Tampereen ammattikorkeakoulu

Inversion of Control in Game Development

Strange IoC

Jerri Ahonen

BACHELOR'S THESIS
November 2019

Business Information Systems
Games Academy

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Pelituotanto

AHONEN, JERRI:
Inversion of Control in Game Development
Strange IoC

Opinnäytetyö 44 sivua
Marraskuu 2019

Pelien tekeminen on pitkä ja jatkuvasti muuttuva prosessi. Hauskaan ja mielenkiintoiseen pelikokemukseen pyrkiminen vaatii jatkuvasti muutoksia pelin logiikkaan ja sen ominaisuuksiin. Tavoitteena oli selvittää, kuinka Inversion of Control -kehyksen käyttö parantaa ylläpidettävyyttä sekä helpottaa mobiilipelien kehitysprosessia. Työssä kerrottiin, miksi juuri Strange IoC on valittu, kuvattiin sen toiminnallisuutta ja tutkittiin sen eri ominaisuuksien hyötyjä ja haittoja. Greener Grassilla toteutettiin haastattelu selvittämään Inversion of Control -kehyksen liittyviä päätöksiä ja niiden syitä, siihen tehtyjä muutoksia ja sen käyttöperiaatteita.

Strange on rakennettu Unitya varten. Unity on suosittu 2D- ja 3D-pelimoottori, jota Greener Grass käyttää. Strange on oiva vaihtoehto käytettäväksi Unityn kanssa, vaikka nykyään on muitakin vaihtoehtoja. Inversion of Control -kehyksen käyttämistä suositellaan käsiteltäessä suurempaa projektia, johon kuuluu paljon näkymien ulkopuolista logiikkaa, ja projekteja joista tehdään useita versioita usealle alustalle. Toisaalta, jos projekti on puhtaasti visuaalinen demo, jossa ei tarvitse tilan tallentamista, näkymistä irrotettua logiikkaa tai ulkoisia palveluita, Inversion of Control -kehyksen käyttöä ei suositella.

Inversion of Controllin käyttämiseen siirtyminen perinteisemmästä lähestymistavasta kontrollin virtaukseen koodissa voi aluksi tuntua ylitsepääsemättömältä. Oppimiskäyrä on jyrkkä kokonaan uutta ajatusmallia luokkien rakenteeseen ja niiden vastuiden määrittämiseen sovellettaessa. Yrityksen joka palkkaa ohjelmoijan jolla ei ole aikaisempaa kokemusta Inversion of Controllista tarvitsee tukea työntekijää alkuun pääsemisessä ja tarjota tarvittava määrä ohjausta uuden konseptin oppimisessa.

---

pelinkehitys, inversion of control, strange ioc

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Business Information Systems
Games Academy

AHONEN, JERRI
Inversion of Control in Game Development
Strange IoC

Bachelor's thesis 44 pages
November 2019

Making games is a long and constantly shifting process. Pursuing fun requires frequent changes to the logic and features of the game. The objective of the thesis was to study how using an Inversion of Control framework can improve maintainability and facilitate the process of making mobile games. The purpose is to explain why Strange IoC was chosen for the task, to explain its functionality and to study the benefits and disadvantages of its various features. An interview was conducted at Greener Grass to shed light on decisions taken around the usage of an Inversion of Control framework, modifications made to it and its usage principals.

Strange is purpose built for Unity, the popular 2D and 3D game engine used at Greener Grass. This makes it an excellent choice as IoC container for Unity, even though other options are also available today. Using an Inversion of Control framework is recommended when dealing with a larger project that contains much logic outside the views and with projects that involve many different platforms and variations. On the other hand, if a project is purely a visual demo that does not need state saving, logic outside of views or external services the usage of IoC is not recommended.

Changing to using Inversion of Control coming from a more traditional flow of control approach can be overwhelming at first. The learning curve is steep and a whole new ideology around building classes and thinking about their purpose and responsibilities needs to be adapted. This transition needs to be facilitated by companies hiring programmers that do not have prior experience of Inversion of Control frameworks.

---

game development, inversion of control, strange ioc

CONTENTS

TERMINOLOGY

IoC	Inversion of Control.
IoC container	A framework for implementing automatic dependency injection.
API	Application Programming Interface.
C#	A widely used, general-purpose programming language.
GameObject	Base class for all entities in Unity scenes.
Component	Base class for everything attached to GameObjects.
Feature	A feature of a game, such as collectable cosmetics or a minigame.
Module	An independently functioning part of an application.
Prototype	Smaller than a demo, used to test possibly interesting mechanics and concepts.

1 INTRODUCTION

Greener Grass is a Finnish game development studio formed in 2015. They develop brand new titles from concept to release with an eye on easy updating and maintenance. Keeping a game easy to update and modify all while making new features is not an easy task. From a technical point of view the code should be modular and it should be easy to swap or entirely remove features on the go. The constant changing and evolving aspect of the gaming industry brings many challenges to game making.

Many developers and companies tackle these difficulties every day. In 2018 the App Store and Google Play Store offered over 1.5 million games in total (Gough C. 2019a, Clement J. 2019). In comparison at that time Steam offered approximately 30 000 games (Gough C. 2019b). While it is nearly impossible to tell the exact number of games on different platforms, this does give a perspective on the number of titles being made all the time. With all these games comes plenty of updating and bug fixing, constantly modifying the logic behind it all.

Figure 1. Global games market (Mobvista 2018)

According to Bonfiglio (2018), approximately half of mobile games are made with Unity, a game engine used to build both 2D and 3D games. Developers using Unity range from young hobbyists to game industry professionals. Unity offers both a free version and a paid Pro-version. Being so popular, one could think Unity is the perfect solution for developing and maintaining game projects. However perfect solutions do not exist.

The mentality between developing small titles or bigger scale games is fundamentally different as will be discussed in the following chapters. When aiming for a larger, more engaged audience, many different aspects need to be taken into consideration: many features to keep the game interesting, long time support and updates to keep players returning and constant polishing and bug fixing to assure the best gaming experience possible.

This bachelor's thesis aims to delve deep into the technical side of making mobile games. Chapter 2 discusses common difficulties in game development. Chapter 3 takes a look at Unity's offerings out of the box and how they can be expanded and improved upon. Chapter 4 introduces the concept of Inversion of Control frameworks and their basic functionality. Chapters 5-9 go into detail on why Strange was chosen at Greener Grass, different features of Strange and whether or not they should be used.

This thesis was commissioned by Greener Grass Oy.

2 DIFFICULTIES IN GAME DEVELOPMENT

In his thesis “Dependency Injection in Unity3D” (2017) Parviainen covers differences between software development and game development. He brings up an answer on stackoverflow.com which has the following arguments.

Software is always designed to fulfil a business need. Because of this, it is possible to list tasks that need to be performed to accomplish this need. With games, this need is fun. And writing a technical specification for fun is not exactly a straightforward task. (How is game development different from other software development? 2011.)

This leads to constant changes to the end goal of the project, as the game gets tested and iterated over and over again. In order to find the sweetspot that clicks with the audience and makes the game fun, many features need to be shifted around, or dropped completely.

In 2009, a survey on problems in game development was conducted and an article was written on the findings. In this article Petrillo, Pimenta, Trindade and Dietrich survey the problems in the development process of electronic games. The results were mainly collected from game post-mortems. One of the findings brought up in the article goes as follows:

> Cutting Features During Development. As well as feature creep there is another problem that is mentioned frequently: that of cutting features during the development process. Due to the fact that projects initiate an overly ambitious number of functionalities, some of which are even implemented, but for a variety of reasons they end up being cut further on in the project. (Petrillo, Pimenta, Trindade & Dietrich 2009.)

