Roni Jokinen

Contextual dependency injection container in third-party game engine

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology Degree Programme

Thesis

19 April 2018

Metropolia

Author Title Number of Pages Date	Roni Jokinen Contextual dependency injection container in third-party game engine 50 pages 19 April 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Game applications
Instructor	Miikka Mäki-Uuro, Lecturer

Bachelor's thesis aimed to explore principles of contextual dependency injection create an implementation of such dependency injection container called Dependency Injection Tool, which is designed to work in Unity game engine. In addition to dependency injection capabilities, this container was supposed to implement additional development tools and model–view–controller architecture model to further utilize contextual dependency injection pattern.

During development major design problems considering type safety, initialization order and creation of new instances were faced. These problems were solved by using different software libraries available in C#.

It was observed that hierarchical contextual dependency injection enables component collections to completely change their behavior based on their contexts. Model–view–controller architecture model created tightly defined interfaces which can be extended and used to reduce the time required to design complex objects. Relationships between Model, view and controller were observed to be straight forward and flexible, allowing software design to be started from the data associated with the component. Controller would transform this data using tasks called commands and view would act upon changes to the data model.

Dependency Injection Tool can be used to modularize, add testability and rapidly change implementations. Further developed version of this dependency injection container can be used with any project using Unity engine.

Keywords	game development, Unity, dependency injection, inversion of control, software architecture, software design patterns, context-oriented programming

Tekijä Otsikko Sivumäärä Aika	Roni Jokinen Kontekstipohjainen riippuvuusinjektiosäiliö kolmannen osapuolen pelimoottorissa 50 sivua 19.4.2018
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	tieto- ja viestintätekniikka
Ammatillinen pääaine	pelisovellukset
Ohjaaja	lehtori Miikka Mäki-Uuro

Insinöörityön tarkoitus oli perehtyä kontekstipohjaisen riippuvuusinjektion perusperiaatteisiin ja toteuttaa riippuvuusinjektiosäiliö (Dependency Injection Tool), joka toimii Unity-pelimoottorissa. Kontekstipohjaisen riippuvuusinjektion lisäksi tämän säiliön tarkoitus oli toteuttaa erilaisia kehitystyökaluja ja malli–näkymä–ohjain-arkkitehtuurityyppi, joka auttaa kontestipohjaisen riippuvuusinjektion hyödyntämisessä.

Kehityksen aikana esiintyi suunnitteluongelmia liittyen tyyppivarmuuteen, alustusjärjestykseen ja uusien instanssien luomiseen, ja ne onnistuttiin ratkaisemaan käyttämällä erilaisia valmiita luokkakirjastoja.

Työssä huomattiin että hierarkkinen kontekstipohjainen riippuvuusinjektio mahdollistaa komponenttikokoelmien käytöksen muuttamisen kontekstista riippuen. Malli–näkymä–ohjain-arkkitehtuurityyli loi tiukasti määritellyt laajennettavat rajapinnat, jotka vähentävät kompleksisten olioiden suunnitteluun kuluvaa aikaa. Malli–näkymä–ohjain-arkkitehtuurityylin eri osien väliset suhteet olivat selkeitä ja joustavia, mikä mahdollisti ohjelman suunnittelun aloittamisen komponenttien tietomallin määrittelystä. Ohjain muuttaa mallin tietoja käyttäen tehtäviä, jotka perustuvat komento-suunnittelumalliin, ja näkymä muuttuu mallissa tapahtuvien muutosten mukaisesti.

Insinöörityön yhteydessä ohjelmoitu kontekstipohjainen riippuvuusinjektiosäiliö auttaa ohjelman modularisoimisessa, testattavuudessa ja toteutuksien nopeassa muuttamisessa. Säiliötä voidaan pidemmälle kehitettynä hyödyntää missä tahansa Unity-pelimoottorilla tehdyssä pelissä.

Avainsanat	pelinkehitys, Unity, riippuvuusinjektio, ohjauksen kääntöperiaate, ohjelmistoarkkitehtuuri, ohjelmiston suunnittelumallit, konteksti-orientoitunut ohjelmointi

Contents

List of Abbreviations

DI Dependency Injection design pattern. Whereby one object supplies the other object its dependencies.

IoC Inversion of control. Software architecture with inverted control design delegates implementation decisions from business logic to a generic framework.

DIT Dependency injection tool. Name of the framework presented in this report.

LSP Liskov's substitution principle. Describes rules how deriving type should behave when handled as less deriving type.

DLL Dynamic-link library. Microsoft's implementation of the shared library concept.

MVC Model–view–controller. Software architecture principle which separates business logic, data and user interface.

Terminology

Dependency Module, class or library which is required by the dependent object to function properly.

Unity 3D Game engine by Unity Technologies.

Scene Represents a level or the game world and/or certain collection of Gameobjects.

Gameobject Represents composition of scripts in Unity engine. Has position in world space of a scene.

Entity Collection of different objects and scripts. Common appointment for meta-objects consisting of multiple other modules.

Field	Instance variable of a class with no separate get or set method.
Property	Syntactic sugar for fields with getter and/or setter generated by the compiler. Can be employed to work like a method. Supports assignment if setter is provided
Method	Code block that contains series of statements. Can invoked with multiple parameters.
Constructor	Default Initialization method for classes with unique syntax.
Game loop	Main loop of the program which updates all object states to their next state. Each loop cycle is called a tick and they can be compared to frames. Loops multiple times each second.

1 Introduction

This work aims to introduce how dependency injection framework can be implemented and used in commercial Unity environment. Project will highlight and introduce how inversion of control (IoC) can be implemented and how changing the control flow of an application allows developers to write modular and reusable code to implement high levels of flexibility and complex automation.

Dependency Injection Tool (DIT) is and has been developed to meet personal needs in software development. The core framework implements lower level operations which then allow further automation and abstraction to be built, reducing the amount of boilerplate code needed to complete common tasks. Model–view–controller (MVC) implemented in the framework aims to reduce the time needed to design complex encapsulated objects by simplifying the programs architectural model to very specific interfaces.

Chapter 2 shows basic examples of how the framework can be used in Unity's environment.

