jamk.fi

Dependency Injection in Unity3D

Niko Parviainen

Bachelor's thesis
March 2017
Technology, communication and transport
Degree Programme in Software Engineering

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s)	Type of publication	Date
Parviainen, Niko	Bachelor's thesis	March 2017
		Language of publication: English
	Number of pages: 57	Permission for web publication: x

Title of publication
Dependency Injection in Unity3D

Degree programme
Degree Programme in Software Engineering

Supervisor(s)
Rantala, Ari
Hämäläinen, Raija

Assigned by
Psyon Games Oy

Abstract

The objective was to find out how software design patterns and principles are applied to game development to achieve modular design. The tasks of the research were to identify the dependency management problem of a modular design, find out what the solutions offered by Unity3D are, find out what the dependency injection pattern is and how it is used in Unity3D environment.

Dependency management in Unity3D and the dependency injection pattern were studied. Problems created by Unity3D's solutions were introduced with examples. Dependency injection pattern was introduced with examples and demonstrated by implementing an example game using one of the available third-party frameworks. The aim of the example game was to clarify if the use of dependency injection brings modularity in Unity3D environment and what the cost of using it is.

The principles of SOLID were introduced with generic examples and used to assist dependency injection to further increase the modularity by bringing the focus on class design.

Dependency injection with the help of SOLID principles increased the modularity by loosely coupling classes even though slightly increasing the overall complexity of the architecture. Increased loose coupling and separation of concerns brought by SOLID principles were important aspects for modularity which dependency injection cannot bring on its own.

Keywords/tags (subjects)
game development, software design patterns, software design principles, software architecture

Miscellaneous

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Parviainen, Niko	Opinnäytetyö, AMK	Maaliskuu 2017
	Sivumäärä 57	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Riippuvuusinjektio Unity3D-ympäristössä

Tutkinto-ohjelma
Insinööri (AMK), ohjelmistotekniikan tutkinto-ohjelma

Työn ohjaaja(t)
Ari Rantala
Raija Hämäläinen

Toimeksiantaja(t)
Psyon Games Oy

Tiivistelmä

Työn tavoitteena oli selvittää, miten ohjelmistokehitysmalleja ja -periaatteita voidaan soveltaa pelikehitykseen tavoiteltaessa modulaarista arkkitehtuuria. Tehtäviin kuului tunnistaa luokkariippuvuuksien hallinnan ongelma modulaarisessa arkkitehtuurissa, selvittää Unity3D-ympäristön tarjoamat ratkaisut, selvittää mikä on riippuvuusinjektiomalli ja kuinka sitä käytetään Unity3D-ympäristössä.

Luokkariippuvuuksien hallintaa Unity3D:ssä ja riippuvuusinjektiomallia tutkittiin. Unity3D:n esittelemien ratkaisujen ongelmat tuotiin esille esimerkein. Riippuvuusinjektiomalli esiteltiin esimerkein ja demonstroitiin kehittämällä esimerkkiprojektina peli käyttäen yhtä saatavilla olevaa kolmannen osapuolen sovelluskehystä riippuvuusinjektiolle. Esimerkkipelin tavoitteena oli ottaa selvää, tuoko riippuvuusinjektion käyttö Unity3D-ympäristössä modulaarisuutta pelikehitykseen ja millä hinnalla.

SOLID-ohjelmistokehitysperiaatteet esiteltiin esimerkein ja tuotiin riippuvuusinjektion tueksi korostamaan modulaarisuutta siirtämällä huomiota myös luokkasuunnitteluun.

Riippuvuusinjektion käyttö yhdessä SOLID-periaatteiden kanssa tuo modulaarisuutta sitomalla luokat löyhästi toisiinsa, vaikkakin samalla nostaa arkkitehtuurin monimutkaisuutta kokonaisuudessaan. SOLID-periaatteiden tuoma laaja löyhien sidosten määrä ja vastuiden erottelu olivat tärkeitä osia modulaarisuutta, jota riippuvuusinjektio ei itsessään voi tuoda.

Avainsanat (asiasanat)
pelikehitys, ohjelmistokehitysmallit, ohjelmistokehitysperiaatteet, ohjelmistoarkkitehtuuri

Muut tiedot

Contents

Figures

Terminology

API

Application Programming Interface is a set of methods usable by a client.

Entity

In Unity3D, an instance of GameObject-class representing any object in the game.

Dependency

Class A requiring another class or an interface to perform its function makes class A dependent on the class or interface.

DI

Dependency Injection, a software design pattern where dependencies of a class are supplied (injected) from the outside.

IoC

Inversion of Control, a software design pattern where flow of control is inverted when compared to traditional procedural programming.

Pure DI

Use of dependency injection without third-party dependency injection tools. (Seemann 2014.)

Scene

File in Unity3D where all entities are stored in a hierarchy.

Unit Test

Piece of code where output / behavior of a single method is validated.

Lazy initialization

Programming technique where the object is instantiated when requested instead of instantiating beforehand.

1 Introduction

Psyon Games is a Finnish Jyväskylä-based start-up games company. Their main cause is to develop games based on real science to teach and inspire players about different science topics while having fun. Getting people interested in science through games is a great idea; however, game development is not easy and making a game fun is not a particularly easy task. The constant flow of changes to find the most fun combination of game mechanics can be a very difficult and long process. From a programmer's perspective, these constant changes present a difficulty: the written code should be as effortless as possible to adapt to the constantly changing requirements.

This bachelor's thesis aims to identify and solve the technical aspects of the problem in Unity3D environment in particular. Unity3D is a popular free and commercial game engine which has been used for several popular games such as Cities: Skylines, Firewatch and Kerbal Space Program (Games Made with Unity n.d).

Chapter 2 chapter introduces the difficulties that arise in game development from a programmer's perspective and identifies the technical difficulties. As the thesis focuses on Unity3D, the basics of programming in Unity3D are introduced next. After the basics, an introduction of the tools Unity3D provides or the techniques that are often used in Unity3D as solutions for technical problems are presented.

After describing these common ways and the problems they cause, dependency injection pattern is demonstrated as a solution for the problem. Additionally, the SOLID principles are introduced to get more out of dependency injection by bringing focus to class design. Next, the dependency injection in Unity3D is demonstrated with the usage of third-party tools called dependency injection frameworks.

The methods are demonstrated by building a simple Unity3D game prototype using one of the dependency injection frameworks available for Unity3D. The aim of this project was to create a loosely coupled structure for the game which would be easy to change and extend.

2 Difficulties in Game Development

In software development, the goal is to fulfill a business need. Therefore it is possible to create a fairly straight-forward specification and schedule to complete every required task the business needs require. If the application performs the tasks and outputs valid data according to the specification, it fulfills the needs. (How is game development different from other software development 2011.)

In game development, the business need is *fun*. There can be a specification and schedule to implement everything that is needed to the game; however, this does not guarantee the game is fun to play. Writing a technical specification for *fun* is not an easy task. (How is game development different from other software development, 2011.)

The lack of specification for *fun* is one of the difficulties in game development as the game may be changed drastically during the development in order to find the correct gameplay mechanics for a fun experience.

There are cases of changing the game drastically even after it has been published, e.g. Diablo 3 by Blizzard Entertainment. Diablo 3 featured an auction house where the players could sell in-game items they had found for real money or for the currency used in-game. When Diablo 3 launched, it received much criticism and one of the targets was the auction house. Blizzard Entertainment responded to this by completely removing the auction house from the game almost two years after the game was released. (Hight 2013.)

The auction house is not the only feature that changed in Diablo 3 after its release. The game is currently different from the version of the release day. (Kaiser 2016.)

Information about the technical details of Diablo 3 are not available; however, from the developer's perspective situations like this may still raise questions about how to deal with such cases as smoothly as possible. Cases like removing an auction house have business related concerns but also engineering concerns. How to design the required systems so that they can be extended, replaced or completely removed as the case was with Diablo 3? Modularity is a key aspect of such design.

In object-oriented code, classes need to know about other classes and there are many ways to create the relations between them, and there are many ways to design the classes itself. If the relationship between classes cannot be removed without modifying the classes a great deal, it is not a modular design, which could possibly be the result of tightly coupled classes, classes with low cohesion and the lack of Inversion of Control (IoC).

Tightly coupled classes depend on each other directly. If making a change in one class requires changes to another class, there is coupling. It is clear that tight coupling does not favor modularity. (Durand 2013.)

Low cohesion in a class means that the class does several things that are not related to each other very well. It is difficult to see what the responsibility of the class is as parts of it are arbitrarily grouped together. (Skrchevski 2015.) When a class does too many things, it may lead many other classes to depend on it, which makes all the classes dependent on this single class, thus decreasing modularity.

Inversion of Control is a generic software design pattern where the flow of control is inverted by using events, triggers, callbacks or similar concepts. Subscribing to an event to perform additional actions instead of modifying the method that could raise the event increases modularity as the method raiser does not need to know about the subscribers. (Fowler 2005.)

Robert C. Martin (n.d) said whenever a nasty batch of tangled legacy code is brought on the screens, the results of poor dependency management are experienced. Poor dependency management leads to code that is hard to change, fragile, and non-reusable.

It can be stated that the problem is the dependency between classes and the management of dependencies. Class A needs the functionality provided by class B to perform an action, however, the requirement is to allow that without sacrificing modularity. Instead of tight coupling and low cohesion, the aim is exactly the opposite: high cohesion and loose coupling.

The symptoms of a bad dependency management are rigidity and fragility. In rigidity, making a change in one method requires changes in another class and possibly the

changes in this other class require changes to another etc. The change affects the dependencies deeply. In fragility, changes to a class cause something that seems to have no relationship with the changed class to break. (Martin 2014.)

Before introducing the tools provided by Unity3D to solve the problem, the next chapter introduces the basics of programming in Unity3D.

3 Programming in Unity3D

Unity3D is based on entity framework where new behaviors are added to entities by attaching components to them. In Unity, entities are called GameObjects and components are called MonoBehaviours. (Mandalà 2012a.)

GameObjects can be created programmatically or manually with the editor. GameObjects can also be saved as prefabs which can be used to programmatically instantiate multiple GameObjects with a specific set of MonoBehaviours in them.

Instantiated GameObjects are stored in and available from the Scene. GameObjects can also be nested, thus one GameObject can act as a parent for multiple GameObjects. This forms the hierarchy of current entities in the Scene. (Unity - Manual: The Hierarchy Window n.d.)

Figure 1 shows a MonoBehaviour component for a “Hello World” program.

```
using UnityEngine;

public class HelloWorld : MonoBehaviour
{
    private void Start ()
    {
        Debug.Log("Hello World!");
    }
}
```

Figure 1. Hello World example for Unity

The GameObject with the “Hello World” component in the hierarchy and in the Inspector is illustrated in Figure 2. The Inspector tool also shows a Transform-component being attached which is a mandatory component for every entity.