In his thesis, Parviainen brings up the case of Diablo 3 by Blizzard Entertainment, where Blizzard had to make drastic changes to the game even after its release. According to Hight (2013) originally the game included an auction house which purpose was to provide a convenient and secure system for trades. However, after the game’s launch the auction house received a lot of criticism from the

players. As a result, Blizzard solved this backlash by completely removing the auction house from the game.

One key principal used in coding to help in these situations is called loose coupling. The opposite is called tight coupling. According to Durand (2013) “Tight coupling, also known as strong coupling, is a generalization of the Singleton issue. Basically, you should reduce coupling between your modules. Coupling is the degree to which each program module relies on each one of the other modules.” In our example of Diablo 3, the auction house acts as one of these modules. By having it operate as an independent part of the game, Blizzard was able to extract it with reasonable effort.

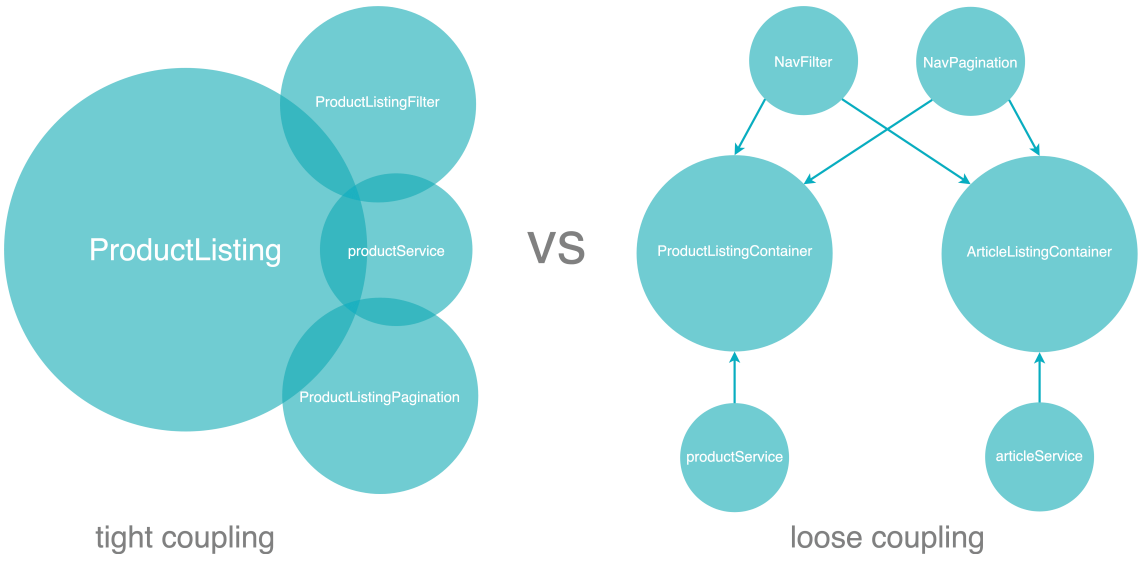

Figure 2. Tight coupling versus loose coupling (Oberlehner M. 2019)

3 CURRENT TOOLS IN UNITY

3.1 General

Unity is a widely used game engine first announced in 2005. Unity gives the ability to create games both in 2D and 3D. It offers a primary scripting API in C#. As of 2018, approximately half of new mobile games on the market have been made with Unity. (Bonfiglio 2018).

Unity is very small-project friendly. Most of its code dependency and flow management structures favour small, almost prototype-like games. They are fast to create and very useful when a developer needs something done quick. It is when a project grows larger and needs to be maintained by a bigger team that things start to fall apart with this approach.

There is not much to choose from when it comes to managing dependencies in Unity. It does provide basic functions like `GameObject.Find` and `GameObject.FindObjectOfType`, although it is not recommended to use these methods. The reasons for this will be expanded on later in the thesis. Unity also offers a basic Singleton pattern, which too has its drawbacks.

3.2 The inspector

One way to manage object referencing within Unity is to use the inspector window. Public fields can be declared in a MonoBehaviour (Figure 3). MonoBehaviour is the base class from which every Unity script derives. Once the script is attached to a GameObject in Unity, the public fields are displayed in the inspector window in the script component (Figure 4). The wanted reference can then be simply dragged from the hierarchy into the slot in the inspector (Figure 5).

Assets > GameManager.cs > ...

```
1   using UnityEngine;
2
3   public class GameManager : MonoBehaviour
4   {
5       public GameObject player;
6
7       // Start is called before the first frame update
8       void Start()
9       {
10          // Do something with our player
11      }
12  }
```

Figure 3. Declaring a public field in GameManager for the player GameObject

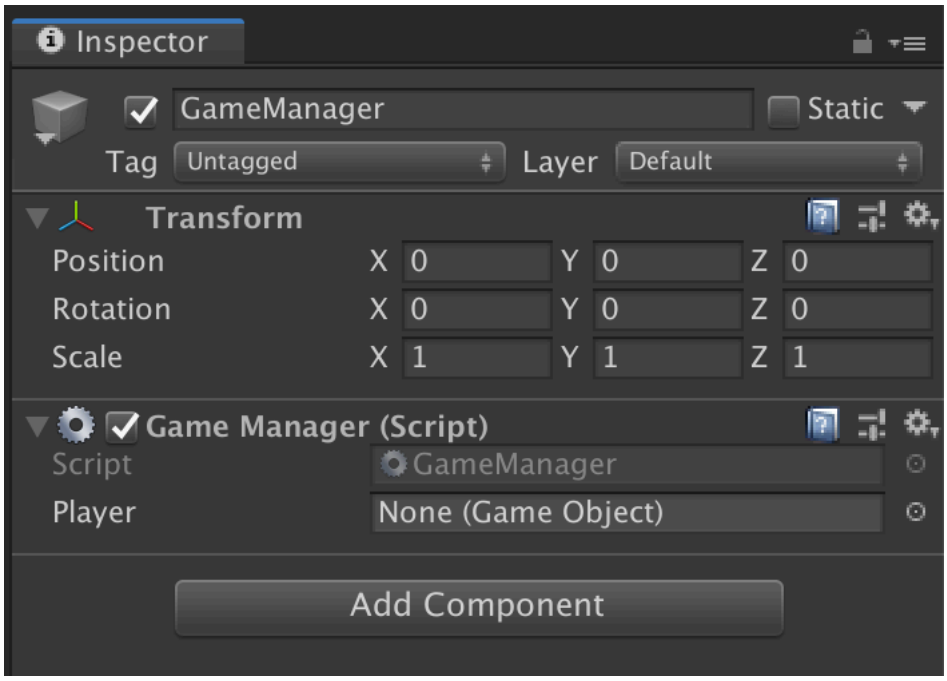

Figure 4. The inspector displaying a field in GameManager for the player GameObject

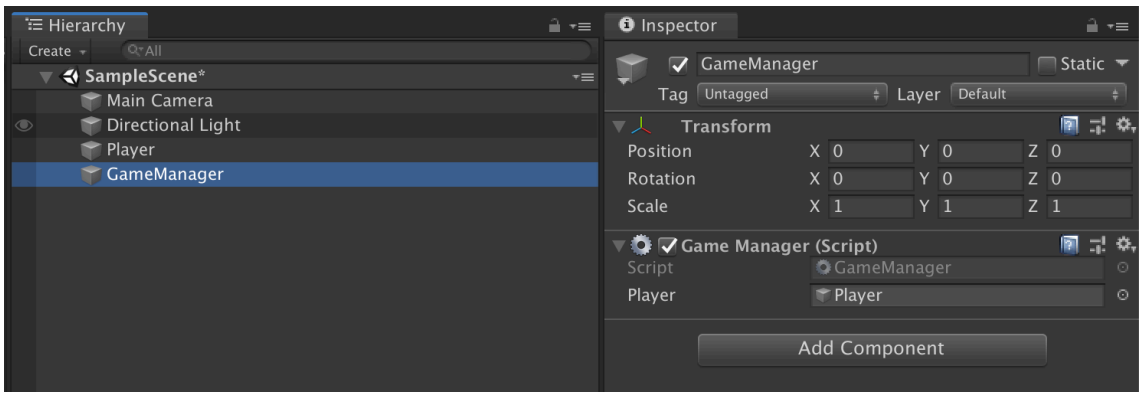

Figure 5. Assigning a value for GameManager from the hierarchy

Variables should only be declared `public` when code from outside the script needs to access it. In order to set the field `private`, it needs to be declared with the `[SerializeField]` -attribute. Serialization allows the developer to save the state of an object and recreate it as needed, providing storage of objects as well as data exchange. (Microsoft Docs 2018). It is often confused that the `[SerializeField]` -attribute in Unity only serves to expose variables to the inspector window, when in fact, it is only a side effect of allowing the user to manually select which states of objects to save to a specific component.

```
1   using UnityEngine;
2
3   public class GameManager : MonoBehaviour
4   {
5       [SerializeField]
6       private GameObject player;
7
8       // Start is called before the first frame update
9       void Start()
10      {
11          // Do something with our player
12      }
13  }
```