Chapter 3 will introduce all technical requirements needed to understand DIT and dependency injection in general. This chapter will briefly explain what dependency injection is and what it is useful for.

Chapter 4 explains the largest design obstacles encountered during development. The problems are briefly introduced and a working solution used in DIT is given to each of them.

Chapter 5 introduces the features, design and implementation of the library. All core and optional features of this framework are explained in this chapter. The framework implements hierarchical contextual dependency injection based on Unity's Gameobject hierarchy, which enables many useful patterns.

Chapter 6 will evaluate efficiency and extensiveness of the framework and summarizes everything presented in this report.

2 Project Introduction

2.1 Purpose

DIT was built to decrease the amount of thinking and iteration required to create robust, flexible and reusable code by implementing a type safe interface for DI and encouraging the user to create independently functioning entities, which are able to control and execute their own complex initialization logic. Entities can self-resolve cross contextual external dependencies without knowing the execution or loading order of itself or the object(s) the entity is dependent on.

In this chapter, main features of DIT are briefly introduced in form of examples of how the framework can be used and how the implementation of said features can be changed without rewriting any previous code, apart from the class which configures its dependencies.

Multiple libraries were used as a reference during the development of DIT, with most notable of them being Strange IoC and Zenject, which are both designed to work with Unity engine. Reason for writing DIT was to ease and automate personal software development, and therefore it was built to last and to be extended on right from the beginning. The program code has been refactored countless times to support more complex and wider range of operations, which allow further extensibility of the framework.

2.2 Examples

In order to understand the advantages of DIT, it is required to be familiar with common patterns available in the framework. Simple test cases are needed to demonstrate how the framework can be used.

Inside FirstTestEntityContext's RegisterBindings method, it is declared that any type of object which is dependent on IFirstDependency and is created in this context, will receive an instance of FirstDependency to fulfill it. Inside OnStart method, new instance of FirstTestEntity is created. Before FirstTestEntity is completely constructed and its constructor is executed, the context will attempt to fulfill all of its dependencies.

```
public interface IFirstDependency {}
public class FirstDependency : IFirstDependency {}

public class FirstTestEntityContext : SceneContext
{
      protected override void RegisterBindings()
      {
            Binder.Bind<FirstTestEntity>().To<FirstTestEntity>();
            Binder.Bind<IFirstDependency>().To<FirstDependency>();
      }

      protected override void OnStart()
      {
            FirstTestEntity entity =
                  InstanceProvider.GetInstance<FirstTestEntity>();
      }

}
```

Figure 1. RegisterBindings method is used for configurating dependencies in contexts. OnStart method is run last in contexts initialization process.

```
public class FirstTestEntity
{
      public FirstTestEntity()
      {
         Debug.Log(First.GetType());
      }

      [Inject]
      private IFirstDependency First;
}
```

Figure 2. FirstTestEntity has a dependency on IFirstDependency. Dependencies are declared with Inject attribute, which can be placed on top of fields and properties.

As the context registered a binding to IFirstDependency, field named “First” now has a reference to an instance of FirstDependency. After dependencies have been fulfilled, the constructor attempts to print the type of the variable “First” onto console.

```
Debug.Log: DIT.TestClasses.FirstDependency
```

For now, the amount of boilerplate code needed to reach very simple outcome seems very excessive, but in order to utilize contextual DI properly, more complex requirements are needed. Such examples are shown in chapter 4.

DIT can create bindings for multiple different types, as long as the bound interface or class is properly implemented or inherited by the class it is bound to. This rule is ensured by the Binder, which has type safe binding interface.

```
public interface IFirstDependency {}
public interface ISecondDependency {}
public interface IThirdDependency {}

public class FirstDependency : IFirstDependency {}
public class SecondDependency : ISecondDependency {}
public class ThirdDependency : IThirdDependency { }

public class SecondTestEntityContext : SceneContext
{
      protected override void RegisterBindings()
      {
            Binder.Bind<SecondTestEntity>().To<SecondTestEntity>();
            Binder.Bind<IFirstDependency>().To<FirstDependency>();
            Binder.Bind<ISecondDependency>().To<SecondDependency>();
            Binder.Bind<IThirdDependency>().To<ThirdDependency>();
      }

      protected override void OnStart()
      {
            SecondtestEntity entity =
                  InstanceProvider.GetInstance<SecondTestEntity>();
      }

}

public class SecondTestEntity
{

      public SecondTestEntity()
      {
            Debug.Log(First.GetType());
            Debug.Log(Second.GetType());
            Debug.Log(Third.GetType());
      }


      [Inject]
      public IFirstDependency First;

      [Inject]
      protected ISecondDependency Second { get; set; }

      [Inject]
      private IThirdDependency Third { get; set; }
}
```

Figure 3. SecondTestEntity is dependent on multiple different interfaces. Implementation for First, Second and Third variables can be resolved regardless of having different accessibility levels.

After all dependencies have been fulfilled, SecondTestEntity's constructor is executed and it attempts to print the types of all three dependencies.

```
Debug.Log: DIT.TestClasses.FirstDependency
Debug.Log: DIT.TestClasses.SecondDependency
Debug.Log: DIT.TestClasses.ThirdDependency
```

In addition to multiple different bindings, DIT can add constraints on bindings to allow further complexity in object hierarchies (see 5.1.2). The user can set constraints to which

types the binding applies to, and which name the binding requires. Names can be used to differentiate two dependencies with shared type.

```
public interface IFirstDependency {}
public class FirstDependency : IFirstDependency {}
public class DerivingDependency : FirstDependency {}
public class DefaultDependency: IFirstDependency {}

public enum TestEnum
{
      First,
}

public class ThirdTestEntityContext : SceneContext
{
      protected override void RegisterBindings()
      {
            Binder.Bind<ThirdTestEntity>().To<ThirdTestEntity>();

            Binder.Bind<IFirstDependency>()
                        .To<DerivingDependency>()
                        .ToName("First");

            Binder.Bind<IFirstDependency>()
                        .To<FirstDependency>()
                        .ToName(typeof(FirstDependency));

            Binder.Bind<IFirstDependency>().To<DefaultDependency>();
      }

      protected override void OnStart()
      {
            ThirdTestEntity entity =
                  InstanceProvider.GetInstance<ThirdTestEntity>();
      }
}

public class ThirdTestEntity
{
      public ThirdTestEntity()
      {
            Debug.Log(First.GetType());
            Debug.Log(SecondFirst.GetType());
            Debug.Log(ThirdFirst.GetType());
      }

      [Inject("First")]
      private IFirstDependency First;

      [Inject(typeof(FirstDependency))]
      private IFirstDependency SecondFirst;

      [Inject(TestEnum.First)]
      private IFirstDependency ThirdFirst;
}
```

Figure 4. ThirdTestEntity declares multiple dependencies of the same type. Inject attribute allows the user to add name identifier for dependencies which enables additional constraints to be used. If no dependency with a matching name were found, DIT attempts to fulfill the dependency using binding with no name.