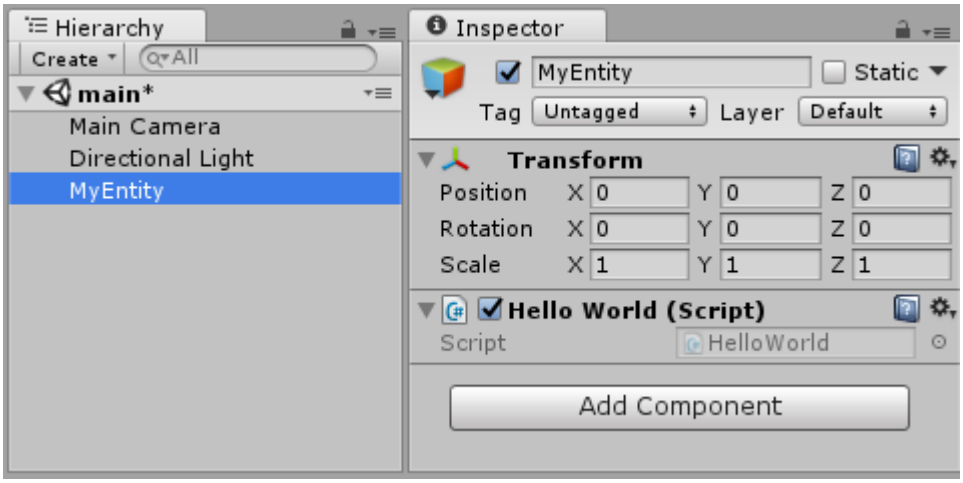

Figure 2. Scene Hierarchy and Inspector in Unity

The Inspector tool is used to set the data in the components of the entities. In Figure 3, the public fields of a MonoBehaviour can be filled by using the Inspector tool, which makes it easier for designers without programming experience to work with the design choices of a programmer. Unity also allows the creation of customized Inspector tools. These custom Inspectors can be used to add buttons or display additional fields for more complicated components.

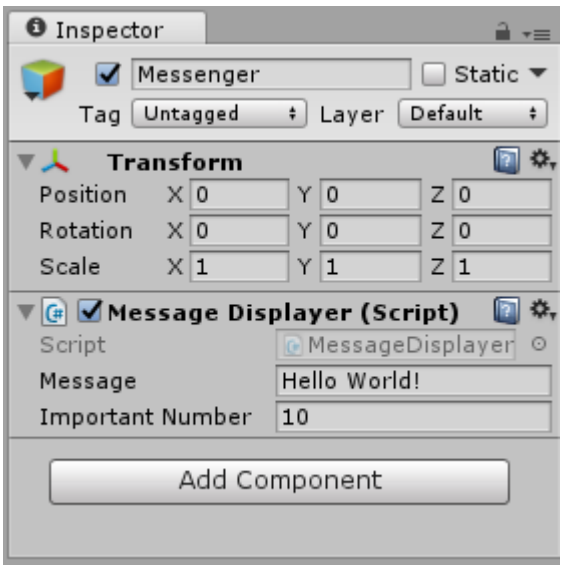

Figure 3. MonoBehaviour with fields to fill in the Inspector

Figure 4 shows the code for the MessageDisplayer MonoBehaviour.

```
using UnityEngine;

public class MessageDisplayer : MonoBehaviour
{
    public string Message;
    public int ImportantNumber;

    private void Start ()
    {
        Debug.Log(Message);
        Debug.Log(ImportantNumber);
    }
}
```

Figure 4. Implementation of a MonoBehaviour with fields

The Inspector is also able to display regular classes when they are fields of a Mono-Behaviour and they have the Serializable-attribute (Figure 5).

```
public class AnotherThing : MonoBehaviour
{
    public RegularClass InstanceOfRegularClass;
    public RegularClass[] ArrayOfRegularClasses;
}

[System.Serializable]
public class RegularClass
{
    public string Message;
}
```

Figure 5. MonoBehaviour with serializable classes as fields

This feature can be used to insert the values of the fields of the object right from the editor as seen in Figure 6.

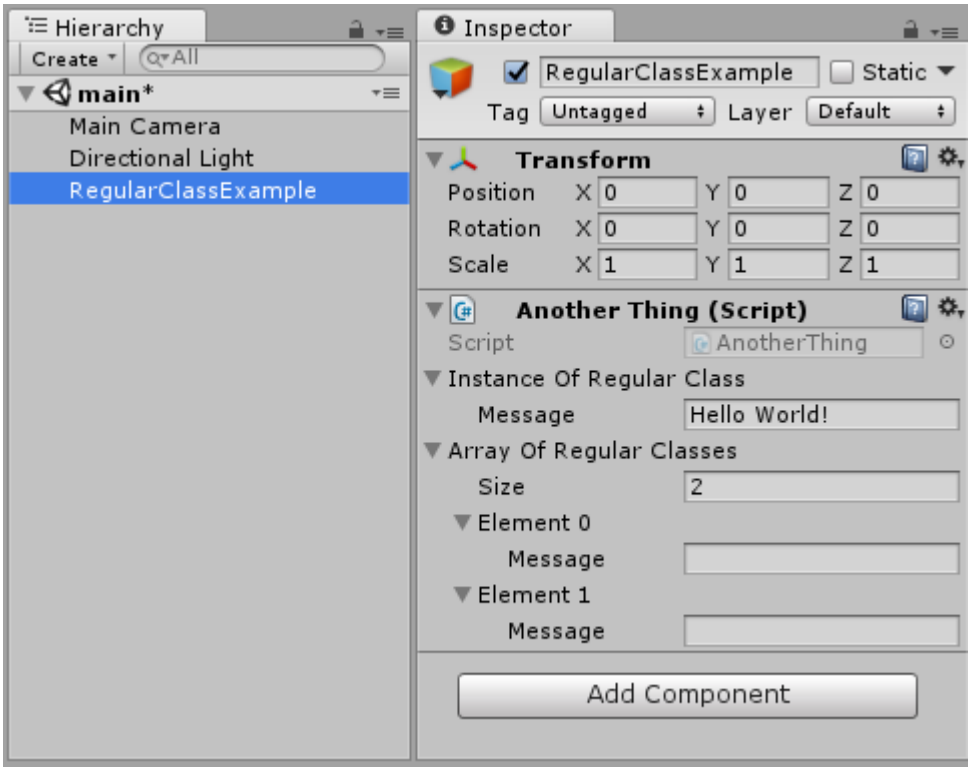

Figure 6. Field of a class-type displayed in the Inspector

MonoBehaviours are instantiated by Unity itself, however, MonoBehaviours can hook into two different phases of the process by creating methods called Awake or Start. The Start-method is used in the previous examples (Figure 1, Figure 4). There are also methods such as Update, FixedUpdate and LateUpdate to perform actions periodically. Awake is called first, even if the component were to be disabled. Start is called next but before the next Update. (Unity - Awake and Start n.d)

The next chapter covers how to manage dependencies in Unity3D using the tools provided by Unity or using the instructed ways.

4 Managing Dependencies in Unity3D

Unity3D does not provide a very large set of tools for handling dependencies. Unity3D provides find methods like *GameObject.Find* and *Object.FindObjectOfType* which can be used to satisfy dependencies. Another easy way to get certain objects available for other objects is to use the Singleton pattern. (Mandala 2012a.)

Unity3D does neither provide a single entry-point to the application, which makes the managing of dependencies even more difficult as there is not that much control over what is instantiated and when (Mandalà 2012b). It is possible to hook into different stages of the start-up process by creating Awake and Start methods in MonoBehaviour derived classes which are then attached to GameObjects in the Scene. However, Unity has full control over the actual instantiation of these MonoBehaviours in GameObjects.

There is also the editor tool called Inspector which is used to visually drag and drop object references for other objects. It is a good tool for designers and not really meant for handling all the dependencies. The reference setting is limited only to objects that are visible in the editor, which are Unity3D's MonoBehaviour and GameObject classes.

A real downside is that the Inspector is not capable of displaying interfaces; meaning, even if one's MonoBehaviour based class were to implement an interface and another MonoBehaviour based class had a dependency on that interface as seen in Figure 7.

```
// Depends on the interface
public class InterfaceExample : MonoBehaviour
{
    public Thing DirectReference;
    public IThing Abstraction;
}

public interface IThing { }

// Implements the interface
public class Thing : MonoBehaviour, IThing
{
    public string Example;
}
```

Figure 7. MonoBehaviour with interface field

The dependency will not show up in the Inspector as seen in the example (Figure 8).

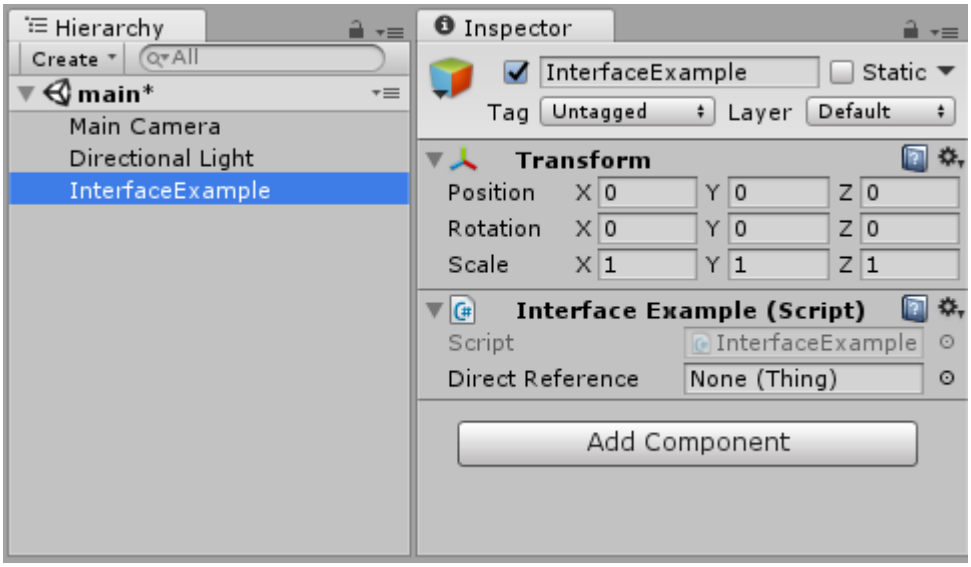

Figure 8. Interface is not displayed in the Inspector.

Following sections describe the Singleton pattern and the find methods commonly used for handling dependencies in Unity3D game development.