Figure 6. Declaring a private variable with the `[SerializeField]` -attribute

The example above results to the same view in Unity's inspector as with Figure 3. This only restricts access to GameManager's variables and assures that the player value can only be modified in the inspector, or in GameManager.

3.3 Find -methods

Besides being quite a slow function, `GameObject.Find` is one example of how awkward the Unity Framework is for big projects development. What happens if someone in the team decides to rename the GameObject? Should GameObjects renaming or deletion been forbidden? (Mandalà 2012a.)

The `Find` -method is highly dependent on the naming of the GameObject that it is trying to fetch. This can very quickly lead to run-time errors that cannot be predicted in development. It is this dependency on the actual naming of the wanted object instead of the object itself that makes the `Find` -method a big red flag in game development.

`Object.FindObjectOfType<T>` is another method Unity provides for acquiring references. This method returns the first active loaded object of Type T, null if none is found. As Unity's documentation (Unity Documentation – Object.FindObjectOfType N.d.) notes, this function is very slow and should not be run every frame. The same goes for `GameObject.Find` (Unity Documentation – GameObject.Find N.d.).

```
1   using UnityEngine;
2
3   public class GameManager : MonoBehaviour
4   {
5       private GameObject player;
6       private Enemy enemy;
7
8       // Start is called before the first frame update
9       void Start()
10      {
11          // GameObject.Find
12          player = GameObject.Find("Player");
13
14          // Object.FindObjectOfType
15          enemy = (Enemy)FindObjectOfType(typeof(Enemy));
16          // or
17          enemy = FindObjectOfType<Enemy>();
18      }
19  }
```

Figure 7. Demonstration of different Find -methods

This method of getting references could be used in a quick prototype or concept when figuring out different mechanics. It is easy to setup, and useful if the job needs to be done quickly. A simple “GameManager” is unlikely to change name in the lifespan of a prototype. Even if it does, it is still feasible to catch the change and update the code. But as soon as talks about building something on top of this prototype arise, one should stray away from using these.

3.4 Singleton pattern

Essentially, a singleton is a class which only allows a single instance of itself to be created, and usually gives simple access to that instance. (Skeet n.d.) An example of implementing such Singleton can be found in Figure 8.

```
1   using UnityEngine;
2
3   public class GameManager : MonoBehaviour
4   {
5       public static GameManager Instance;
6
7       private void Awake()
8       {
9           if (Instance == null)
10              Instance = this;
11          else if (Instance != this)
12              Destroy(gameObject);
13      }
```

Figure 8. A simple Singleton pattern

It is recommended to use Singletons for things that do not need to be copied multiple times during a game. This is great for controller classes like GameManager or AudioController. (Unity Geek 2016.) This type of Singleton can often be found to be recommended on forums and tutorials, as it is easy to set up and works well in small scenarios. It does also have its drawbacks.

Firstly, this type of Singleton is not persistent across Unity scenes. This can be fixed by calling the `DoNotDestroyOnLoad(GameObject go)` -method. Secondly, this code works only for SingletonController, but if another singleton controller eg. AudioController needs to be created, the same code has to be copy

and pasted with some minor changes. This leads to boilerplate code. (Unity Geek 2016.) Boilerplate code or just boilerplate refers to code that has to be implemented to many places throughout classes with little to no alterations.

```
1   using UnityEngine;
2
3   public class GameManager : MonoBehaviour
4   {
5       public static GameManager Instance;
6
7       private void Awake()
8       {
9           if (Instance == null)
10              Instance = this;
11          else if (Instance != this)
12              Destroy(gameObject);
13
14          DontDestroyOnLoad(gameObject);
15      }
```

Figure 9. A Singleton that persists across scenes

These are not the only problems with creating Singletons, and there are plenty of solutions for these issues. Those will not be discussed here, as Strange offers its own solution, which will be explained later on in the study.

3.5 GetComponent<T>

`GameObject.GetComponent<T>` is a way to get references to Components that are attached to GameObjects. This is often used when fetching scripts or components from a certain GameObject. It can also be used to search for components in the GameObject's parents or children. Figure 11 describes a manager GameObject *Manager*, which has a component *ParentGameObject* and a *ChildGameObject* in the hierarchy. In figure 12 the references of *ParentComponent* and *ChildComponent* are fetched to *Manager* through the `GetComponentInParent<T>` and `GetComponentInChildren<T>` -methods.