When the constructor of ThirdTestEntity is executed, following message is seen on the console:

```
Debug.Log: DIT.TestClasses.DerivingDependency
Debug.Log: DIT.TestClasses.FirstDependency
Debug.Log: DIT.TestClasses.DefaultDependency
```

In these three examples, basic usage of DIT was shown. Declaring and injecting dependencies is the main functionality of DIT, and on top of this feature, more complex patterns can be built.

Next chapter will introduce basic technical requirements to understand more advanced usages of DIT. Additionally, the chapter will briefly explain principles behind dependency injection and inversion of control.

3 Technical Pretext

3.1 Design Patterns

In software engineering, a design pattern is generally applicable repeatable solution to a common design problem. Design pattern is not design itself and cannot be transformed directly into code, but rather a template or description for certain type of solution. [3]

Design patterns can speed up development process by providing tested and proven solutions to complex problems. Effective software design requires considering issues which are not seen until later in development and the use of design patterns make them easier to predict. Design patterns also create common platform for software developers and architects, which helps with effective communication of ideas and understandability of code base. [3]

Design patterns can be separated into 3 different categories

- **Creational Patterns** abstract creation of new objects. They help to separate process of creating objects from how they are composed and represented. [8]
- **Structural Patterns** describe ways to compose objects to implement new functionality. They are concerned with how objects are composed to form broader entities. [8]

- **Behavioral Patterns** are concerned with responsibilities and algorithms. They describe patterns of objects and communication between them. [8]

DIT can be considered being creational pattern as it deals with creating and configuring objects. DIT can be used in the process of implementing many different patterns, as it attaches itself to the initialization process of all objects.

In next sub chapters, few of the most important patterns are introduced.

3.1.1 Bridge

Bridge pattern decouples abstraction from the implementation, so that each implementation can vary independently [8]. As previously seen in the examples (see 2), classes were dependent on interfaces rather than concrete implementations. This allows the developer to change the actual implementation of dependencies by providing different generic type argument during binding registration.

```
public class BaseClass : IInterface {}
public class DerivingClass : BaseClass {}
public interface IInterface {}
```

Figure 5. BaseClass and DerivingClass implement IInterface, therefore IInterface acts as a bridge to both of the implementations.

Abstract and base classes can be used as interfaces to different implementations, but the limitations of programming languages with no multi inheritance support complicate this process. As a class can implement as many interfaces as needed and one can only inherit one other class, using interfaces to create a common bridge between different implementation is powerful tool of abstraction.

```
BaseClass dependency = new BaseClass();
dependency = new DerivingClass();
IInterface bridge = dependency;
```

Figure 6. LSP states that if module of a program requires an object of type BaseClass, then the reference can be substituted with an object of type DerivingClass without affecting functionality of the program [11].

3.1.2 Singleton

Singleton ensures there’s only one instance of a class and a common access point is provided to it [8]. Dependency injection can be employed to solve this pattern in very

simple way. As shown in chapter 2, all dependencies have to be registered in order for them to be properly fulfilled. Adding the possibility of making any dependency a singleton allows every dependent object to receive a same instance of the object they are dependent on.

```
public interface ISharedDependency { }
public class SharedDependency : ISharedDependency { }


public class ReceiverEntity
{
      public ReceiverEntity()
      {
            Debug.Log(SharedDependency.GetHashCode());
      }

      [Inject]
      private ISharedDependency SharedDependency;
}
```

Figure 7. Debug.Log method calls for SharedDependency.GetHashCode. GetHashCode method returns a numeri value that is used to identify and insert objects in collections based on hash. GetHashCode method can be used to quickly check object equality [9].

```
public class SingletonContext : SceneContext
{
      protected override void RegisterBindings()
      {
            Binder.Bind<ReceiverEntity>().To<ReceiverEntity>();
            Binder.Bind<ISharedDependency>()
                  .To<SharedDependency>()
                  .AsSingleton();
      }

      protected override void OnStart()
      {
            var instance0 = InstanceProvider.GetInstance<ReceiverEntity>();
            var instance1 = InstanceProvider.GetInstance<ReceiverEntity>();
      }
}
```

Figure 8. ISharedDependency is bound to SharedDependency and AsSingleton. Binding something to a singleton means that every ISharedDependency requested in this context is guaranteed to be the same instance of the same object.

When OnStart method is executed, following message appears in console window:

```
Debug.Log: 831735137163
Debug.Log: 831735137163
```

When two different instances of ReceiverEntity were made and both instances received the same instance of SharedDependency, as it was registered as singleton. DIT implements contexts which can have local singletons, and multiple identical contexts with local

singletons can exist in the same scene. As more than one instance of the same object can exist in unrelated contexts, the pattern could be called Multiton [10].

3.1.3 Strategy

Strategy pattern solves the problem of changing algorithm or implementation for different users or for certain action during run time. Interface for the algorithm is defined in terms of input and/or output which enables the algorithm to be changed to any class implementing the matching interface [8].

```
public interface IJumpAction
{
    void Execute();
}

public class DoubleJump : IJumpAction
{
    public void Execute()
    {
        Debug.Log("Double jump");
    }
}

public class NormalJump : IJumpAction
{
    public void Execute()
    {
        Debug.Log("Normal jump");
    }
}

public class GameEntity
{
    [Inject]
    private IJumpAction JumpAction;

    public void Jump()
    {
        JumpAction.Execute();
    }
}

public class StrategyContext : SceneContext
{
    protected override void RegisterBindings()
    {
        Binder.Bind<GameEntity>().To<GameEntity>();
        Binder.Bind<IJumpAction>().To<NormalJump>();
    }

    protected override void OnStart()
    {
        var entity = InstanceProvider.GetInstance<GameEntity>();
        entity.Jump();
    }
}
```

Figure 9. IJumpAction is bound to Normal jump. IJumpAction is an abstract interface, a bridge pattern for this strategy. NormalJump and DoubleJump implement the interface and JumpAction can be changed to either one of the implementations during run time.