4.1 Find -methods

GameObject.Find method looks for the desired object from the Scene hierarchy. The method accepts a string as a parameter and tries to find a GameObject with that string as its name. Another find method is the Object.FindObjectOfType method which similarly tries to find an instance of the desired class from the Scene hierarchy.

The problems in the GameObject.Find method are easy to see. It looks for the name of the GameObject in the Scene hierarchy based on a string, which is not very ideal as there can be many objects with the same name. In such a case, it returns the first one it finds. Relying on string names may cause errors that can only be detected on run-time and not during compiling if the object cannot be found. If the object is found, this object then must be checked with GetComponent method to get the possibly desired component. Additionally, the method is slow, which leads to poor performance. (Mandalà 2012a.)

Similarly, the Object.FindObjectOfType method returns the first object it finds in the Scene hierarchy. Unity documentation warns that this method is also slow (Unity - Manual: Object.FindObjectOfType n.d.). The method also only accepts types derived

from the Object-class, which means instances of classes implementing a certain interface cannot be looked for.

```
using UnityEngine;

public class DependencyFindTest : MonoBehaviour
{
    private GameObject dependency1;
    private Thing dependency2;

    private void Awake()
    {
        dependency1 = GameObject.Find("Something");
        dependency2 = FindObjectOfType<Thing>();
    }
}
```

Figure 9. Demonstrations of Find-methods

Both of these methods are clumsy for managing dependencies as the returned instance may vary during runtime by outside factors. This can potentially make their usage quite dangerous. Unity documentation guides to use the singleton pattern for most cases (Unity - Manual: Object.FindObjectOfType n.d.).

4.2 Singleton Pattern

Singleton pattern can be implemented in many ways; however, the main idea is the same in every implementation. In the example (Figure 10), the singleton pattern is accomplished with the generic singleton class. This implementation lazily initializes an object from the desired class and stores it privately in a static field. Then the same instance can be accessed by other classes via the public static property. This is the main point of the singleton pattern: to give a global access to one and only instance of a class, the singleton. (CSharp in Depth 2011.)

```
using System.Collections.Generic;

public abstract class Singleton<T> where T : class, new()
{
    private static T instance;
    public static T Instance
    {
        get
        {
            if (instance == null) instance = new T();
            return instance;
        }
    }
}

public class MySingleton : Singleton<MySingleton>
{
    public List<string> Things;
}

public class SomeManager
{
    public void DoSomething()
    {
        foreach(string thing in MySingleton.Instance.Things)
        {
            /* .. */
        }
    }
}
```

Figure 10. Implementation of the singleton pattern

The Singleton pattern is very simple and can be learned and used with very little effort, which might make it a tempting design choice. However, even if it presents very simple design, it brings more complicated problems that may be hard to see at first. These problems are all related to modularity. One problem is that the access to the object is global. It does not have any restrictions. It makes it possible to access the singleton from anywhere, which makes it very difficult to maintain any architecture as the developers can do anything they want. When every developer does everything differently in a tight project schedule, a clear architecture cannot be established. (Mandala 2012a.)

Singleton also makes it difficult or nearly impossible to create proper unit tests, considering e.g. the situation in the example (Figure 11). The Client class uses the Server singleton, and the server makes a connection to an actual server. If the server is of-

fline, the unit test fails which does not tell if the method does what it should. Another problem would be if the Server singleton preserves a state which could possibly affect other unit tests (Densmore 2004).

```
public class Server : Singleton<Server>
{
    public Data GetData()
    {
        /* Fetches data from a server */
    }
}

public class Client
{
    public int DoAction()
    {
        var data = Server.Instance.GetData();
        /* Do something with the data */
    }
}

public class ClientTest
{
    [Test]
    public void TestDoAction()
    {
        var client = new Client();
        var result = client.DoAction();
    }
}
```

Figure 11. Singleton used in a unit test

Additionally, having unit tests pass or fail depending on the factors outside the scope of the unit test is not efficient. It would be much more convenient to be able to replace the Server class with a fake class. This fake class could be set to always provide valid or invalid data. Then the unit tests could indicate in a much clearer way that the Client class handles the data correctly. This approach is discussed in more detail in Chapter 5.

Another view to the same problem of Client class always using the Server-class is if the Server class is replaced with another implementation, it also requires changes to the Client class or any other class using the Server class. It may not be a big task to do in the early development; however, later it would be a nightmare for the programmer.

It can be concluded that the tight coupling between Client and Server classes is causing problems and clearly reduces the modularity.

5 Dependency Injection

5.1 General

Dependency Injection (DI) is a more common pattern in software development but it can also be used in Unity3D for game development. It is a promising pattern for dependency managing to achieve modular code. DI is often called a fancy term for simple concept.

DI takes a simple but different approach to the problem. Instead of class A instantiating or fetching the dependencies, these dependencies are “injected” from the outside for the class A. Injecting means passing an instance of a class for another class, usually as a parameter in the constructor. (Shore 2006.)

In the example (Figure 12), the Singleton example (Figure 11) is replaced by DI for better flexibility and loose coupling. In the example, Client class depends on the IServer interface instead of directly depending on a specific class. The usage of interface abstraction makes the Client class unaware of the implementation details of the interface.

```
public interface IServer
{
    Data GetData();
}

public class FakeServer : IServer
{
    public Data GetData()
    {
        return new Data();
    }
}

public class Client
{
    private IServer server;

    public Client(IServer server)
    {
        this.server = server;
    }

    public int DoAction()
    {
        var data = server.GetData();
        /* Do something with the data */
    }
}
```

Figure 12. Example of dependency injection

This loose coupling gives the developer freedom to create the FakeServer class for testing purposes as seen in Figure 13. Then the developer can create the “normal” Server-class which implements the same IServer interface for the actual application. When it is possible to swap implementations easily, the modularity is already greatly improved.

```
public class ClientTest
{
    [Test]
    public void TestDoAction()
    {
        var server = new FakeServer();
        var client = new Client(server);
        var result = client.DoAction();
    }
}
```

Figure 13. Injecting a dependency

However, this relies heavily on the fact that the interfaces are well designed. Changes in the interface may cause difficulties. This subject is presented in more detail in Chapter 7.

In technical terms, what DI does is called Inversion of Control (IoC) which is a more generic pattern of inverting the control of something as compared to traditional procedural programming. IoC helps keeping classes with high cohesion and loosely coupled. (Inversion of Control 2016.)

For example, the usage of events achieves IoC. In the example below (Figure 14), the upper class diagram is converted to use events to achieve more modularity. It can be seen from the diagram that the control between Orc class and DeathAnimation and AchievementManager classes is inverted. The event raiser does not need to interact with or even know about the objects interested in its event. In other words, the event raiser is unaware of the observers. Similarly, the inversion of control is present in dependency injection: a class depending on an interface or another class does not determine which instance it is going to use. Once again in other words, the class depending on the interface is unaware of the actual implementation.

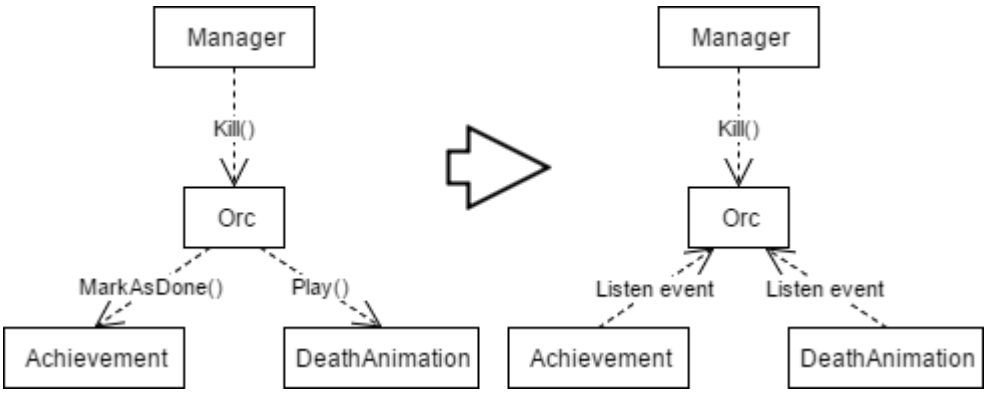

Figure 14. Inversion of control using events.

Building a game or application by using dependency injection entirely raises a question where all the objects should be created. According to Mark Seemann (2011), the Object Graph should be composed as close as possible to the application's entry-

point. For example, for a console application it would be the Main method. This special place is called the Composition Root. When objects are injected for other objects, they form a graph of dependencies called the Object Graph.

However, in Unity3D, there is not a single entry-point as stated before, which may create complications if considering using dependency injection without any third-party tools. Third-party tools for DI called Dependency Injection Frameworks can be used with Unity3D to help with the entry-point problem. More information about the frameworks is found in Chapter 6.

5.2 Advantages of Dependency Injection

When using dependency injection with interface abstraction, the code becomes loosely coupled, which allows the developer to replace implementations of components to another implementation as seen in Figure 12. While being able to swap components is an advantage itself, it can be seen as different advantage depending on the perspective.

5.2.1 Extensibility

Loose coupling helps with extending the application or game. In game development where the requirements may change a great deal during the lifetime of the project, this can be very useful. An old component not meeting the new requirements can be substituted with a new one.

Depending on the situation, the old component could also be extended by using the decorator pattern. In decorator pattern, a new implementation of the same interface is created, however, it will have a dependency on the old one. In other words, the new one works as a wrapper for the old one. The new implementation then uses the old one and adds the new behavior. (Shvets n.d.)

5.2.2 Testability and Mocking

Unit testing becomes much easier as the developer can change the implementations. For example, to test if a component of the application correctly processes certain data. Instead of fetching the data from a real database, the developer can create a

fake class (a mock) which creates the data in code without accessing a database. The test becomes more reliable as the functionality of the database does not affect the test. (Baharestani 2013, 27)

5.2.3 Late Binding

If using a dependency injection container, the container is expected to return an instance of the given type. The decision of the type can be delayed to the runtime, which gives the ability to create a configuration to determine which type to use without needing to recompile the application. (Baharestani 2013, 28)

5.3 Problems with Dependency Injection

When using dependency injection without any third-party frameworks or libraries, as Pure DI, there are couple of problems. One of the problems is Unity specific problem and another one a more generic problem. A third-party dependency injection tool can be used as a solution for both problems. Below these problems are described in more detail.

5.3.1 Complex Composition Root

When not using a DI framework, the Object Graph is created at the Composition Root manually. If the project is big and has many dependencies, the Composition Root may become very big and even difficult to maintain.