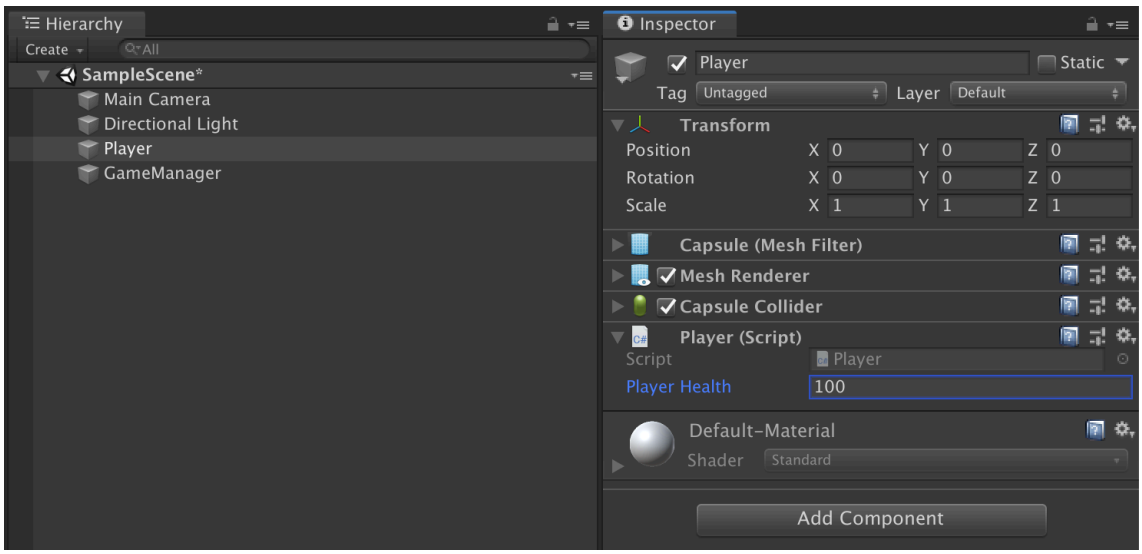

Figure 10. A Player script attached to the Player GameObject

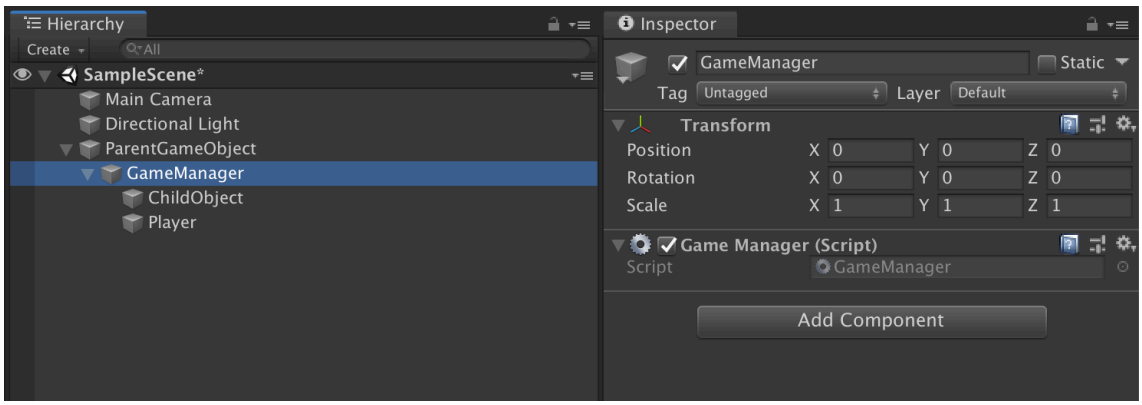

Figure 11. GameManager with parent and child GameObjects

```
1   using UnityEngine;
2
3   public class GameManager : MonoBehaviour
4   {
5       [SerializeField]
6       private GameObject playerGameObject;
7
8       [SerializeField]
9       private Enemy enemy;
10
11      private Player player;
12
13      // Start is called before the first frame update
14      void Start()
15      {
16          // Search for Player -component in playerGameObject
17          player = playerGameObject.GetComponent<Player>();
18
19          // Search for Player in GameManager's parent GameObjects
20          player = GetComponentInParent<Player>();
21
22          //Search for Player in GameManager's chil GameObjects
23          player = GetComponentInChildren<Player>();
24      }
25  }
```

Figure 12. Examples of different GetComponent methods

In this case the reference is not tied to the GameObjects name, but the type of component that is wanted. This is a way better approach. A key difference is that the `Find` -method returns the whole GameObject, whereas `GetComponent<T>` returns a component on that GameObject.

A great example of using this is when one needs to get the root canvas. In figure 13 a reference to the first canvas in parent GameObjects is searched. The canvas returned knows its root canvas.

```
using UnityEngine;

public class GameManager : MonoBehaviour
{
    private Canvas canvas;

    void Start()
    {
        if (canvas == null)
            canvas = GetComponentInParent<Canvas>().rootCanvas;
    }
}
```

Figure 13. Fetching the root canvas from parents

4 INVERSION OF CONTROL FRAMEWORKS

4.1 General

Inversion of Control is a principle in software engineering by which the control of objects or portions of a program is transferred to a container or framework. It is most often used in the context of object-oriented programming. (Crusoveanu 2019.) When building a large application or game, the maintainability of said application or game is a top priority. Companies need to keep bringing updates and bugfixes to keep the product relevant and desirable. One key aspiration to obtain such maintainability is object-oriented programming which aims to promote reusability.

One of the principal advantages of object-oriented programming techniques over procedural programming techniques is that they enable to create modules that do not need to be changed when a new type of object is added. (Beal n.d.). An object is similar to a value in an abstract data type---it encapsulates both data and operations on that data. (Johnson & Foote 1988). This approach to programming promotes dependency decoupling, which in turn leads to modularity and maintainability.

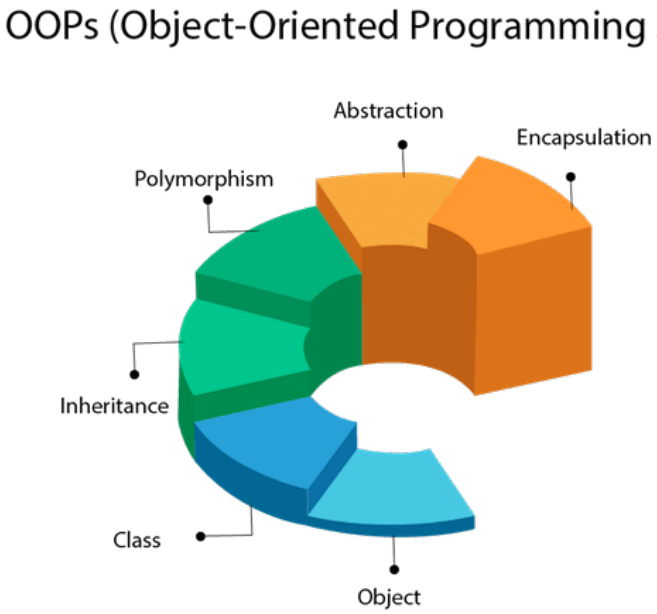

Figure 14. Concept of Object-Oriented Programming (JavatPoint N.d.)

To achieve all these programming ideals, a very useful tool is an Inversion of Control framework, also referred to as an Inversion of Control container. They act as scaffolding to the application, allowing easy navigation through classes and dependencies. Some popular IoC frameworks are Strange, Zenject, Ninject, and Robotlegs.

4.2 Dependecy Injection

“Now it seems that IoC is receiving attention from the design pattern intelligentia: Martin Fowler renames it *Dependency Injection*, and, in my opinion, misses the point: IoC is about enforcing isolation, not about injecting dependencies. The need to inject dependencies is an effect of the need to increase isolation in order to improve reuse, it is not a primary cause of the pattern.” (Mazzocchi 2004).

Inversion of Control (IoC) and Dependency Injection (DI) are two terms that are often heard together. It is easy to mix them or to get the impression that they are the same thing. However, Mazzocchi (2004) outlines that this is not the case; IoC is more of a way of thinking, and DI is an answer to the problems that it brings along. To achieve loose coupling of classes, an IoC framework can pass dependencies to a class by injecting them to it. Instead of the class fetching all the dependencies it needs by itself, it simply raises a flag letting the framework know what it needs. Then, through dependency injection, the framework goes through all these flags and assigns all the dependencies needed.

To achieve modularity, interfaces are used as dependencies. This enables a certain service or function to be changed without affecting any of the other parts of the program, as long as the new substituent fulfils the needed interface. How the assigning of these interfaces to the actual implementation of the logic works will be discussed later in this thesis in chapter 9.1. By using interface abstraction, the receiver of said interface does not need to worry about its implementation. It simply needs to call the available properties and methods that the interface offers and get the data without knowing where it came from. In Figure 15 an example of simple interface abstraction shows how the Client class does not need to know about how the server is implemented, but simply an IServer interface to operate.

The implementation of IServer can be changed freely without it affecting the Client whatsoever.

```
1   public interface IServer
2   {
3       IPlayer GetPlayer();
4   }
5
6   public class DummyServer : IServer
7   {
8       public GetPlayer()
9       {
10          return new Player();
11      }
12  }
13
14  public class Client
15  {
16      private IServer server;
17
18      public Client(IServer server)
19      {
20          this.server = server;
21      }
22
23      public void InstantiatePlayer()
24      {
25          var Player = server.GetPlayer();
26          // Do something with our new Player
27      }
28  }
```

Figure 15. Simple use case example of interface abstraction

5 STRANGE IOC

5.1 General

In this thesis we will take a closer look to Strange IoC, which is specially built for Unity3D. I conducted an interview at Greener Grass to find out why Strange was chosen to be the designated framework to be used in the company. The interview included questions such as “What problems needed to be solved by an IoC framework?”. General consensuses were concluded from the received answers.

After the interview was concluded, I had an encompassing understanding of the reasons why Strange was opted as our IoC container. The theory behind the tools Unity offers to fight dependencies has been covered in previous chapters. The next chapters will explain what aspects of Strange translate well into day to day development, and what can be improved upon. As will be displayed, Strange is far from perfect. It offers a lot of features that are advertised as almost mandatory for a working project structure, but the truth is quite the opposite.