In this case, executing Jump method of GameEntity would produce the following output on the console:

```
Debug.Log: Normal jump
```

When the strategy of IJumpAction is required to change to some other implementation, StrategyContext's RegisterBindings method can be modified in the following manner:

```
public class StrategyContext : SceneContext
{
    protected override void RegisterBindings()
    {
        Binder.Bind<GameEntity>().To<GameEntity>();
        Binder.Bind<IJumpAction>().To<DoubleJump>();
    }

    protected override void OnStart()
    {
        var entity = InstanceProvider.GetInstance<GameEntity>();
        entity.Jump();
    }
}
```

Figure 10. IJumptAction is now bound to DoubleJump.

When the program is executed, following message is shown on the console window:

```
Debug.Log: Double jump
```

Changing strategy allows the developer to change GameEntity's implementation of jump by only modifying one generic argument in a method call, without ever changing any program code inside GameEntity.

3.2 Unity

In this chapter, very basic overview of Unity engine and editor is given.

Unity is commercial game engine made by Unity Technologies. Unity game engine supports multiple platforms such as Android, PC, Linux and iOS. Unity Technologies offer additional services integrated in the game engine, such as advertising and analytics solutions.

Unity game engine provides multiple features, such as extensible editor, art tools, design tools, graphics rendering, performance profiler and multiplayer support. Unity engine supports JavaScript and C# programming languages. Unity Technologies and third-party contributors have produced extensive amount of beginner friendly tutorials which help new developers to start learning Unity.

Unity editor allows developers to create, delete and manipulate objects in the game world, configure different options, simulate the created game and build the program.

3.2.1 Hierarchy

In order to understand hierarchy, one has to understand what scenes are first. Scene is a “level”, “space” or “context” which holds data about the environment and actors called Gameobjects. In image 1 hierarchy has ViewModelContext scene currently open. Below this scene in the hierarchy, Main Camera, Directional Light and ViewModelContext can be seen. These are called Objects or Gameobjects, latter being more deriving type of Object. For sake of clarity, all Objects in the scene are called Gameobject further in the report.

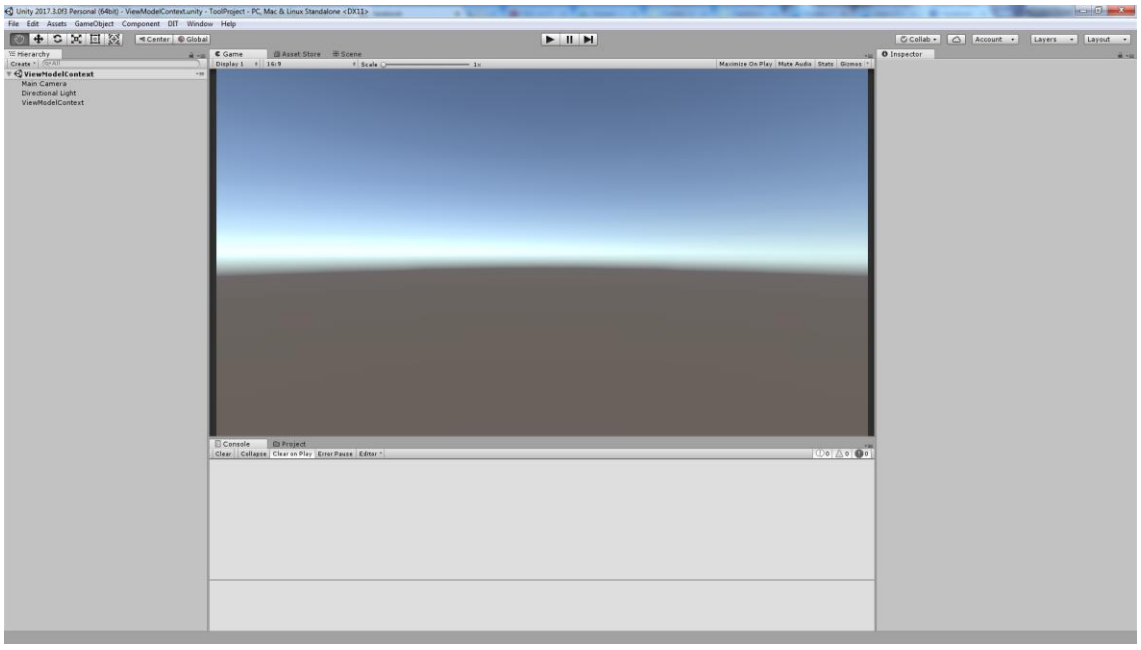


Image 1. Overview of Unity editor. Hierarchy (left), console window (bottom), project folder (bottom), inspector (right).

Gameobject are objects which can have multiple components and have position in world space of a scene. Component system allows one Gameobject to have multiple different properties, from moving and shooting to having a collider.

Gameobjects operate on game tick system, which refers to executing update method once for each Gameobject and its associated scripts during each iteration of game loop. One iteration of game loop can be considered one tick; multiple ticks are run each second. Game loops work similarly to graphic rendering, during each “frame” or “tick”, all computation is done to update the game to its new state.