The next example (Figure 15) has an EnemyManager class which has the responsibility to keep track of all enemies in the level. There is no need for more than one EnemyManager class, therefore, the only instance is injected for anything that needs the manager. In this example, the number of other classes that depend on the EnemyManager class is high and injecting them one by one creates a messy looking and less maintainable entry point.

```
namespace ConsoleApplicationExample
{
    class Program
    {
        static void Main(string[] args)
        {
            var player = new Player();
            var enemyManager = new EnemyManager();

            var enemy = new Enemy(new RegularEnemyAI(enemyManager, player));
            var smartEnemy = new Enemy(new SmartEnemyAI(enemyManager, player));
            var something = new Something(enemyManager);
            var anotherThing = new AnotherThing(enemyManager);
        }
    }
}
```

Figure 15. Messy looking Composition Root

In a larger project, the number of classes depending on the EnemyManager class could be much higher. Injecting them manually one by one would create a Composition Root that is very difficult to read.

5.3.2 MonoBehaviour components in Unity3D

In Unity3D when the game is launched, there is not much control over what happens as there is not a single-entry point to the application (Mandalà 2012b). Instead every GameObject's MonoBehaviour component can implement special methods that Unity itself calls to hook custom operations by the developer into certain points of the initialization. The Composition Root is an important part when using DI for the entire application or game.

Another problem related to the MonoBehaviour class is the fact that they are attached to GameObjects and Unity3D itself instantiates them, the control of injecting dependencies in the Composition Root is lost. (Mandalà 2012b.)

To solve these problems, there are third-party tools that can be used. These tools are called Dependency Injection Frameworks. They provide various of methods to create complex dependency injections with less code. They also provide more than the traditional constructor injection. The frameworks targeted for Unity3D also help with the entry-point problem.

6 Dependency Injection Frameworks

Dependency injection frameworks are also known as IoC Containers or DI Containers.

6.1 General

DI containers are a piece of software that handle the resolution of dependencies in objects. They hold the information about dependencies that can be injected into another object by demand. (Adic Documentation 2016.)

The dependencies are determined by configuring the container by setting various bindings. A binding is the definition of the relationship between the requested dependency and the injected dependency. For example, a class may depend on an interface, thus the binding defines which instance is injected to satisfy that dependency.

A previously introduced problem of a complex Composition Root (Figure 15) can be simplified with a DI container a great deal. The following example (Figure 16) demonstrates bindings with Zenject. Now, the same instance of EnemyManager and Player classes are injected with a single line of code for any number of classes. This means that the classes depending on EnemyManager do not need to be referred to in the bindings. Any class that has dependency to the EnemyManager will have the same instance injected by the framework.

```
using Zenject;
using UnityEngine;

public class GameInstaller : MonoInstaller<GameInstaller>
{
    public override void InstallBindings()
    {
        Container.Bind<Player>().To<Player>().AsSingle();
        Container.Bind<EnemyManager>().To<EnemyManager>().AsSingle();
    }
}
```

Figure 16. Dependency bindings set using Zenject

The following chapters introduce a few of DI containers targeted for Unity3D. They are not for Unity3D only but many of them are built on the research done by Sebastian Mandalà. The research was to provide proof of concept of using dependency injection in Unity3D. All the frameworks are very similar to each other and all of them support the injection of MonoBehaviour scripts. They all do the same thing, however, have slightly different features and syntax.

6.2 Zenject

Zenject is a dependency injection container targeted especially for Unity3D. It is built on the work of Sebastian Mandalà and inspired by another DI container called Ninject. (Zenject Documentation 2016.)

Some mobile games for Android and iOS are using Zenject. The most famous of them is Pokemon Go by Niantic Labs. (Zenject Documentation 2016.)

Zenject comes with additional features to make the development for Unity3D easier. These features are meant to reduce the amount of MonoBehaviour based classes by bringing some of the MonoBehaviour features for normal C# classes.

Zenject provides ITickable, ILateTickable and IFixedTickable to bring the MonoBehaviour class features of Update, LateUpdate and FixedUpdate to normal classes. Implementing these interfaces in a regular class and setting the binding as shown in the example is enough. (Zenject Documentation 2016.)

Another feature is the IInitializable interface, which can be used to create initialization logic for objects. Initialization should not be executed in the constructor as it would occur in the middle of Object Graph creation which could cause problems. Instead, the interface can be used in the same way as the other interfaces as seen in Figure 17. The initialization is then executed after the whole Object Graph is created. (Zenject Documentation 2016.)

```
using Zenject;
using UnityEngine;

public class GameInstaller : MonoInstaller<GameInstaller>
{
    public override void InstallBindings()
    {
        Container.Bind<IInitializable>().To<RegularClass>().AsSingle();
        Container.Bind<ITickable>().To<RegularClass>().AsSingle();

        // Or shorter:
        Container.BindAllInterfaces<RegularClass>().
        To<RegularClass>().AsSingle();
    }
}

public class RegularClass : IInitializable, ITickable
{
    public void Initialize()
    {
        Debug.Log("Hello World!");
    }

    public void Tick()
    {
        Debug.Log("I'm called every frame!");
    }
}
```

Figure 17. Demonstration of Zenject's Unity specific features

The last additional feature is the support of C#'s IDisposable interfaces which can be used to do clean up after a scene is changed or the application is closed or any other reason that causes the object to be destroyed. (Zenject Documentation 2016.)

Figure 18 shows a "Hello World"-program in Unity3D with Zenject. SceneContext is created in the editor and the HelloWorldInstaller is attached to any GameObject and reference to the component is set to the SceneContext.

```
using Zenject;
using UnityEngine;

public class HelloWorldInstaller : MonoInstaller<HelloWorldInstaller>
{
    public override void InstallBindings()
    {
        Container.Bind<string>().FromInstance("Hello World!");
        Container.Bind<HelloWorldProgram>().AsSingle().NonLazy();
    }
}

public class HelloWorldProgram
{
    [Inject]
    public HelloWorldProgram(string message)
    {
        Debug.Log("I say: " + message);
    }
}
```

Figure 18. “Hello World” example using Zenject

6.3 Adic

Adic is a dependency injection container targeted for Unity3D and any other C# project. It is based on the DI container by Sebastiano Mandalà and studies of Strange-IOC. The main goal of Adic is to be simple to use and extend.

Adic comes with additional features like Zenject. When Zenject comes with ITickable interfaces, Adic comes with identical features with IUpdatable, ILateUpdatable and IFixedUpdatable. In addition to those same features as in Zenject, Adic comes with IPausable to detect when application is paused, IFocusable to detect when application is focus is changed and IQuitable when the application exists. As in Zenject, implementing the interface and binding it in a container is enough to use them. (Adic Documentation 2016.)

Figure 19 shows a “Hello World” program for Adic. The context root inherited from ContextRoot is attached to a GameObject. The Init method would contain initialization code to start the game, however, for an example this simple nothing is required.

```
using Adic;
using UnityEngine;

public class GameRoot : ContextRoot
{
    public override void SetupContainers()
    {
        var container = this.AddContainer<InjectionContainer>();

        container.Bind<string>().To("Hello World!");
        container.Bind<HelloWorldProgram>().ToSelf();

        var helloWorldProgram = container.Resolve<HelloWorldProgram>();
        helloWorldProgram.Display();
    }

    public override void Init()
    {
        // To initialize the game
    }
}

public class HelloWorldProgram
{
    private string message;
    public HelloWorldProgram(string message)
    {
        this.message = message;
    }

    public void Display()
    {
        Debug.Log("I say: " + message);
    }
}
```

Figure 19. "Hello World" example using Adic

6.4 Forms of Dependency Injection

Dependency injection containers present additional ways to inject dependencies. Probably the most common way, already introduced in Figure 12 is the constructor injection. Each of the forms of injection is present in each of the introduced frameworks.

Constructor injection should be the first choice for injection as it is the most portable one and it guarantees the non-existence of circular dependencies.

6.4.1 Constructor Injection

When an instance is created, dependencies are injected in the constructor. In the example (Figure 20), dependencies B and C are passed as parameters to A which has dependencies on them.

```
public class A
{
    private IFoo b;
    private IBar c;

    public A(IFoo b, IBar c)
    {
        this.b = b;
        this.c = c;
    }
}
```

Figure 20. Constructor injection

The constructor injection is the simplest form of injection and does not require any kind of framework to work. However, in Unity3D, this cannot be used as MonoBehavior classes should not have constructors as it may create unexpected behavior. Instead the method injection is preferred for MonoBehaviours (Adic Documentation 2016).

6.4.2 Field/Property Injection

In the field or property injection, the dependencies are injected directly into the fields. In the example, B and C fields are marked with an “Inject” attribute of the framework such as Zenject (Figure 21). The attribute tells the IoC container that a dependency must be injected for the field or property.

```
public class A
{
    [Inject]
    private IFoo b;
    [Inject]
    private IBar c;
}
```

Figure 21. Field injection

This approach is sometimes used when the constructor injection starts looking messy, i.e. has many dependencies injected. However, the messy constructor is usually a sign of bad code, which could mean that the class does far too many things, which then again hurts modularity. (Kainulainen 2013.)

More about improving class structure to improve modularity and to use dependencies injection more efficiently is found in chapter 7.

6.4.3 Method Injection

Method injection is similar to constructor injection, except it is also used with IoC containers. Similarly to field injection, the Inject-attribute is used to tell the framework of required injections.

```
public class A
{
    private IFoo b;
    private IBar c;

    [Inject]
    public void SomeMethod(IFoo b, IBar c)
    {
        this.b = b;
        this.c = c;
    }
}
```

Figure 22. Method injection

Method injection is often recommended to use for MonoBehaviours in Unity3D as it is closest to the constructor injection. (Zenject Documentation, 2016).

There are still ways to get more out of dependency injection by considering the way classes and interfaces are structured. For example, if one class does too many things, it will result in many other classes to depend on it, even if they are using only a small

part of its functionality, and if almost every class depends on a certain class, it reduces modularity. This is something dependency injection cannot solve on its own. The next chapter introduces a collection of software design principles to improve the class and interface structure to further improve modularity and get more out of dependency injection.

7 The Principles of SOLID

7.1 General

SOLID is an acronym introduced by Michael Feathers for a collection of design principles named by Robert C. Martin. The acronym stands for Single Responsibility Principle (SRP), Open / Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP) and Dependency Inversion Principle (DIP). (SOLID 2017.)

The purpose of these principles is to make it more likely for the programmer to write systems that are easier to maintain and extend over time (SOLID 2017). Some of them may seem identical with each other; however, they still cover different aspects. Just like DI, these principles are more common in software development but nothing stops using them in game development as well.