“StrangeIoC is a super-lightweight and highly extensible Inversion-of-Control framework, written specifically for C# and Unity.” (Strange IoC 2015). Some of its key features include Dependency Injection, View mediation, a core binding framework and multiple modular contexts. Strange works on web, standalone, iOS and Android platforms.

To explain the core mechanics of Strange, Third Motion Inc. have made a starting guide called “The Big, Strange How-To”, here on after referred to as the “Strange How-To”. (Third Motion Inc. 2013-2015). The next chapters explain the major features mentioned in the Strange How-To. Reading through the Strange How-To is recommended when getting started with Strange. It is not a perfect or complete guide, but it is a good place to get started. Many further expanding guides can be found online to help with a new Unity project using Strange. The following chapters are a collection of the most crucial parts of Strange and a synopsis of The Big, Strange How-to.

5.2 Why Strange

In addition to Strange IoC we only found one IoC library designed for Unity (Zenject). Generic C# IoC implementations did not fit in the Unity environment because MonoBehaviour components are created by Unity. Unity's support for newer C# and .NET features was also limited. On top of this in Unity's case it is important for the IoC implementation to avoid unnecessary memory allocation and reflection. At the time Strange IoC seemed as the more mature option so we ended up trying it. (Kauko 2019.)

As the project size grows, unclear areas of responsibility and implicit dependencies lead to situations where one functionality disperses into various places and is tightly coupled to other functions. Altering this functionality requires changes to many classes and the maintenance of the application suffers. (Kauko 2019.)

Like other IoC solutions Strange IoC facilitates the organisation of functionalities into modular classes which have restricted dependencies to the rest of the application. A well-organized functionality is easier to understand, upkeep and test with unit-tests. In addition, the dependencies are easy to replace with others so that for example online and offline versions are easy to make from the same game. (Kauko 2019.)

It was a combination of all these reasons listed above and many more that lead to choosing Strange. It is not to say that Strange is not flawless and would not have its drawbacks.

5.3 Biggest drawbacks

According to Kauko (2019), the biggest problem with Strange is error handling and error messages. He explains that especially an exception happening in a constructor remains completely invisible and causes an error message later on which tells nothing about the real issue.

Error handling and error message stack traces were a reoccurring theme when researching the flaws of Strange IoC. Juhala (2019) adds that Strange has many layers of abstraction and finding a specific implementation is difficult. He also notes that Strange uses reflection for searching dependencies, which is slow. Other more task-specific drawbacks are discussed in the following chapters that take a closer look at Strange's functionality.

6 STRANGE INJECTION

6.1 Constructor injection

There are two ways to inject dependencies to classes in Strange. The first way is Constructor Injection that will be discussed in this chapter. The second is Property Injection, that will be discussed in the next chapter (5.2). Both ways of injection are commonly used and have benefits and disadvantages over each other.

One of the biggest advantages of constructor injection is private properties remaining private. It is always advised to keep the variables confined to the extent that they are needed. Another benefit of constructor injection is that as soon as the constructor has been called, all the injections are ready for use.

In the Figure 16, `Client` declares a dependency `IServer server` it needs in its constructor. As soon as the constructor is called, all values are available.

```
1   public class Client
2   {
3       private IServer server;
4
5       public Client(IServer server)
6       {
7           this.server = server;
8       }
9   }
```

Figure 16. Example of constructor injection

The disadvantage of this approach is the amount of code it generates. In order to get a single value to be used in the class, one must declare the class variable, tell Strange in the constructor what value is wanted, then assign this value to the variable. Only after these steps can the rest of the class utilise the injected value. Another disadvantage of constructor injection is that it does not allow named injections.

6.2 Property Injection

Property injections are marked with the `[Inject]` -attribute. This tells Strange to inject a value to this property. This can be seen on the third (3) row in Figure 17. In addition of the inject attribute, it can be made a named injection. A third option is to mark the injectable property with a marker class.

One of the best features of Property Injections is that it requires only one line of code. This greatly reduces the time and code needed to get an injection to a class. When working with property injections, a constructor is not needed for a class. Instead, Strange provides the `[PostConstruct]` -attribute. This allows any method marked by it to be called immediately after injection. In a way this method replaces the constructor used in constructor injection. In the following Figure 17 a property is declared with the `[Inject]` -attribute. This property is then used in the method `PostConstruct()` marked with the `[PostConstruct]` -attribute. It is recommended to name the method this way so that it stays clear which method is called after injection. This is not obligatory but is a convenient naming convention for readability.

```
1   public class SomeMediator : EventMediator
2   {
3       [Inject] public ISomeService SomeService { get; set; }
4
5       [PostConstruct]
6       public void PostConstruct()
7       {
8           SomeService.DoSomething();
9       }
10  }
```

Figure 17. A Property injection being used in the PostConstruct method

One disadvantage listed on the Strange How-To is that the injected dependencies are not available in the constructor, but as demonstrated above, in most cases a `PostConstruct()` -method marked with the `[PostConstruct]` -attribute is just as good as a constructor.

7 STRANGE VIEW MEDIATION

7.1 General

View mediation is the only part of Strange that is written exclusively for use with Unity. This is because in game development views are highly volatile and constraining that natural chaos in the view classes themselves is advisable.

A view can be anything the player can see or interact within the game. For example, a cube is a view, whereas the logic that follows an interaction with said cube is not. Transferring input information from the cube to the logic of the game and back is called mediation. The core structure consists of two MonoBehaviours: a view and its mediator.

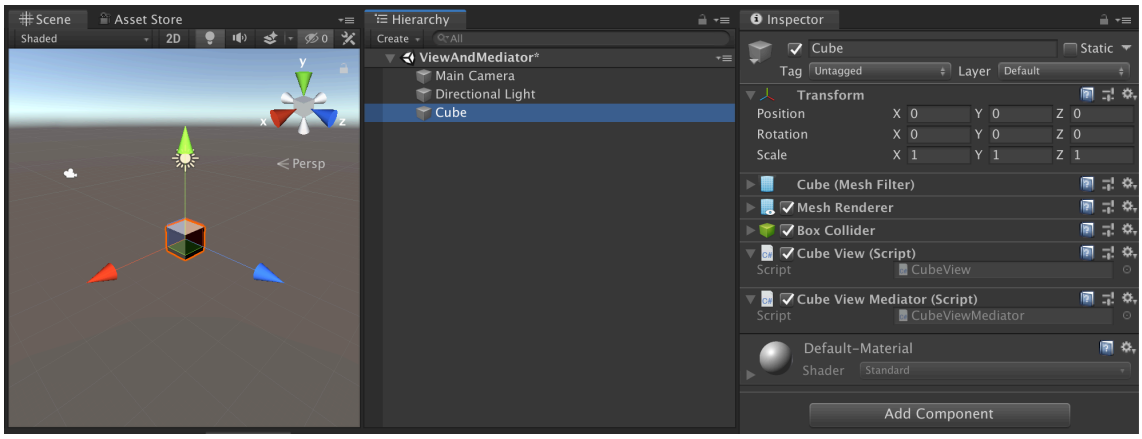

Figure 18. A View with a CubeView and CubeViewMediator attached to it

7.2 View

The View class represents the ‘V’ in the MVCS structure. A View is a MonoBehaviour that you extend to write the behaviour that controls the visual (and audible) input and output that a user sees. (Third Motion Inc. 2013-2015.) In Figure 18 is an example of a View being attached to a GameObject in the editor. It is that simple. Every GameObject present in the scene do not have to be views, but they can be. Simply put, everything that the player needs to send information

through to the logic needs to be a View. Also, anything that needs to be controller by custom logic needs to be a View. A simple cube that is affected by gravity and other physic engines does not need to be a View, if it does not need to send any data anywhere.