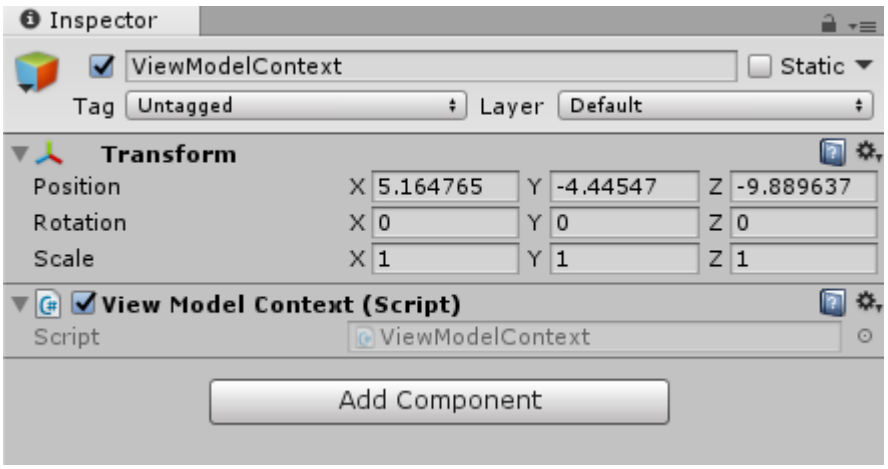
Transform component allows each Gameobject to have parent(s) or children and all Gameobjects in the scene form the hierarchy.

3.2.2 Console window

Console window shows messages generated by Unity engine. When the developer has compiler error in their code, console window will show a message where such compiler error occurred. In addition to error logging, console window can show messages and warnings as demonstrated in chapter 2. [1]

3.2.3 Inspector

Inspector shows the details of the chosen file in project folder or Gameobject in the hierarchy. If Gameobject is chosen, components attached to the Gameobject can be seen and modified.



3.2.4 MonoBehaviour

MonoBehaviour is the base class for unity scripts [2]. MonoBehaviours work quite differently from normal C# scripts, as they are directly tied to the Gameobjects which they are attached to. Script deriving from MonoBehaviour implement many default tools needed for Unity's component and tick system.

MonoBehaviours can implement Unity event functions by declaring new methods with certain names. Some of the most common methods used in Unity development are listed below.

- **Awake()** is run directly after the object is created, provided the game object is in active state.
- **Start()** is run on the next game tick after creation.
- **Update()** is executed on every game tick.
- **OnCollisionEnter(Collider col)** is triggered when another Gameobject with collider hits the collider of the Gameobject this script is attached to.

DIT aims to avoid using MonoBehaviours for business logic as there are certain limitations to using them.

3.3 Dependency Injection and Inversion of Control

Dependency is an implementation required by another object to function properly. Most objects rely on different components or services to implement the functionality they were

designed to. Such dependencies can be fulfilled by using IoC container, creational design patterns or by using service locator.

Creational design patterns are design patterns that deal with problems regarding object creation, trying to create them in manner which best suits the situation. Creational design patterns attempt to control any complex creational processes. [3]

Service locator is a design pattern which allows objects to request for service implementations and the locator will provide them with one. This leaves the burden of fulfilling dependencies to the dependent object itself. [4]

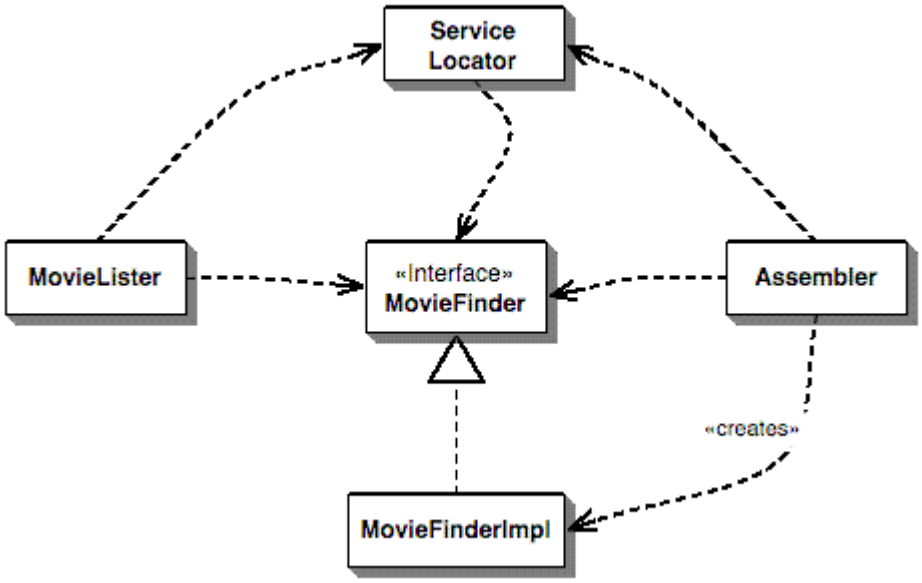


Image 2. MovieLister is dependent on MovieFinder. MovieLister explicitly requests ServiceLocator for MovieFinder. This is a design pattern called Service Locator. [4]

To lessen the amount of boiler plate code and complex dependencies needed, dependency injection can be employed.

The idea of dependency injection is to have a separate part of the program, a creator object for instance, to assemble objects with dependencies and populate their dependent fields with appropriate implementations [4].

As the responsibility of dependency fulfillment and control of the creation process in general has been shifted to the Assembler as shown in Image 3, Inversion of Control is therefore implemented [4].

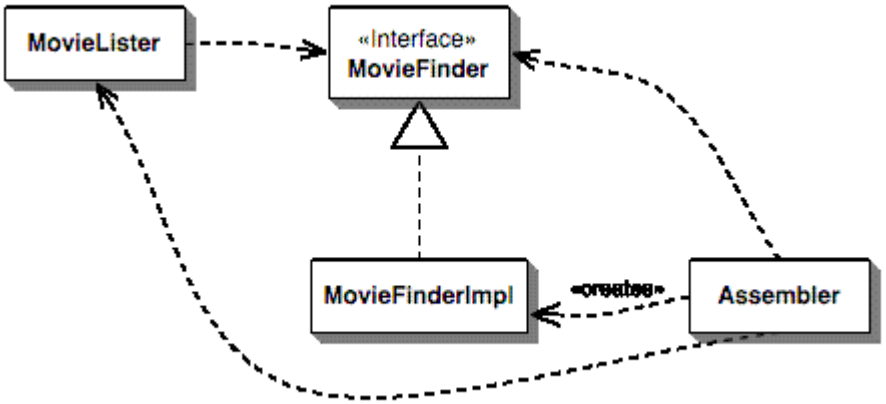


Image 3. MovieLister is dependent on MovieFinder. Assembler object will detect the dependency when creating a MovieLister and fulfills the dependency with MovieFinderImpl [4].