Dependency injection with interface abstraction is one way to apply the dependency inversion principle. Following the other principles allows to get more out of dependency injection as they point the developer towards writing more modular and maintainable code. These principles are not laws but more like guidelines (Durand 2013). In some scenarios, it may not be the best solution to follow them, however, one can be sure that it is not a bad solution either.

Following chapters explain and demonstrate each principle in game related cases. For simplicity, the examples are not directly Unity3D related.

7.2 Single Responsibility Principle

The single responsibility principle states that every class should have only one reason to change: the class has only one responsibility. Following this principle helps to

avoid creating a god-class, a class with far too many responsibilities. (Durand 2013.) Following this principle also helps to write unit tests easier as the classes are significantly smaller when they have only one concern.

Responsibilities can be thought as people with different roles. Somebody may have one role and somebody else two or more roles. These people want changes to the code depending on their role. (Martin 2012.)

For example, some artist wants to change that numerical health presentation of enemies to a health bar. The game designer wants to make changes to the way damage is calculated. Yet another role wants to change the format of data persistence.

Making one of those changes should not require changes in another area. When SRP is applied, it helps to reduce fragility. Fragility means when one makes changes in one class or method, some feature that seems to have no relation with the change breaks. (Martin 2012.)

Following example (Figure 23) shows a simple class that has too many responsibilities. The Character can attack, take damage and greet. It also validates the health that is set. The class is simple but it has too many responsibilities already, it can be changed for more than one reason. For example, the attacking logic or health validation may change. Both are completely unrelated; the attack logic has nothing to do with validation of health value.

```
public class Character
{
    private string name;
    private float health;
    private float maxHealth;
    private float damage;

    public Character(string name, float damage, float health, float maxHealth)
    {
        this.name = name;
        this.damage = damage;
        this.maxHealth = maxHealth;
        this.health = ValidateHealth(health);
    }

    public void Attack(Character target)
    {
        target.InflictDamage(damage);
    }

    public void InflictDamage(float amount)
    {
        health -= amount;
    }

    public void Greet()
    {
        Debug.Log("Hi, I am " + name);
    }

    private float ValidateHealth(float newHealth)
    {
        return (newHealth <= maxHealth) ? newHealth : maxHealth;
    }
}
```

Figure 23. Class with several responsibilities

The following example (Figure 24) shows the same Character class but stripped from other responsibilities.

```
public class Character
{
    public HealthComponent Health;
    public CombatComponent Combat;
    private string name;

    public Character(string name, HealthComponent health, CombatComponent com-
bat)
    {
        this.name = name;
        this.Health = health;
        this.Combat = combat;
    }

    public void Greet()
    {
        Debug.Log("Hi, I am " + name);
    }
}
```

Figure 24. Responsibilities removed from the base class

Other responsibilities are divided into multiple classes (Figure 25). For example, to change the health validation, one only needs to touch the HealthComponent class and to change the attack logic, one touches the CombatComponent class. However, there is still room for improvements in the example as it is not following all the other principles.

```
public class HealthComponent
{
    private float health;
    private float maxHealth;

    public HealthComponent(float health, float maxHealth)
    {
        this.maxHealth = maxHealth;
        this.health = ValidateHealth(health);
    }

    public void InflictDamage(float amount)
    {
        health -= amount;
    }

    private float ValidateHealth(float newHealth)
    {
        return (newHealth <= maxHealth) ? newHealth : maxHealth;
    }
}

public class CombatComponent
{
    private float damage;

    public CombatComponent(float damage)
    {
        this.damage = damage;
    }

    public void Attack(Character target)
    {
        target.Health.InflictDamage(damage);
    }
}
```

Figure 25. Responsibilities in separates classes

7.3 Open / Closed Principle

Open Closed Principle states that classes should be open for extension but closed for modification (Durand 2013.), which means one should be able add new features without modifying already existing code.

Following example (Figure 26) shows a class where another overloaded method needs to be added to handle a new kind of character; it requires modifying of already existing code which violates the principle. For example, adding a NeutralCharacter would require changes.

```
public class CharacterManager
{
    private float damage;

    public void DamageCharacter(FriendlyCharacter target)
    {
        target.InflictDamage(damage);
    }
    public void DamageCharacter(EnemyCharacter target)
    {
        target.InflictDamage(damage);
    }
}
```

Figure 26. Badly designed class easily violating OCP

In the next example (Figure 27), the problem is solved by a simple interface. When the method takes an interface as parameter, it does not need to care about what class it is handling as long as the class implements the interface. Because of this, new types can be created and used without modifying the existing code. Another way would be to use an abstract base class.

```
public interface IKillable
{
    void InflictDamage(float amount);
}

public class FriendlyCharacter : IKillable
{
    private float health;

    public void InflictDamage(float amount)
    {
        health -= amount;
    }
}

public class CharacterManager
{
    private float damage;

    public void DamageCharacter(IKillable target)
    {
        target.InflictDamage(damage);
    }
}
```

Figure 27. More friendly design for new types

7.4 Liskov Substitution Principle

Liskov Substitution Principle states that if type S is a subtype of type T then type T can be substituted by type S (Durand 2013). This means that when a method has a parameter of type T and a type S is given as the parameter, the result should be expected to be the same without exceptions. Following this principle can help to avoid creating code that is misleading in that sense.

The reader is asked to consider the following example (Figure 28). IEnemy defines the behaviors of enemies: attacking and moving. It may seem fine, however, unexpected behavior is introduced by the EvilTree because even if it is an enemy, it is not supposed to move. As IEnemy requires the EvilTree class to implement the Move-method, this would result in an empty method, however, such a method is very confusing for someone else to read, and the result would not be expected either.

```
public interface IEnemy
{
    void Attack(IEnemy target);
    void Move(Vector2 position);
}

public class Wolf : IEnemy
{
    private Vector2 position;

    public void Attack(IEnemy target) { /* Attack logic */ }

    public void Move(Vector2 position)
    {
        this.position = position;
    }
}

public class EvilTree : IEnemy
{
    public void Attack(IEnemy target) { /* Attack logic */ }

    public void Move(Vector2 position)
    {
        /* EvilTree should not be able to move */
    }
}
```

Figure 28. Unexpected behavior introduced by empty method

The unexpected behavior is the cause for violating the principle. A workaround would require the type to be checked for EvilTree in special cases; however, this would violate the previously introduced Open / Closed Principle where one should not need to modify existing classes to introduce new features. (Durand 2013.)

From unit testing point of view, the unexpected behavior can be understood better as a problem. One could imagine a case where it was to be tested if every IEnemy type of unit moves correctly. Looping through a list of IEnemy objects and calling the Move-method for each will cause the test to fail if there is an enemy like the EvilTree which would not move to the desired position.

This problem often occurs when trying to model the real world in code. The next example (Figure 29) tries to define birds as classes and interfaces. IBird interface says that birds sing and fly, which sounds acceptable at first, however, penguins are also birds; however, they are not capable of flying.

```
public interface IBird
{
    void Sing();
    void Fly();
}

public class Hawk : IBird
{
    public void Sing() { /* Sing logic */ }
    public void Fly() { /* Fly logic */ }
}

public class Penguin : IBird
{
    public void Sing() { /* Sing logic */ }
    public void Fly()
    {
        /* Penguins are not able to fly */
    }
}
```

Figure 29. A try to model the real world

Another similar problem with the same idea is the rectangle and square problem, where square is a subtype of rectangle. Setting the width of square also sets the

height, which may cause unexpected behavior when passing square as a substitute for a method which accepts rectangles. (Durand 2013.)

It can be said that objects in the real world may have a clear relationship but in object-oriented design the relationship should depend on the object behavior instead (Ancheta 2015).

The solution for the problem is to not define methods in interfaces that cannot be implemented in all the classes. Also, avoiding trying to model the real world may help as seen in the example (Figure 29). This principle is one of those where there probably are going to be multiple solutions depending on the situation. For the problem introduced in Figure 28, the next principle (Interface Segregation Principle) may offer a suitable solution.

7.5 Interface Segregation Principle

Interface segregation principle states that a client should not be forced to depend upon interfaces that they do not use (Durand 2013). This is similar to Single Responsibility principle by being about roles.

In the previous example (Figure 28), an assumption was made that every enemy can move, however, the EvilTree type should not. Nevertheless, it still has the move method which it does not need. Therefore, it is violating ISP in addition to LSP. ISP can be applied as a solution by splitting the IEnemy interface into smaller interfaces.

In the next example (Figure 30), the IEnemy interface has been split into smaller interfaces that define smaller behaviors. Now, the Wolf and EvilTree both can attack but only the Wolf type implements the IMovable interface. Now, the EvilTree will not have methods it does not need.

```
public interface IMovable
{
    void Move(Vector2 position);
}
public interface IAttacking
{
    void Attack(IKillable target);
}
public interface IKillable
{
    void InflictDamage(float amount);
}

public class Wolf : IKillable, IMovable, IAttacking
{
    public void Attack(IKillable target) { /* Attack logic */ }

    public void InflictDamage(float amount) { /* Damaging logic */}

    public void Move(Vector2 position) { /* Move logic */ }
}

public class EvilTree : IKillable, IAttacking
{
    public void Attack(IKillable target) { /* Attack logic */ }

    public void InflictDamage(float amount) { /* Damaging logic */}
}
```

Figure 30. Multiple interfaces defining smaller capabilities

In a real case scenario, the interfaces would most likely introduce more than one method. For the sake of simplicity, only one is introduced in the example.

7.6 Dependency Inversion Principle

Dependency Injection is one way to easily follow this principle if interface abstraction is also applied. DIP states two things. First, high-level modules should not depend on low-level modules. Both should depend on abstractions. Second, abstractions should not depend on details. Details should depend on abstractions.

(Durand 2013.)

In other words, instead of class A directly depending on class B, an abstraction is used between them. Dependency injection example (Figure 12) already follows this principle by using the interface IServer. Now, the Client class (high-level module) depends on the abstraction. And the FakeServer class (low-level module) implements the interface. Dependency inversion of Figure 12 is visualized in the Figure 31.

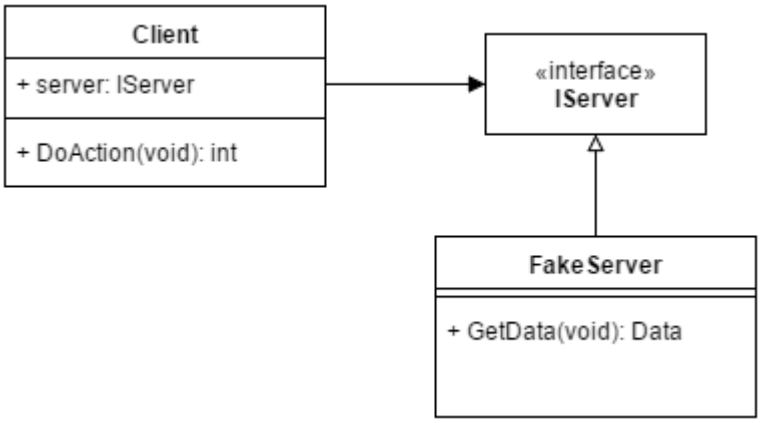

Figure 31. Dependency Inversion demonstrated as a class diagram.