One thing worth noting is that in order to be able to have a mediator, a class needs to extend View. This way Strange gets information about its existence. Another solution is to create a custom View, in which for example debugging code can be inserted. This could later be accessible to all Views.

While Views are Injectable, it is often bad practice to tie Views directly to models and services. (Third Motion Inc. 2013-2015). This is suggested due to the natural tendency of view code getting messy. The view should only be responsible for three key tasks: the wiring of the visual components, dispatching events when said components are interacted with and exposing an API which allows the state of the visual component to be changed.

7.3 Mediator

The mediator class is a separate MonoBehaviour whose responsibility is to know about the View and about the app in general. It is a thin class, which means its responsibilities should be very lean. (Third Motion Inc. 2013-2015.) The role of the mediator is to intimately know the view and know enough about the app to send and receive needed events or signals. An injection of the view is added to the mediator for it to know about its view.

The mediator has a method `OnRegister()` that acts as the constructor of the mediator. It fires immediately after injection and can be used to initialize the view and request important data for example. Any mediator that extends EventMediator, so almost every mediator, gets injected with the contextDispatcher granting access to the context-wide event bus. In place of Unity's `OnDestroy()` method, a mediator has `OnRemove()`, which is meant for clean-up and is called just before a view is destroyed. As the mediator is always created for a view, when said view gets destroyed the mediator goes with it. Here is an example of how to bind

a view to a mediator (Figure 19), and how the mediator then uses this view (Figure 20).

```
1    mediationBinder.Bind<SomeView>().To<SomeMediator>();
```

Figure 19. Binding a view to a mediator

```
1    public class SomeMediator : EventMediator
2    {
3        [Inject] SomeView View { get; set; }
4    }
```

Figure 20. A mediator injects a view

7.4 Good practises

A helper class has been implemented at Greener Grass to simplify the injection of the view to the mediator. Instead of injecting the view as in Figure 18, the Mediator inherits `ViewMediator<V>`. `ViewMediator<V>` acts as a middleman between the Mediator and the EventMediator. Implementing and using this method can be seen in Figure 21.

```
1    public class MenuPageMediator : ViewMediator<MenuPage> { }
2
3    public class MenuPage : View { }
4
5    public abstract class ViewMediator<V> : EventMediator
6    {
7        [Inject] public V View { get; set; }
8    }
```

Figure 21. Using the ViewMediator<V> helper class

It has also been found that while the absolute isolation of the view from the rest of the app sounds good on paper, such isolation of the class can lead to unnecessary code in the form of an unneeded Mediator. In Figure 22 the View injects `IProfileService` to itself, and that is all it needs. The use of a mediator here

would have led to a class that only servers as an injector for the view. As the Big, Strange How-To suggests, the mediator is made for transferring information to and from the view. In this case there is no information to be transferred, only a simple injection to be made. It is much easier to inject it straight into the view. When the PostConstruct is run, `ClientView` can simply set the `playerName.text` to the name it gets from `IProfileService`.

```
1   public class ClientView : View
2   {
3       [Inject] IProfileService ProfileService { get; set; }
4
5       [SerializeField]
6       private Text playerName;
7
8       [PostConstruct]
9       public void PostConstruct()
10      {
11          playerName.text = ProfileService.Name;
12      }
13  }
```

Figure 22. A view directly injecting to itself

There is one catch to this. When injecting to a class A, if any of the classes that class A implements have the same injection, only the first class marked with the injection gets injected. This will lead to an “InjectionException: Attempt to instantiate a null binding.” It is easy to quickly add an injection to a class without checking the base classes for the same injection. Thankfully Strange gives a comprehensive error log message that is easy to debug and fix.

8 STRANGE PROMISES

8.1 General

A promise is a powerful tool provided by Strange. A blog post named “Promises, promises” (Strange IoC 2015) was written to explaining everything related to them. This chapter goes through the general usage of promises.

A promise is a pattern for handling asynchronous callback behaviour. (Strange IoC 2015). If a service is needed by the client, but the client does not know when the service is ready for use, it can ask the service via promise. The promise is then returned or “Dispatched” back to the client once the service is ready. This way the client does not have to constantly check when the service is ready but instead is notified upon completion of the service instead.

```
1   public class Service : IService
2   {
3       private IPromise readyPromise;
4
5       public IPromise WaitUntilReady()
6       {
7           if (readyPromise == null)
8               readyPromise = new Promise();
9           return readyPromise;
10      }
11
12      private void OnServiceReady()
13      {
14          readyPromise.Dispatch();
15      }
16  }
17
18  public class Client : View
19  {
20      [Inject] public IService Service { get; set; }
21
22      [PostConstruct]
23      public void PostConstruct()
24      {
25          Service.WaitUntilReady().Then(UseService);
26      }
27
28      private void UseService()
29      {
30          //The service has confirmed it is ready for use.
31      }
32  }
```

Figure 23. A Client using a service after it is ready to use

The IPromise interface contains two main methods: `Dispatch` and `Then`. `Dispatch` is used to fire off the event signalling the completion of the promise. Every class listening to the promise gets notified that it has completed. It should be noted that promises are one time use only and dispatching them a second time will result in a bug where the `Finally` callback is triggered.

`Then` is used to listen to the completion of the promise and triggering a callback once this happens. `Then` is also useful for chaining events as promises get dispatched. This also produces a very comprehensive and easy to read chain of events in code, making their maintenance easy.

```
1   public class Client : View
2   {
3       [Inject] public IService Service { get; set; }
4       [Inject] public ISecondService SecondService { get; set; }
5
6       [PostConstruct]
7       public void PostConstruct()
8       {
9           Service.WaitUntilReady().Then(() =>
10          {
11              SecondService.WaitUntilReady().Then(() =>
12              {
13                  UseServices();
14              });
15          });
16      }
17
18      private void UseServices()
19      {
20          //The services have confirmed they are ready for use.
21      }
22  }
```

Figure 24. Chaining of events using promises

8.2 Usage of promises

A method can return a promise instead of an actual value. This is very useful when waiting for user input for example. A simplified example of the client opening a dialog asking the user for input, then after receiving it doing something with the given input is shown in Figure 25. First the dialog is opened. Our Dialog class contains a method `OpenDialog()` that returns a promise of a result instead of the actual result, as the dialog does not have it at the time of opening. The client then assigns a callback listener `HandleResult()`. Once the dialog gets the input form the user, it fires the `SendResult()` -method, which dispatches the `resultPromise` giving it the integer chosen by the user.

```
1   public class Client
2   {
3       Dialog.OpenDialog().Then(HandleResult);
4
5       void HandleResult(int result)
6       {
7           // Do something with the result
8       }
9   }
10
11  public class Dialog
12  {
13      IPromise resultPromise;
14
15      IPromise OpenDialog()
16      {
17          resultPromise = new Promise<int>();
18          return resultPromise;
19      }
20
21      void SendResult(int result)
22      {
23          resultPromise.Dispatch(result);
24      }
25  }
```

Figure 25. Example of a promise returning a value

A promise does not have to return any value, it can simply be used to signal something happening, for example the dialog being closed. In Figure 26 the client does not need any information from the dialog, only that it was closed. After this the client can proceed with its tasks.

```
public class Client
{
    Dialog.OpenDialog().Then(() =>
    {
        NotifyDialogClosing();
        // continue after dialog is closed.
    });
}

public class Dialog
{
    IPromise closePromise;

    IPromise OpenDialog()
    {
        closePromise = new Promise();
        return closePromise;
    }
    void OnClose()
    {
        closePromise.Dispatch();
    }
}
```

Figure 26. Example of simple promise usage

Chaining promises one after another becomes a powerful tool when multiple asynchronous events have to be completed in order. In Figure 27 the client opens a dialog and after receiving a result passes it to some manager. After the manager has dispatched a confirmation that it has handled the result the client can proceed with its tasks.

```
public class Client
{
    Dialog.OpenDialog().Then(result =>
    {
        NotifySomeManager(result).Then(() =>
        {
            // do something after the manager has dispatched
            // a response to the notification sent to it.
        });
    });
}
```

Figure 27. Example of chaining promises and passing results

9 OTHER STRANGE COMPONENTS

9.1 Core Binding Framework

A Strange binding is made up of two required parts and one optional part. The required parts are a key and a value. The key triggers the value; thus, an event can be the key that triggers a callback. Or the instantiation of one class can be the key that leads to the instantiation of another class. The optional part is a name. Under some circumstances, the name serves as a discriminator. (Third Motion Inc. 2013-2015.)