3.4 Reflection

> Reflection is the process of reasoning about and/or acting upon oneself. [5]

In C# programming language, reflection is listed under namespace System.Reflection. The library together with System.Type enables the program to obtain information about loaded assemblies and types defined within them. This information can be used to create new instances, invoke methods and change values of an object at run time. [6]

In strongly typed programming languages such as C#, reflection can be employed to implement complex behavior based on inspecting types during run time. In DIT, reflection is mainly used to look for properties with Inject attribute and for finding default constructor with no parameters.

System.Reflection.Emit is additional reflection library which allows the creation and building of new types during run time. It is important to understand that reflection is relatively expensive and error prone operation compared to normal type safe operations, therefore it should be used with caution.

3.4.1 Attribute

Attributes associate system or user-defined information with target elements. A target element can be a class, constructor, delegate, assembly, enum, event, field, interface,

method, portable executable file module, property, parameter, return value, struct or another attribute. [7]

Information provided by an attribute is also known as metadata. This metadata can be inspected using reflection and the data can be used to control how the program processes data or how the application itself is maintained. [7]

Attribute class can be extended by inheriting from it, allowing user-defined metadata. Attributes can have constructors which can receive parameters as seen in Figure 11.

```
[Inject(“First”)]
private IFirstDependency First;
```

Figure 11. Inject above the variable is an attribute.

Next chapter will explain common design problems encountered during the development.

4 Design Problems

This chapter will explain some of the most important design problems encountered during development of DIT. Most of these problems existed due to limitations or lack of features in C# or purely because of the nature of Unity engine.

4.1 Instance Creation

Most conventional way of creating new instances is to use new

```
Dependency dep = new Dependency();
```

It is very hard to create a binding which knows the type of the instance it wants to create at compile time. Most certainly this should be possible, but this is not actually desired, as there are other problems which cannot be solved when using new keyword for instance creation. Additionally, the binder interface already ensures all registrations are valid and type safe (see 4.2 and 5.1.1). This means other means of object creation have to be found.

Activator class in System namespace allows dynamic creation of objects during run time by passing the type of the object as a parameter. [13]

```
Type type = typeof(Dependency);
object dependency = Activator.CreateInstance(type);
Dependency dep = dependency as Dependency;
```

In C#, when new object is created using the new keyword or by using Activator library, constructor is executed before any changes can be made to the object. This is a common problem in DI containers, as constructor is run before any dependencies are set. In many cases, the developer wants to access dependencies during construction, and this is usually solved with either constructor injection or tagging method with an attribute which indicates a pseudo constructor. However, this creates extra boilerplate code and in case of using the attribute approach, resource expensive method lookup with reflection.

As it is possible to invoke constructors for objects by employing reflection (see 3.3), the problem can be reduced down to creating an object without invoking the constructor.

```
ConcreteObject = ConcreteType.GetUninitializedInstance();
```

Which invokes the following extension method:

```
public static object GetUninitializedInstance(this Type type)
{
      return FormatterServices.GetUninitializedObject(type);
}
```

In namespace System.Runtime.Serialization. This method allows the creation of new objects dynamically, without invoking any constructor(s) or without populating fields with set initial values.

Microsoft Developer Network describes this method in the following manner:

> Because the new instance of the object is initialized to zero and no constructors are run, the object might not represent a state that is regarded as valid by that object. The current method should only be used for deserialization when the user intends to immediately populate all fields. It does not create an uninitialized string, since creating an empty instance of an immutable type serves no purpose. [14]

When this method is used to create new objects, dependencies can be injected before constructor is executed. This results in desired initialization order and solves the problem of creating objects, fulfilling their dependencies and initializing them correctly.

4.2 Type Safety

Type safe code accesses only the memory locations it has been authorized to access. Type safe code cannot read values from other object's private fields and can access types only in well-defines ways. During just-in-time compilation, verification process examines metadata and verifies that the program is type safe before the source code is compiled into native machine code. [15]

Type safety ensures larger set of program code validity as it won't even compile when type safety is violated, unless otherwise specified. Compiler errors mean less possibility for run time errors, which are very prone to being hard to debug and fix.

DIT guarantees type safety in binders (see 5.2.1) and instance provider because complex DI container itself requires reflection and violation of type safety. By ensuring type safety on inputs entering the non-type safe part of the system, DIT shields the user from most of the possible run time errors.

4.3 Initialization Order

Initialization order of objects is very complex problem to solve if one wants to preserve conventional Unity scripting rules and reach maximum error safety.

Unity event function Awake is always run when a component is added to a Gameobject. It is impossible to manipulate the attached component before Awake is run, unless the Gameobject it is attached to is in inactive state. Other more complex way to achieve this would be to manually recreate AddComponent method and attach extra operations to the process by reverse engineering the Component system, but this seemed bit too excessive for such seemingly simple problem.

Executing scripts in certain order seemed to be the best way to solve this problem, but because Gameobjects in hierarchy do not have reliable execution order it had to be manually implemented. Unity provides script execution order interface to control the execution order of scripts. This interface is very poor way of ensuring initialization order as it is based on manual entries of concrete classes. One way would have been to inspect the whole assembly each time code is compiled and set script execution error for all new classes after default time.