DIP may look very similar to DI; however, they are very different things. DIP does not care about when or where the dependency is instantiated. DI on the other hand does not care about the abstraction. However, DI with DIP is a great deal more useful because the abstraction brings loose coupling.

8 Test Project

The test project is to create a simple game prototype using Zenject as the dependency injection container.

8.1 Design and Requirements

The game is called Mirror Puzzle. the player controls photon emitters and tries to shoot photons to correct photon receivers using mirrors which reflect the photons. The player has a limited amount of tries and must manage to land photon in each receiver to complete the game. Photon receivers only accept photons of specific wavelength. The player can alter the wavelength of the emitted photons by changing the amount of energy supplied to the photon emitters. Higher energy input results in shorter wavelength in photons and vice versa. The wavelength is visualized for the player as a color.

On the technical side, the aim is to create entities and systems as loosely coupled as possible. It should be possible to implement new features without any or with minimal changes in existing code. The implementation should also take advantage of dependency injection and aim to follow the principles of SOLID.

The technical design should not limit the options for art or game designers. Unity3D's Inspector tool should be usable to configure visuals and implementation details.

8.2 Tools and Technology

The design choices take the advantage of C# 6.0 features which are not supported by Unity's .NET 2.0/3.5-like API compatibility. However, there is a special editor-only beta version of Unity (5.5.0b9) that has .NET 4.6 compatibility. This version is required. As a side note, the beta release uses an updated garbage collector which runs in a special debug mode, which decreases the overall performance. (Chambers 2016).

Adic was the first choice for dependency injection container. It has method descriptions available in IntelliSense and is very well documented, however, it requires a separate Inject method call in MonoBehaviour Start methods which eventually lead to change the choice of the dependency injection container to Zenject. Zenject is also very well documented; however, lacks the method and method parameter descriptions for IntelliSense.

Microsoft Visual Studio 2015 Enterprise with Tools for Unity is used as the main development environment for programming. Git with GitHub is used as the version control.

8.3 Implementation

In the following chapters, the most interesting points and features are explained in detail. First the application flow is explained without going too deeply into details and then specific areas are explained in more detail.

The project with all its source code is available at GitHub for free.

8.3.1 Application Flow

The game starts with Zenject reading the bindings set in various Installer scripts. These bindings determine how the dependencies are injected. First, bindings are set in GameInstaller. After all bindings, various initializations are run before the game is playable.

As designers need to insert photon receivers, photon emitters, mirrors and walls around the Scene using the Unity editor, the first action is to fetch these Entities for the EntityManager, so other systems can get their hands on these Entities. Additionally, EntityManager needs to get hold of Spawners that spawn Entities which should be available via the EntityManager. These actions are done by the SceneEntityTracker and EntityManagerInitializer classes that are instantiated as singletons by Zenject. The next steps are followed with more initialization process for round managing, resource managing and exchange managing.

Round managing contains initialization for the dependency injection layer for storing the instances meant for win and lose conditions in an instance of ConditionContainer. This class and its instance are meant for the bindings as a helper. Because there needs to be only one instance of win and lose conditions that systems can observe, they are stored in this helper object for easier access. The ConditionContainer object can then be used in Installer scripts to get the specific instance using Zenject's FromResolveGetter method on it. The initialization is demonstrated in Figure 32 with identical case with the Resource managing.

```
// Here we bind ResourceContainer as singleton
Container.Bind<ResourceContainer>().To<ResourceContainer>().AsSingle();

// And here we Resource instances to the container.
// Other bindings can use 'FromResolveGetter' on 'ResourceContainer' to get a
instance for 'Charges' or 'Energy'
Container.Bind<IResource<int>>()
.WithId(ResourceContainer.Resource.Charges)
.To<Resource>().WithArguments(ChargeStock)
.WhenInjectedInto<ResourceContainer>();

Container.Bind<IResource<int>>()
.WithId(ResourceContainer.Resource.Energy)
.To<Resource>().WithArguments(EnergyStock)
.WhenInjectedInto<ResourceContainer>();
```

Figure 32. Bindings for the ResourceContainer

Additional initialization for round managing is in EmitterTrigger and RoundResetter which listen to OnRoundStart and OnRoundEnd events in the IRoundManager implementation. Round ending is determined with separate end condition which is observed by the IRoundManager implementation. Win and lose conditions also listen to the OnRoundEnd event.

Resource managing has the exactly same helper “pattern” used for storing instances of energy and charge Resources. The helper class is ResourceContainer for these instances. Figure 33 demonstrates the usage of the ResourceContainer helper for setting the Resource instance for displaying energy in the UI.

```
Container.Bind<IResource<int>>()
.FromResolveGetter<ResourceContainer>(x => x.Energy)
.WhenInjectedInto<ResourceText>();
```

Figure 33. Usage of the ResourceContainer

Exchange management has initialization for exchanging energy resource to wavelength in photon emitters. ExchangeInitializer creates the ResourceWavelengthExchange objects for every photon emitter. These exchange objects can then be used by other systems to exchange energy Resource to specific photon emitter’s

wavelength. One of these systems is the ExchangeItemLister, which creates UI elements for each exchange object to allow the player to affect the wavelengths of the photon emitters.

After all initializations, the game is playable. However, nothing happens without player input. The player can manipulate the wavelength of photon emitters, rotate the photon emitters or emit photons from every emitter to try to solve the puzzle.

By moving the sliders on the left, the UI script commands an exchange object do an exchange by sending a value based on the value of the slider. In the prototype, the only exchange objects are instances of the ResourceWavelengthExchange.

The player can also rotate the photon emitters. Clicking an emitter enables the mouse-based rotating and moving the mouse commands an emitter to rotate to look at the mouse position.

To solve the puzzle, the player can click the button on the bottom of the screen. This button commands the IRoundManager implementation to start a round. As mentioned before, at initialization, EmitterTrigger starts listening for the OnRoundStart event and causes all emitters to emit single photons. Then again, the IRoundManager implementation listens for the end condition to trigger which causes the round to end and possibly win and lose conditions to be met. If win or lose conditions are not met, nothing happens, except the cleaning done by RoundResetter. Win and lose conditions are listened by the UI elements.

8.3.2 Entities

Entities can differ a great deal, and it is possible that the definition of any entity could change during the development. Entities can also share same behaviors. For this reason, the capabilities of an entity are composed instead of inherited by using a concept called mixin. In practice, units of functionality are created in separate classes and then instances of these classes are used by a parent class (Mixin 2017.). It is similar to multiple inheritance; however, without actually inheriting anything. In addition to composition, all entities inherit from a base Entity class which brings OnDestroyed event for each entity. This event can be listened to know when specific entity on completely removed from the Scene.

In this mixin implementation, the interface segregation principle can be seen easily as different behaviors are defined by interfaces. The parent class for specific entity implements each of the behavior interfaces the entity requires and also has a dependency on each of those interfaces. The interface implementations on the parent class forward each method, property and event to the corresponding dependency. This also enforces the usage of dependency inversion. However, this may create a lot of boilerplate code for the parent class, however, it also brings the ability to change the behavior logic without needing to modify the parent class. Another benefit is that the exactly same behavior logic can also be used by multiple entities. Additionally, the behavior logic can be changed in runtime. Figure 34 demonstrates the mixin's parent class (Photon) and its composed capabilities of IMovable, IRotatable, IKillable and IWave.

```
public class Photon : Entity, IKillable, IMovable, IRotatable, IWave
{
    private IKillable killable; // Changeable kill logic
    private IRotatable rotatable; // Changeable rotation logic
    private IMovable movement; // Changable movement logic
    private IWave wave; // Changeable wave logic

    [Inject]
    public void Construct(IMovable movement, IRotatable rotatable, IWave wave,
IKillable killable)
    {
        this.killable = killable;
        this.movement = movement;
        this.rotatable = rotatable;
        this.wave = wave;
    }

    public void Rotate(Vector3 amount)
    {
        rotatable.Rotate(amount); // Method forwarding
    }

    ...
}
```

Figure 34. Composition of the photon entity

When each entity implements several interfaces, it also makes it easy to get every entity that share same behavior. For example, to get each entity that can rotate or move. The EntityManager can be used to get every instance as seen in Figure 35. It

also makes it clear for the developer to understand what each entity is capable of doing. When systems are favored to depend on interfaces instead of concrete entity classes, removing a behavior from an entity, automatically rules out the entity from the system. In an ideal case, removing a behavior from entity would not require any changes in any system. However, this is difficult to achieve.

```
var killables = entityManager.GetAll().OfType<IKillable>();
```

Figure 35. Example of fetching entities with specific capability

The Wall entity is slightly different as it is so simple that it does not do anything else except destroy any entity that touches it. However, even though it differs slightly from other entities, it would be very easy to extend its capabilities in the same way as others.

8.3.3 Wavelength Modifying

Wavelength modification of the photon emitters is considered as an exchange between a certain resource and the wavelength. Therefore, there is an interface called IExchangable that can be implemented to create a specific kind of exchange and by calling the Exchange-method of the instance will know how to perform an exchange. In the case of wavelength, there is ResourceWavelengthExchange instance that holds the reference to the IWave implementation and any IResource implementation. A factory can be used to create the exchange class instance for specific IWave implementation. For ResourceWavelengthExchange, the dependency injection bindings are set to always inject the Resource instance for energy.

The resources are implementations of the IResource interface. The resources can be used by calling the Spend method and gained by calling the Restock method. In an exchange, Spend method and Restock method are called based on the amount given as a parameter for the Exchange method. The negative value will cause Resource to be gained from the wavelength and vice versa.

Exchange logic is quite simple in the prototype, however, more complex exchange logic could be created with the same interface. For example, an exchange with ratio between two resources.

8.3.4 Game State

The game does not directly have a class that has the responsibility of changing or maintaining the state. The closest thing to such a system is the implementation of IRoundManager, which can be observed for round starting and ending. States for winning or losing do not exist but there are classes implementing the ICondition interface for those scenarios. Classes implementing this interface can be observed for the condition to be met. For example, there are WinCondition and LoseCondition classes which can be observed for player winning or losing. This allows new "states" to exist without modifying any existing code, however, by creating a new ICondition implementation which represents specific case which can be observed for triggering.