The most common use case is binding implementations to interfaces. This is the beating heart of Strange's dependency injection. Strange provides one base class for all bindings. This base class can be extended for additional contexts to separate clear modules of the program, and to keep the binding classes clean. Such an extension structure could go as follow: base context, menu context and a match context.

9.2 Modular Contexts

> The context package puts all your various Binders under one roof, so to speak. For example, the MVCSContext includes an EventDispatcher, an InjectionBinder, a MediationBinder, and a CommandBinder. You can, as we have discussed, remap the CommandBinder to a SignalCommand-Binder. The (Signal)CommandBinder listens to the EventDispatcher (or Signals). Commands and Mediators rely on Injection. The Context is where we wire up these dependencies. To create a project, you'll override Context or MVCSContext and it's in this child class that you'll write all the bindings that make your application do what it does. (Third Motion Inc. 2013-2015.)

It is recommended to have multiple contexts. This makes the app very modular, and all of the above-mentioned management gets done in module specific segments. As discussed in chapter 9.1, such modules could be to have a context for the base game, a context for the menu of the app and a context for the match scene. The menu context takes care of all the meta game tasks and the match context takes care of the core gameplay.

Strange uses the MVCSContext as a base for all contexts. The MVC pattern is a software design pattern which stands for Model-View-Controller. The concept of MVC is that the user uses the Controller, which manipulates the Model, which updates the View, which in turn sees the user (Figure 28).

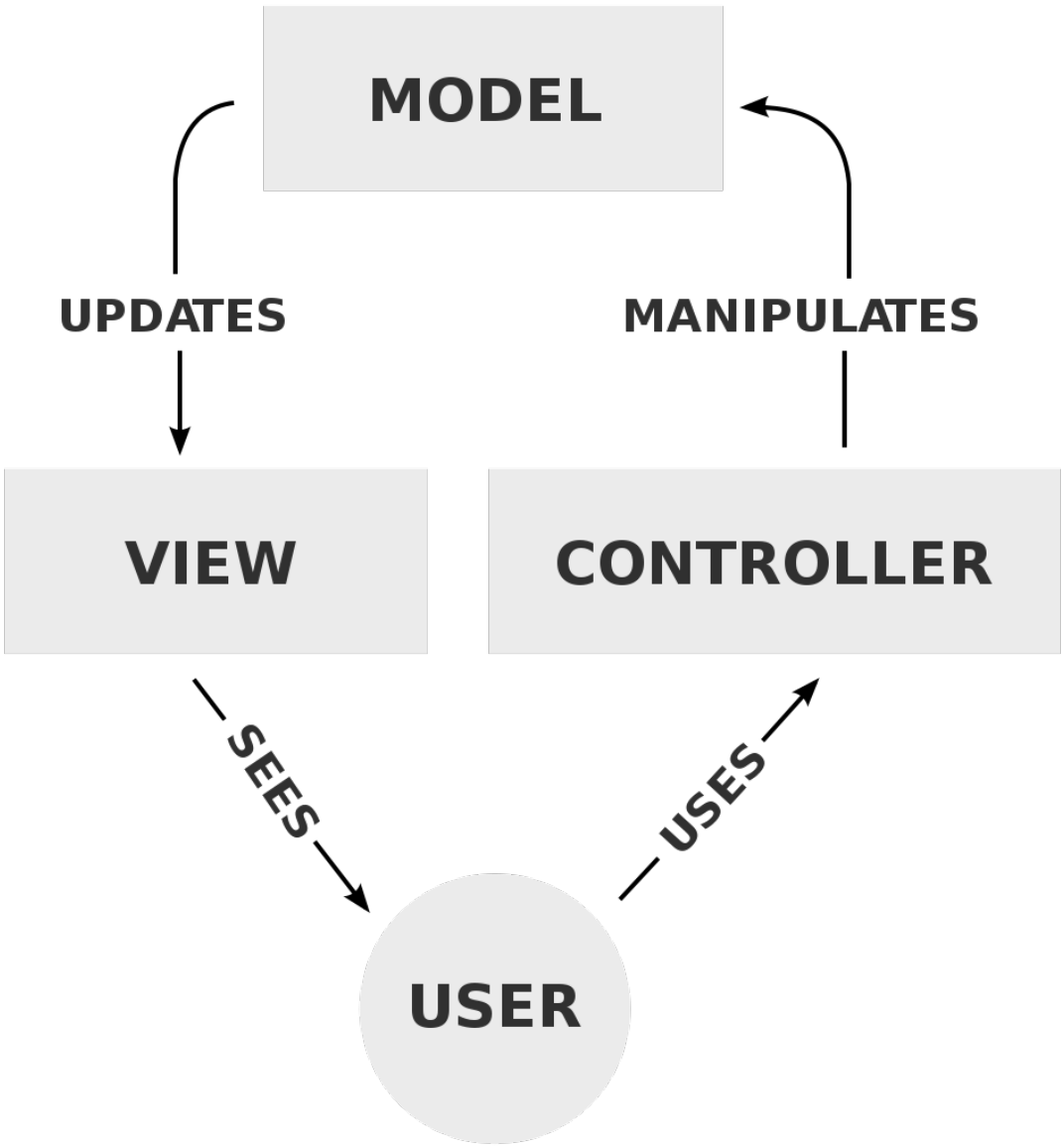

Figure 28. The model, view and controller (MVC) pattern relative to the user (RegisFrey 2010)

Strange adds the “S” to the pattern, and it stands for Service. This refers to anything outside the application, such as a web service. The MVCSContext is

Strange's way of wrapping up the whole micro-architecture into a convenient, easy-to-use package. (Third Motion Inc. 2013-2015.)

9.3 Signals

We do not use Signals for all communication between Mediator and Controller logic. (Kauko 2019). According to Juhala (2019), C#-events are faster and easier to understand, as they are part of a class, and not separated from everything else.

Signals are useful for very general use cases. The fact that they can be injected wherever in the context they are binded in makes them easily accessible throughout code. One example of using signal efficiently is Golden Roll's, a game made by Greener Grass, TopHud. The TopHud includes currency containers, which displays the current amount of currency the player has. This currency can be used and gained all throughout the game. The TopHudMediator injects itself with a `CoinsChanged<int>` signal and adds a listener to it. Every time this signal is dispatched, the TopHud receives information about the change amount and can update the field.

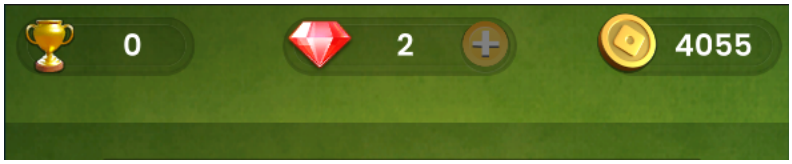

Figure 29. The TopHud when no currency changes are occurring

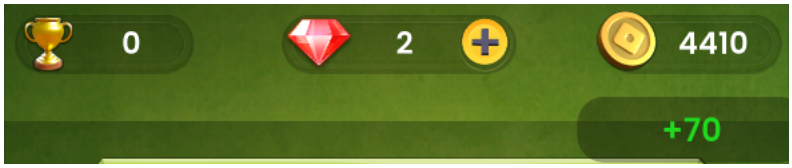

Figure 30. The TopHud reacting to the CoinsChanged<int> Signal

9.4 Commands

Commands are the Controllers in the classic Model-View-Controller-Service structure. In the MVCSContext version of strange, the CommandBinder listens to every dispatch from the dispatcher. Whenever an event fires, the Command-Binder determines whether that event is bound to Commands. If the Command-Binder finds a binding, a new Command instance is instantiated. (Third Motion Inc. 2013-2015.)