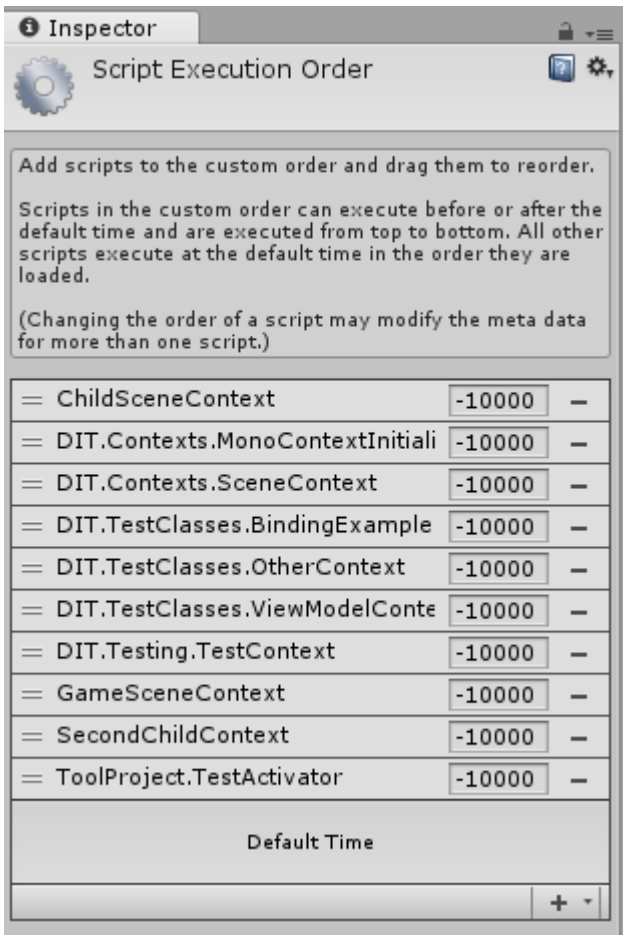


Image 4. Script execution order in Unity editor. If Unity script is not on this list, it is executed at random time during “Default Time” seen in the picture.

However, this solution would be error prone considering imported Dynamic-link libraries (DLL) and other unknown factors as the system would have to find all classes used in the project.

Therefore, in order to solve the initialization order problem, some system had to take control of the whole initialization process and stop all dependent objects from executing Awake method before all dependencies had been fulfilled. DIT achieves this by implementing ScriptOrderAttribute and ScriptOrderManager.

By adding ScriptOrderAttribute on top of context base classes, all deriving classes are automatically placed to the script execution order list by the ScriptOrderManager.

Once it had been guaranteed that contexts would be executed first, they had to take control of the initialization process. Contexts achieve this by executing complex initialization process which seeks all contexts in the scene. After all Gameobjects with contexts

have been found, the first context calls for all initialization methods of all other contexts which properly initialize their own hierarchy.

By inactivating the top most Gameobject, children of such Gameobject are also put in inactive state but are not by themselves marked as inactive. All children stay in inactive state until their parent is activated.

Problem arises when dependencies are injected, as injector attempts to activate the Gameobject the script is attached to. DIT handles this by not allowing the top most Gameobject to activate until the initialization is fully finished. After these steps are completed, DIT now has control of the initialization process and can guarantee that all dependencies are fulfilled before Awake is run.

Next chapter will introduce features of DIT in greater detail.

5 Library

DIT aims to provide developers with extensible development tools without enforcing or excluding other options. Originally DIT was designed to provide DI container with type safe interface, but over the development process more advanced features were developed. In this chapter all DIT features are explained in greater detail.

5.1 Core Features

After prototyping the concept of dependency injection and simple implementation of DI container, standard feature design had to be created. During the development process of DIT, great number of different features were inspected, but many of the them were discarded during the design process. After extensive research of programming forums, reference libraries and relevant literature, the following features remained to be the core of DIT:

- **Binding** is the registration process which maps dependencies to concrete implementations. Example of such can be seen in all RegisterBinding methods of previous examples. Requirement was to have a type safe interface for binding dependencies, which would guarantee that declared bindings would exist if the code would compile.

- **Configurations** for the bindings. Binding can map dependencies to concrete implementations but more constraints and additional features are required to create more complex outcomes. Therefore, configurations such as singletons and named injections were required.
- **Injection** to fulfill the basic principle of dependency injection. Injection had to work in a way that the dependent class had to know as little as possible about the injector, without compromising resource efficiency. All instance creation is primarily delegated to DIT.
- **Hierarchical Contexts** to solve the problem of complex multipart entities and their creation. Contexts themselves encapsulate registered dependencies and apply them only to objects created in the context. Contexts utilizing hierarchy (see 3.1.1) allow new patterns to emerge on entity level of abstraction.

These four principles form the core of DIT. In next chapters, all core features are explained in great detail.

5.1.1 Binding

In context of DIT, binding is the act of forming a pact or mapping some type to other type. In order for binding to be valid and compilable, the bound and target type have to abide LSP. The compiler error however, can be avoided by using dynamic binding interface offered by different binder classes if so desired.

```
public interface IBinderBase
{
      void SetContainer(IContext writer);
}
```

Figure 12. Base interface for all binders. SetContainer method is called when binder is accessed and ensures the binder is writing bindings to the correct context.

```
public interface IBinder : IBinderBase
{
      IRegularBindBuilder<TBindType> Bind<TBindType>();

      IBindBuilder<TBindType>
                  Bind<TBindType>(Expression<Func<TBindType>> factory)
                  where TBindType : class;

      IRegularBindBuilder Bind(Type type);
}
```

Figure 13. IBinder implements type safe and dynamic Bind methods for binding. Overloaded Bind<TBindType> method allows the use of user determinated instance creation with additional configurations.

Usage of the type safe interface is highly encouraged, unless building automated abstractions which are otherwise type safe.

Binders use simple trick to assure type safety which is based on generic interface/classes and “where” constraint in C# programming language.

```
public interface IRegularBindBuilder<TBindType> : IBindBuilder<TBindType>
{
      IBindBuilder<TBindType> To<TConcrete>()
            where TConcrete : TBindType;

      IBindBuilder<TBindType>
            To<TConcrete>(Expression<Func<TConcrete>> factory)
            where TConcrete : TBindType;

      IBindBuilder<TBindType> ToValue<TConcrete>(TConcrete value)
            where TConcrete : TBindType;

      IRegularBindBuilder<TBindType> And<TAdditionalType>()
            where TAdditionalType : TBindType;
}
```

Figure 14. IRegularBindBuilder<TBindType> is generic interface which can retain a static type for further usage.

