implemented mainly in the combat system of Last Planets. The combat system utilizes this pattern by implementing two parts: the model and the controller component. The model is created as a pure C# class. This model has functions that define the behavior of certain objects. The controller component on the other hand contains references to MonoBehaviour and other Unity3D components (SpriteRenderer, Collider, etc.) It routes the command from the model to change specific a view component accordingly. The two parts are linked together via interface referencing.

By using interfaces, the specific implementation of the controller component is not specified within the model class but referenced via the interface. It is only determined at initialization time by a manager script overseeing the whole combat on the client. Decoupling the controller from the model allows changing the controller implementation depending on the situation or environment.

This implementation of the Thin Component pattern allows replicating a combat session without dependence on a specific library. Thus, the server can validate the client's replay data without including the Unity3D library.

5 Discussion

Last Planets has been in development for over one year. During the development process, one internal testing version was always available for internal testing. The interval team meetings were held to discuss improvements for the project. Through such meetings, the following findings and improvement suggestions were drawn and they will be presented in the following subsections.

5.1 Findings

Unity3D relies on a Mono subset .Net framework, which is at the moment version 2.0. However, this is different from the standard .Net framework used in the server architecture, which is version 5.0. This difference caused several discrepancies between the behavior of the same class in the client and server. In order to solve this problem, the common part of the project was made to store also the essential logic. Furthermore, wrappers were built to abstract out or replace changing framework modules.

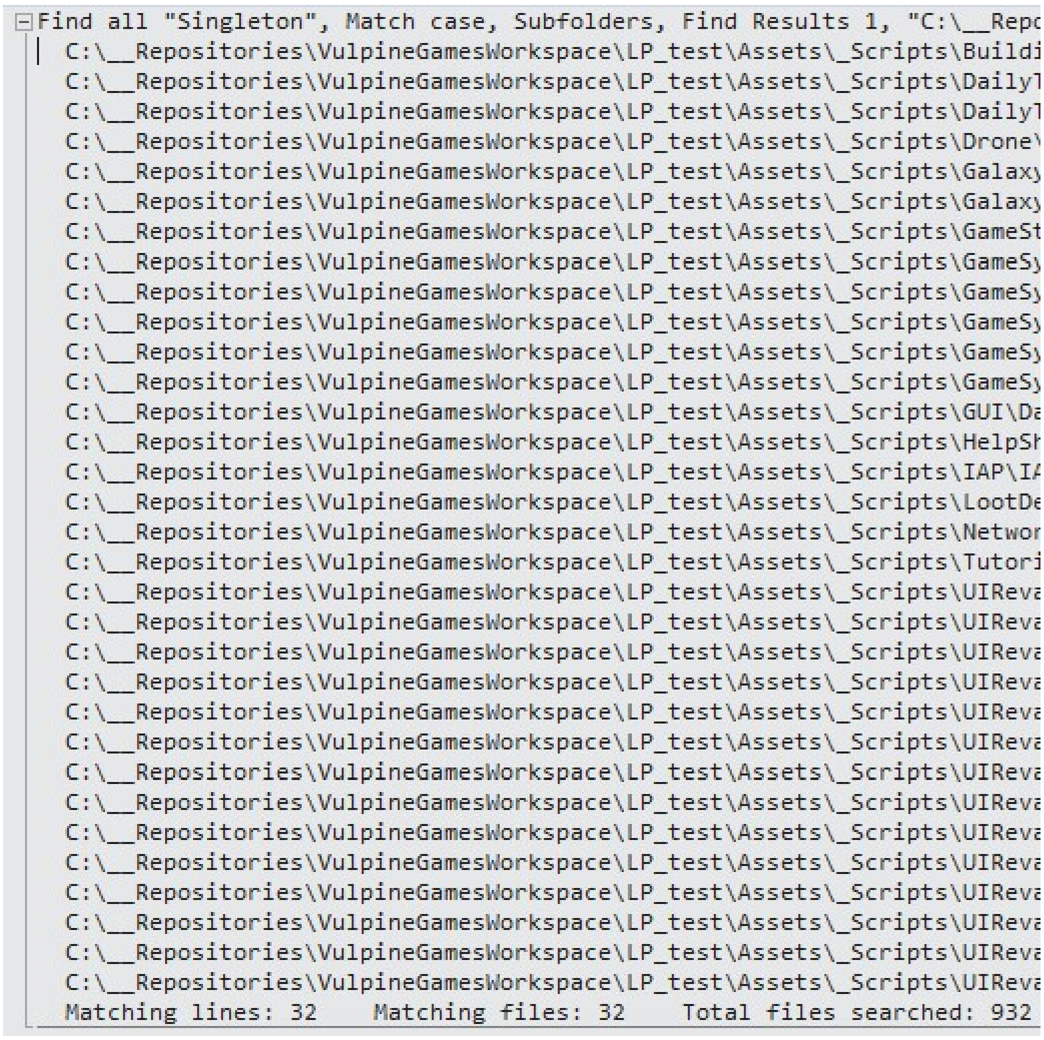
An automated building and deployment system was created to facilitate a fast feedback loop. The automated build automatically retrieves new changes to the project and builds them into binaries. These binaries are uploaded to the corresponding HockeyApp distribution channels for all team members to download and test.

The usage of interfaces proved to be tremendously helpful during the project. By abstracting out concrete implementation, multiple classes could be used interchangeably. The code base has become considerably more reusable and readable.