WinCondition triggers when a round ends and every receiver has received something. LoseCondition triggers when one or more receivers have not received anything, and there are no more charges left. Round ending is triggered by EndCondition which triggers when there are no more photons at the level or every single one of them has been received. In Figure 36, take from WinCondition class demonstrates the logic for winning the game.

```
private void RoundManager_OnRoundEnd(object sender, EventArgs e)
{
    if(AreAllReceiversReady(entities.GetAll().OfType<PhotonReceiver>()))
    {
        if (OnConditionMet != null) OnConditionMet(this, new EventArgs());
    }
}

private bool AreAllReceiversReady(IEnumerable<PhotonReceiver> receivers)
{
    return receivers.Count() == receivers.Where(x => x.HasReceivedAny-
thing).Count();
}
```

Figure 36. Snippet defining winning condition

8.3.5 Inspector Friendlyness

Every entity has its own Zenject installer script. These installer scripts are based on the MonoBehaviour class, and they can contain public fields that are exposed for the Inspector. The installer is placed in its own GameObject under the entity itself.

As seen in Figure 34, the photon is composed of implementations of IRotatable, IMovable, IWave interfaces and more. In Figure 37, it can be seen that there are settings exposed for the Inspector for many of those implementations in the installer.

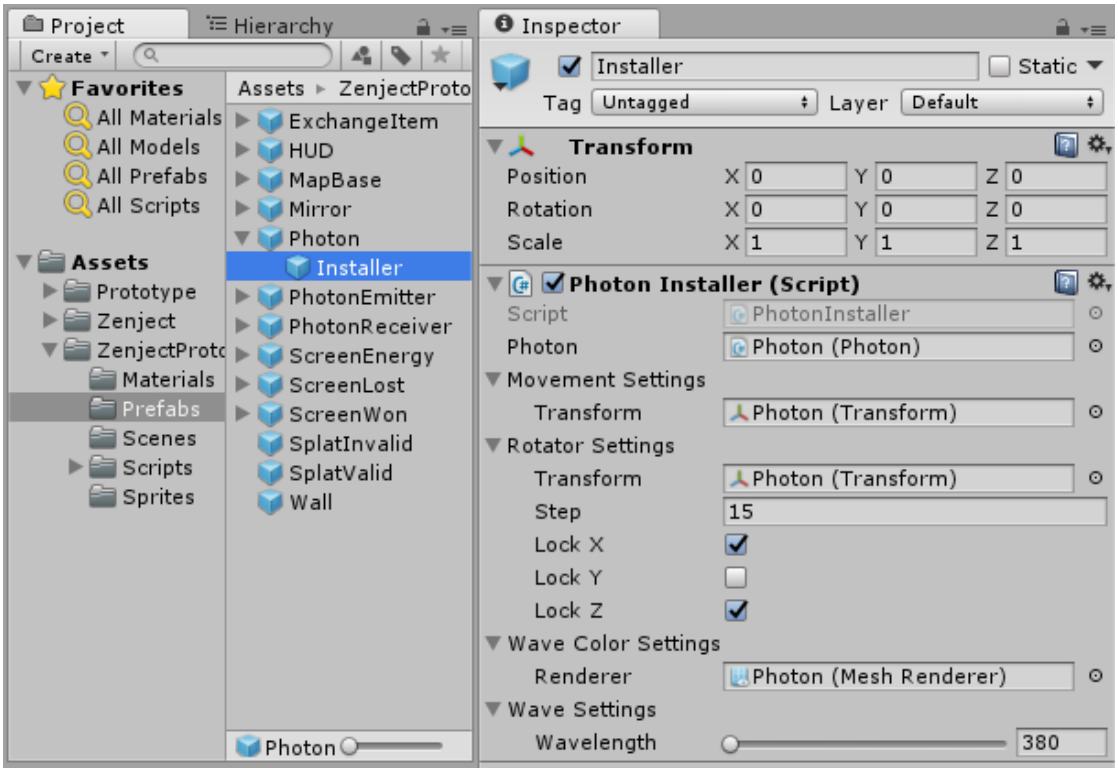

Figure 37. Configurable settings in the Inspector

The developer can use the Inspector to configure the objects that compose the entity. Values set are then injected to the objects. Bindings to inject the set values are defined in the installer script as seen Figure 38. The Figure 38 also shows that each implementation has its own Settings class, which makes it easier to add new fields to specific settings without needing to modify multiple files.

```
public class PhotonInstaller : MonoInstaller<PhotonInstaller>
{
    public Photon Photon;
    public LinearMovement.Settings MovementSettings;
    public Rotator.Settings RotatorSettings;
    public WaveColor.Settings WaveColorSettings;
    public Wave.Settings WaveSettings;

    public override void InstallBindings()
    {
        Container.BindInstance(WaveSettings)
        .WhenInjectedInto<Wave>();
        Container.BindInstance(WaveColorSettings)
        .WhenInjectedInto<WaveColor>();
        ...
    }
}
```

Figure 38. Photon entity's installer script

8.3.6 User Interface

The user interface (UI) is very simple and small and does not contain any complicated parts. However, the UI is a good example of Inversion of Control as it is the most outer layer. This means only the UI knows about the inner components such as WinCondition or LoseCondition. Nothing knows about the existence of the UI. The UI only observes inner components and makes changes on its own based on the observed events.

For example, when the player starts a round by clicking the button on the bottom of the screen, the button does not do anything else but tells the RoundManager to start a round. The RoundManager then spends one of the charge resources. Another UI component observes the charge resource and updates the value in the text element.

The rotation of emitters is also part of the UI layer. Mouse input is caught and used to rotate an emitter as seen in Figure 39.

```
Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
RaycastHit hit;
if (Physics.Raycast(ray, out hit))
{
    rotatable.LookAt(hit.point);
}
```

Figure 39. Snippet of photon emitter rotation logic with mouse

8.4 In Action

Figures 40 and 41 present the game as it is seen in the editor. In Figure 40, the game has not been started. The objects created at runtime are not present, placeholder texts are visible and the photon emitter and receivers do not have their colors set.

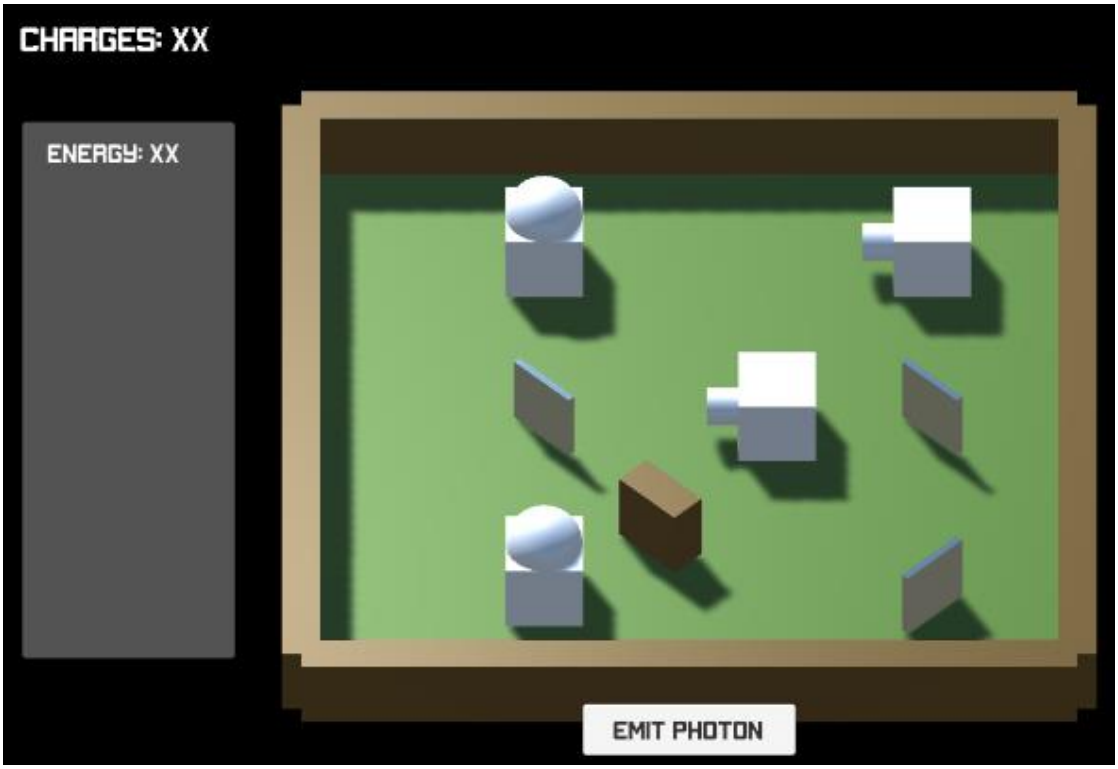

Figure 40. Screenshot of the final product before starting the game

In Figure 41, the game has been started in the editor. The player has rotated one of the photon emitters and changed the energy input for the photon emitters to change the wavelength. If the player clicked on the “Emit Photon” button, the player would win.