Greener Grass established the Commands to be difficult to use and understand. The Command only shows up in bindings and it does not contain information about the event it responds to. (Juhala 2019.) Command chains seemed to complicate simple logic execution and following, so Greener Grass ended up replacing them with singleton controller classes. (Kauko 2019).

10 CONCLUSION

Strange was chosen as our Inversion of Control framework due to it being developed for Unity. At the time of choosing only one other option was available, Zenject. Strange was simply a better fit for the task. The usage of an IoC framework facilitates the organisation of functionalities into modular classes. Even better, it forces the programmer into a mindset where all dependencies and required task for classes are thought out and purposeful.

This thesis has taken a broad look into the benefits Strange brings to the maintainability of a Unity project. By going through all major features that Strange offers, a basic understanding of what benefits it brings and how to use them has been mapped out. On top of this, the thesis has taken a look at how to make even better use of Strange by improving on it and expanding its functionality to better suit game developing needs. Things Strange does not do well, and features that should not be used have been considered.

Personally, I had not heard of Inversion of Control before landing the job at Greener Grass. The learning curve was very steep at first, due to the whole flow of logic being different to what I was used to in smaller school projects. Now that I have been working with Strange for over 6 months, I appreciate the benefits it brings to programming. It makes it easier to write good code by forcing one to think about dedicated tasks and dependencies for classes.

Even though Greener Grass has been updating and modifying Strange to its needs for years, there still remains issues to be solved. One major one is the use of mediators. As discussed in chapter 7, sometimes using a mediator only brings unnecessary complication to an otherwise simple task. This however is not a straightforward problem as a mediator is great when dealing with larger amounts of functionality.

While Strange lays down a solid foundation on the structure and architecture of the code, it still allows for different approaches on how to use its various features.

To maximise the ease of switching from one project to another inside the company, general guidelines on how to use Strange should be decided.

As discussed earlier, getting into Strange's way of thinking about the flow of the logic can be quite the undertaking for a first timer. Possibilities of some sort of introduction to the company's customs of using Strange and starter exercises have already been discussed but need to be put in place. This would greatly shorten the time it takes for a new employee to get up to speed with the code and would help soften the landing into Inversion of Control from a more widely known way of thinking about control flow.

Using some sort of Inversion of Control framework is recommended when dealing with larger project that contain a lot of logic outside the views and projects that involve many platforms and variations. If a project is purely a visual demo that does not need state saving, logic outside of views or external services the usage of IoC is not recommended (Kauko J. 2019.)

BIBLIOGRAPHY

Beal, V. N.d. OOP – Object Oriented Programming. Accessed on August 28, 2019. Retrieved from https://www.webopedia.com/TERM/O/object_oriented_programming_OOP.html

Bonfiglio, N. 2018. DeepMind partners with gaming company for AI research. Accessed on September 25, 2019. Retrieved from https://web.archive.org/web/20181002122834/https://www.dailydot.com/debug/unity-deempind-ai/

Clement J. 2019. Number of active apps from the Apple App Store 2008-2019. Accessed on October 21, 2019. Retrieved from https://www.statista.com/statistics/268251/number-of-apps-in-the-itunes-app-store-since-2008/

Crusoveanu, L. 2019. Intro to Inversion of Control. Accessed on August 28, 2019. Retrieved from https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

Durand W. 2013. From STUPID to SOLID Code! Accessed on September 25, 2019. Retrieved from https://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/#tight-coupling

Gough C. 2019a. Google Play: number of available gaming apps as of Q3 2019. Accessed on October 21, 2019. Retrieved from https://www.statista.com/statistics/780229/number-of-available-gaming-apps-in-the-google-play-store-quarter/

Gough C. 2019b. Number of games released on Steam 2004-2018. Accessed on October 21, 2019. Retrieved from https://www.statista.com/statistics/552623/number-games-released-steam/

Hight, J. 2013. Diablo 3 auction house update. Accessed on September 21, 2019. Retrieved from https://us.diablo3.com/en/blog/10974978/diablo%c2%ae-iii-auction-house-update-9-17-2013

How is game development different from other software development? 2011. An answer from Stack Overflow by corsiKa. Accessed on September 19, 2019. Retrieved from https://gamedev.stackexchange.com/questions/9074/how-is-game-development-different-from-other-software-development/9089#9089

JavatPoint N.d. Java OOPs Concept. Accessed on October 19, 2019. Retrieved from https://www.javatpoint.com/java-oops-concepts

Juhala, M. Game Programmer. Interviewed 18.09.2019. Interviewer Ahonen, J. Greener Grass Oy, Tampere.

Johnson, R. & Foote B. 1988. Designing Reusable Classes. Accessed on August 26, 2019. Retrieved from
http://www.laputan.org/drc/drc.html

Kauko, J. Technical Director. Interviewed 24.09.2019. Interviewer Ahonen, J. Greener Grass Oy, Tampere.

Mandalà, S. 2012a. Inversion of Control with Unity – part 1. Accessed on August 30, 2019. Retrieved from
http://www.sebaslab.com/ioc-container-unity-part-1/

Mazzocchi S. 2004. On Inversion of Control. Accessed on September 2, 2019. Retrieved from
https://web.archive.org/web/20040202120126/http://www.betaversion.org/~stefano/linotype/news/38/

Microsoft Docs. 2018. Serialization (C#). Accessed on August 29, 2019. Retrieved from
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/serialization/

Mobvista 2018. Why mobile gaming is now bigger than console and PC gaming combined -. Accessed on November 12, 2019. Retrieved from
https://www.mobvista.com/en/blog/mobile-gaming-now-bigger-console-pc-gaming-combined-still-growing-always-changing/

Oberlehner M. 2019. Dependency injection in Vue.js applications. Accessed on November 12, 2019. Retrieved from
https://markus.oberlehner.net/blog/dependency-injection-in-vue-applications/

Petrillo, F., Pimenta, M., Trindade, F. & Dietrich, C. 2009. What went wrong? A survey of problems in game development. Accessed on September 19, 2019. Retrieved from
https://www.researchgate.net/publication/220686446_What_went_wrong_A_survey_of_problems_in_game_development

RegisFrey. 2010. Diagram of interactions within the MVC pattern. Accessed on October 19, 2019. Retrieved from
https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller#/media/File:MVC-Process.svg

Skeet, J. N.d. C# in depth. Accessed on August 29, 2019. Retrieved from
https://csharpindepth.com/articles/singleton

Strange IoC. 2015. Promises, promises. Accessed on September 19, 2019. Retrieved from
https://strangeioc.wordpress.com/2015/04/28/promises-promises/

Third Motion Inc. 2013-2015. The big, Strange How-To. Accessed on August 26, 2019. Retrieved from
https://strangeioc.github.io/strangeioc/TheBigStrangeHowTo.html#h.sjblqrdytark

Unity Geek. 2016. Singleton: Implementation in Unity3D C#. Accessed on August 28, 2019. Retrieved from http://www.unitygeek.com/unity_c_singleton/

Unity Documentation - GameObject.Find. N.d. Accessed on October 18, 2019. Retrieved from https://docs.unity3d.com/ScriptReference/GameObject.Find.html

Unity Documentation - Object.FindObjectOfType. N.d. Accessed on October 18, 2019. Retrieved from https://docs.unity3d.com/ScriptReference/Object.FindObjectOfType.html