Design patterns were not applied from the initial phase of development, but rather implemented as a gradual process. The usage of patterns was applied continuously during the project as they were realized. In some cases, massive overhaul refactors were carried out in order to implement the pattern as well as clean up the code base.

5.2 Suggestions for Future Study

As can be seen in Figure 10 below, the number of singletons existing in the project is over 30. This number is too high and causes several troubles for maintaining the project as they are referenced in numerous places across the code base. In addition, the cluttered logic also hampers efforts in understanding and debugging. One particular solution to this might be the Inversion of Control principle. The usage of pattern such as Dependency Injection may help reducing various scattered singleton references.



```
⊟Find all "Singleton", Match case, Subfolders, Find Results 1, "C:\__Repc
 |  C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\Buildi
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\DailyT
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\DailyT
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\Drone\
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\Galaxy
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\Galaxy
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\GameSt
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\GameSy
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\GameSy
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\GameSy
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\GameSy
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\GameSy
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\GUI\Da
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\HelpSh
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\IAP\IA
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\LootDe
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\Networ
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\Tutori
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\UIReva
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\UIReva
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\UIReva
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\UIReva
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\UIReva
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\UIReva
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\UIReva
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\UIReva
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\UIReva
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\UIReva
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\UIReva
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\UIReva
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\UIReva
    C:\__Repositories\VulpineGamesWorkspace\LP_test\Assets\_Scripts\UIReva
    Matching lines: 32    Matching files: 32    Total files searched: 932
```

Figure 10. Singleton search result in Last Planets project

The event handling system has a number of duplications of logic. In order to load a certain scene, multiple server queries have to finish before the loading animation is dispelled. The queries, however, are different from one scene to another. Therefore, a repeated pattern of waiting for queries is scattered among scene loader classes. The usage of the Template pattern might be valuable for this problem. Assembling loader on runtime and registering only scene relevant queries can reduce this duplication considerably.

This thesis has introduced only a very small number of design patterns. Readers are strongly recommended to study design patterns to apply them in their own projects. The materials listed in the reference list are all of high value.

6 Conclusion

Overall, the project succeeded in delivering Last Planets, the social mobile game into the market. The design patterns were applied widely in the code base and improved maintainability as well as readability substantially. However, the applying process was not instantaneous but rather a constant change of refactoring and gradual pattern implementing.

Various improvements were done with the help of design patterns. The Adapter pattern allows separation of SDK and the dependent code logic. The Observer pattern proves tremendously useful with event notifying and event handling. The user interface benefits significantly from adapting the decoupling logic of MVC design pattern. The Thin Component creates an extensive flexibility for the combat system and the possibility of validating it on the server side.

However, overuse or misuse of design patterns can be troublesome for the project. The surplus use of the Singleton pattern has overcomplicated the code base. This case of over-usage can be mitigated with another design pattern instead, such as dependency injection. Furthermore, numerous classes in the project are still unorganized or using ghetto implementation. Such classes can be cleaned up substantially by applying the correct design pattern.

Since the introduction of design patterns to programming by the classic book Design Patterns Elements of Reusable Object-Oriented Software, numerous patterns have appeared and their applications have been well documented. This thesis only managed to present a miniature amount of them, hence readers are well recommended to study more design patterns and even create ones on their own.

References

1 Jesse Schell. The Art of Game Design: A Book of Lenses 2nd Edition. CRC Press Taylor & Francis Group; 2015.

2 Jason Gregory. Game Engine Architecture. Boca Raton CRC Press; 2014.

3 Brian Schwab. AI Game Engine Programming. Charles River Media; 2004.

4 Steve Rabin, Introduction to Game Development. Second Edition. Course Technology PTR; 2009.

5 Steve Horowitz and Scott Looney. The essential guide to game audio: the theory and practice of sound for games. Focal Press; 2014.

6 Joel Spolsky. User interface design for programmers. Apress L.P; 2001.

7 Todd Barron. Multiplayer Game Programming. Course Technology; 2001.

8 Ardolino Alessandro, Alan Chaney, Marc Schèarer, Marc Schèarer. Game Development Tool Essentials. Apress; 2014.

9 Lombard Hill Group. What is Software Reuse? [online]. Lombard Hill Group. URL: http://lombardhill.com/what_reuse.htm. Accessed 13 April 2016.

10 Alen B. Tucker. Programming Languages Principles and Paradigms. McGraw-Hill Higher Education; 2007. 2nd ed.

11 Robert C. Martin. Clean Code A Handbook of Agile Software Craftmanship. Prentice Hall; 2008.

12 Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley; 1994.

13 Steve Holzner, PhD. Design Pattern for Dummies. Wiley Publishing; 2006.

14 Eric Freeman. Head First Design Patterns. O’Reilly; 2004.

15 Martin Fowler. GUI Architectures [online]. ThoughtWorks; 2006. URL: http://martinfowler.com/eaaDev/uiArchs.html. Accessed 13 April 2016.

16 Alex Boyd. Thin Component Controller Pattern. Software Design Pattern for the separation of Model and Controller within Unity3d framework [online]. 2013. URL: https://docs.google.com/document/d/1BPDPe_9c1rx1Vvp7WHqmkn9QbgDyKoo91bN43CiofQo/edit. Accessed 13 April 2016.