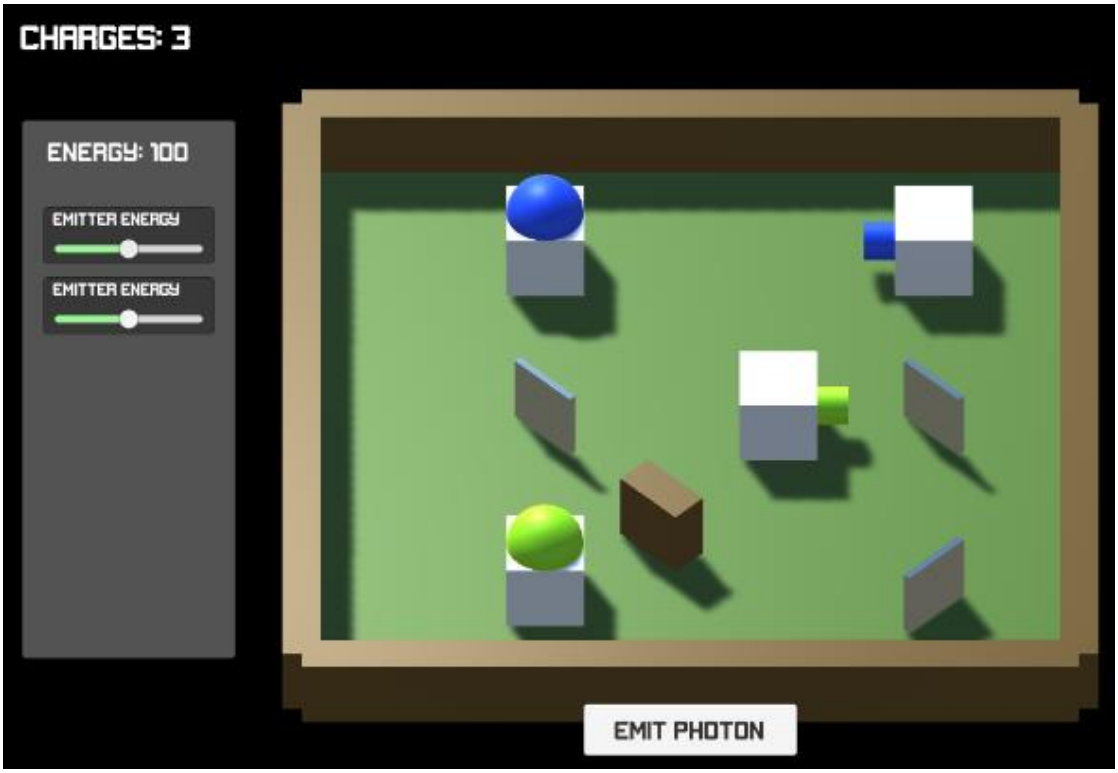

Figure 41. Screenshot of the final product while playing

9 Conclusions

The usage of dependency injection along with the SOLID principles as a solution for the dependency management is debatable. In Unity3D, a dependency injection framework is almost a mandatory requirement if considering to use dependency injection. The way Unity way works makes it difficult to use dependency injection without a framework. Mainly the problems are created by the lack of single entry point and MonoBehaviour instantiation. These problems increase the complexity of using the frameworks. The learning curve for dependency injection and especially the frameworks is already high and the increased complexity in Unity may make simpler solutions like the singleton pattern tempting.

However, the Zenject documentation (2017) said it well that when the project size grows, using singletons makes the code unwieldy. Good code is basically synonymous with loosely coupled code and to write loosely coupled code, you need to be aware

of the dependencies and code to interfaces. With singletons, it is not always clear what the dependencies are as singleton can be referred to anywhere in the code. With DI framework, the management of dependencies requires some more work, however, at the same time forces developers to be aware of the dependencies. It also forces to code to the interfaces. By declaring all the dependencies as constructor parameters, it basically means "in order for me to function, these contracts have to be fulfilled". These constructor parameters might not actually be interfaces or abstract classes; however, this does not matter because in an abstract sense, they are still contracts, which is not the case when creating them within the class or using global singletons.

Personally, I agree with the Zenject documentation that using dependency injection is worth it. Even if the design in the test project is not perfect and may have design flaws, it would not require huge changes in the code itself to change the way things work as everything is separated fairly well by their responsibilities, and all the dependencies are injected. However, dependency injection on its own is not enough. If I had not have tried to follow the SOLID principles at all, the benefits from dependency injection would not be as clear or even there. Class design plays a very important part in dependency management because it defines the dependencies between classes and interfaces. Dependency injection plays its own part in how the dependencies are resolved.

For quick prototyping, it might be better to use other methods like the singleton pattern. However, refactoring to dependency injection would be a wise choice. Prototyping with dependency injection might also work well, even though being slightly slower.

My previous experience with dependency injection has been in the pure form (Pure DI) without frameworks which may have caused partially inefficient usage of the framework. Like the usage of the ResourceContainer (Figure 32), which might be a result of not knowing enough about the framework. However, I think building a game on the current design would be quite enjoyable to work with and in overall, a fairly good level of modularity was reached. The UI is the most modular part as nothing depends on it. It can be removed without requiring any changes anywhere else.

Some of the less modular parts are caused by not using an interface for dependencies which are not too bad in the test project. For example, in Figure 36, the PhotonReceiver class is used directly even though using the generic IReceiver interface would increase modularity. It would be a very simple change but at the same time, this is a great example of why to use extra effort for abstraction.

References

Adic Documentation. 2016. File on Adic's GitHub repository, 10 December 2016. Accessed on 3 January 2017. Retrieved from https://github.com/intentor/adic/blob/master/README.md

Ancheta, F. 2015. *SOLID Review: Liskov Substitution Principle*. Blog post on Runtime Era website, 5 March 2015. Accessed on 13 December 2016. Retrieved from http://runtime-era.blogspot.fi/2015/03/solid-review-liskov-substitution.html

Baharestani, D. 2013. *Mastering Ninject for Dependency Injection*. Packt Publishing.

Chambers, J. 2016. *Upgraded Mono/.Net in Editor on 5.5.0b9*. Forum post on Unity3D website's forums, 27 October 2016. Accessed on 13 February 2017. Retrieved from https://forum.unity3d.com/threads/upgraded-mono-net-in-editor-on-5-5-0b9.438359/

Densmore, S. 2004. *Why Singletons are Evil*. Blog post on MSDN website, 25 May 2004. Accessed on 12 December 2016. Retrieved from https://blogs.msdn.microsoft.com/scottdensmore/2004/05/25/why-singletons-are-evil/

Durand, W. 2013. *From STUPID to SOLID Code*. Blog post on William Durand website, 30 July 2013. Accessed on 12 December 2016. Retrieved from http://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/

Fowler, M. 2005. *Inversion of Control*. Article on MartinFowler's website, 26 June 2005. Accessed on 25 February 2017. Retrieved from https://martinfowler.com/bliki/InversionOfControl.html

Games Made with Unity. N.d. Page on Unity3D's website. Accessed 30 March 2017. Retrieved from https://madewith.unity.com/en/games

Hight, J. 2013. *Diablo 3 Auction House Update*. News article on Diablo 3's official website, 17 September 2013. Accessed on 12 December 2016. Retrieved from http://us.battle.net/d3/en/blog/10974978/diablo%C2%AE-iii-auction-house-update-9-17-2013

How is game development different from other software development. 2011. Answer on Stack Exchange's Game Development website, 27 February 2011. Accessed on 12 December 2016. Retrieved from http://gamedev.stackexchange.com/questions/9074/how-is-game-development-different-from-other-software-development/9089#9089

Inversion of Control. 2016. Article on Wikipedia, 17 November 2016. Accessed on 12 December 2016. Retrieved from https://en.wikipedia.org/wiki/Inversion_of_control

Jenkov, J. 2014. *Dependency Injection Benefits*. Article on Jenkov's website, 26 May 2014. Accessed on 27 February 2017. Retrieved from http://tutorials.jenkov.com/dependency-injection/dependency-injection-benefits.html

Kainulainen, P. 2013. *Why I Changed My Mind About Field Injection*. Blog post on Petri Kainulainen's Blog, 26 June 2013. Accessed on 13 December 2016. Retrieved from https://www.petrikainulainen.net/software-development/design/why-i-changed-my-mind-about-field-injection/

Kaiser, R. 2016. *Diablo III Used to Suck. Here's How It Got Good*. Article on Inverse's website, 19 January 2016. Accessed on 12 December 2016. Retrieved from https://www.inverse.com/article/10361-diablo-iii-used-to-suck-here-s-how-it-got-good

Mandalà, S. 2012a. *Inversion of Control with Unity3D*. Blog post on Seba's Lab website, 30 September 2012. Accessed on 12 December 2016. Retrieved from http://www.sebaslab.com/ioc-container-for-unity3d-part-1/

Mandalà, S. 2012b. *Inversion of Control with Unity3D*. Blog post on Seba's Lab website, 14 November 2012. Accessed on 2 March 2017. Retrieved from http://www.sebaslab.com/ioc-container-for-unity3d-part-2/

Martin, R. 2012. *The Single Responsibility Principle*. Presentation at Norwegian Developer's Conference. Watchable on Youtube. Accessed on 13 December 2016. Retrieved from https://www.youtube.com/watch?v=Gt0M_OHKhQE

Martin, R. 2014. *SOLID Principles of Object Oriented & Agile Design*. Presentation at Yale School of Management. Watchable on Youtube. Accessed on 25 February 2017. Retrieved from https://www.youtube.com/watch?v=QHnLmvDxGTY

Martin, R. N.d. *Principles of OOD*. Page on Martin's website: But Uncle Bob. Accessed on 13 December 2016. Retrieved from http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Mixin. 2017. Article on Wikipedia, 8 February 2017. Accessed on 24 February 2017. Retrieved from https://en.wikipedia.org/wiki/Mixin

Seemann, M. 2011. *Composition Root*. Blog post on Seemann's website: Ploeh Blog, 28 July 2011. Accessed on 24 February 2017. Retrieved from http://blog.ploeh.dk/2011/07/28/CompositionRoot/

Seemann, M. 2014. *Pure DI*. Blog post on Seemann's website: Ploeh Blog, 10 June 2014. Accessed on 24 February 2017. Retrieved from http://blog.ploeh.dk/2014/06/10/pure-di/

Shore, J. 2006. *Dependency Injection Demystified*. Blog post on Shore's website, 22 March 2006. Accessed on 13 December 2016. Retrieved from http://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html

Shvets, A. N.d. *Decorator Design Pattern*. Page on Sourcemaking's website. Accessed on 16 March 2017. Retrieved from https://sourcemaking.com/design_patterns/decorator

Single Responsibility Principle. 2016. Article on Wikipedia website 8.12.2016. Accessed on 13 December 2016. Retrieved from https://en.wikipedia.org/wiki/Single_responsibility_principle

Skeet, J. 2011. *Implementing the Singleton Pattern in C#*. Page on CSharp In Depth's website, 12 February 2011. Accessed on 12 December 2016. Retrieved from http://csharpindepth.com/Articles/General/Singleton.aspx

Skrchevski, B. 2015. *High Cohesion, Loose Coupling*. Post at The Bojan's Blog 8.4.2015. Accessed on 12 December 2016. Retrieved from https://thebojan.ninja/2015/04/08/high-cohesion-loose-coupling/

SOLID. 2017. Article at Wikipedia 7.2.2017. Accessed on 27 February 2017. Retrieved from https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

Unity - Awake and Start. N.d. Video tutorial. Watchable at Unity3d.com. Accessed on 2 March 2017. Retrieved from https://unity3d.com/learn/tutorials/topics/scripting/awake-and-start

Unity - Manual: Object.FindObjectOfType. N.d. Page on Unity3D's documentation website. Accessed on 25 February 2017. Retrieved from https://docs.unity3d.com/ScriptReference/Object.FindObjectOfType.html

Unity - Manual: The Hierarchy Window. N.d. Page on Unity3D's documentation website. Accessed on 2 March 2017. Retrieved from https://docs.unity3d.com/Manual/Hierarchy.html

Zenject Documentation. 2016. File on Zenject's GitHub repository, 2 December 2016. Accessed on 3 January 2017. Retrieved from https://github.com/modesttree/Zenject/blob/master/README.md