In image 5 a binding sequence of type IFirstDependency has been created. All methods in IRegularBindBuilder<TBindType> interface declare where generic type constraint. In generic type definition, where clause is used to specify constraints of generic type arguments [12]. In pseudo program code, To and ToValue’s where constraint states the following

```
where generic type argument TConcrete is of type TBindType
```

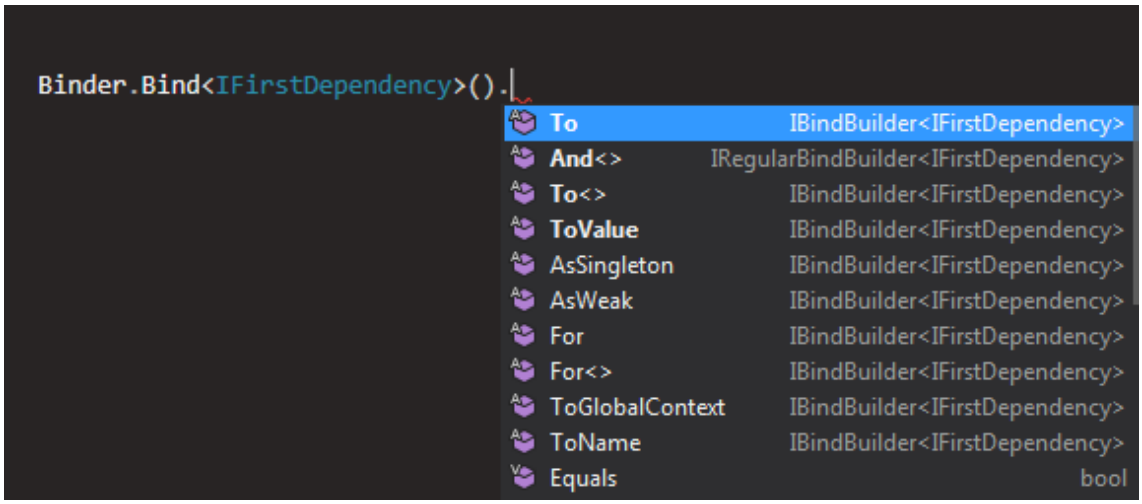


Image 5. All the methods on the right are suggestions. Suggested methods are methods which can be called, resulting in valid compilable code.

In simple words, the generic type argument of IRegularBindBuilder (TBindType) has to be assignable from the generic type argument given to or ToValue's method call (TConcrete), otherwise the code wont compile.

- **To<TConcrete>()** method declares that TBindType will be bound to type TConcrete.
- **To<TConcrete>(Expression<Func<TConcrete>> factory)** method declares that TBindType is bound to type TConcrete, and instances are created using the given expression of type Func<TConcrete>.
- **ToValue<TConcrete>(TConcrete value)** method declares that TBindType will be bound to the value parameter of type TConcrete as singleton.
- **And<TAdditonalType>()** method is shortcut for creating multiple bindings to the same concrete type.

Before explaining what bind builder interfaces are and how bindings are described in the system, few simple examples are required.

```
public interface IDerivingDependency : IFirstDependency { }
public class InterfaceImplementor : IFirstDependency, IDerivingDependency { }
public class SelfImplementing { }
public class OneLiner { }

public class BindingExamples : SceneContext
{
    protected override void RegisterBindings()
    {
           //Both interfaces are bound to InterfaceImplementor
           Binder.Bind<IFirstDependency>()
                 .And<IDerivingDependency>()
                 .To<InterfaceImplementor>();

           //Binds SelfImplementing to itself
           Binder.Bind<SelfImplementing>().To<SelfImplementing>();

           //Binds OneLiner to itself
           Binder.Bind<OneLiner>();

           //Binds IFirstDependency to InterfaceImplementor
           Binder.Bind(typeof(IFirstDependency))
                 .To(typeof(InterfaceImplementor))
    }
}
```

Figure 15. There are multiple different ways of declaring bindings in DIT.

In DIT, contexts have container of bindings which they manipulate using binders. Binders create bind builders which construct the rule set of bindings, which are then added to the contexts bind container.

Bind builders are narrow classes which describe the constraints and options available to a binding. Most important job of bind builder is to centralize the potentially complex construction logic of bindings and to ensure type safety in the binding process.

When a binding is properly created, it is added to contexts container of bindings as unresolved. Unresolved bindings are resolved if the user starts a new binding sequence or requests for new instance of any type. As some part of the program tries to find a binding using BindIntepreter, hash is created to represent the constraints and the hash is looked up in the associated contexts bind container. If no such hash is found in the container, hashes with less constraints are looked up until one is found. If no hash is found, DIT will throw an exception unless otherwise specified.

```
throw new BindingNotImplementedException("Binding "
      + interfaceType + (forType == null ? "" : " for class "
      + forType.ToString()) + (name == null ? "" : " with name "
      + name.ToString()) + " was not found. Did you forget to bind it?");
```

Figure 16. Error thrown in case binding was not found.

Next chapter will explain how binding constraints and configurations work in DIT.

5.1.2 Configurations

Configurations and constraints are used in DIT to create more complex requirements for bindings. Configuration options provide optional tools for the developer to map dependencies which are of the same type but require more specific rules to avoid duplicate bindings.

```
public interface IBindBuilder<TBindType>
{
      IBindBuilder<TBindType> ToGlobalContext();

      IBindBuilder<TBindType> AsSingleton();

      IBindBuilder<TBindType> ToName(object name);

      IBindBuilder<TBindType> For<TThisType>();

      IBindBuilder<TBindType> For(Type type);

      IBindBuilder<TBindType> AsWeak();
}
```

Figure 17. Bind builder configuration interface.

IBindBuilder<TBindType> implements multiple additional configuration options for bindings.

- **ToGlobalContext()** adds the built binding to global contexts bind container (see 5.1.4) instead of the binder's container.
- **AsSingleton()** declares that the binding will provide concrete instance(s) as singleton (see 3.4.2).
- **ToName(object name)** adds a name constraint to the binding (see 2.2).
- **For<TThisType>()** adds a type constraint for the requesting entity. Only types matching TThisType are applicable for the binding.
- **For(Type type)** dynamic version of For<TThisType>()
- **AsWeak()** instructs that this binding will be ignored if binding conflicts arise. If a bind container already has a binding with matching hash or new binding is added with matching hash, the binding marked as weak will be discarded.

For constraint is powerful way to encapsulate bindings for automated systems without cluttering the pool of bindings in context. This allows the developer to declare multiple bindings of the same type without using named injections.

```
public interface IFirstDependency {}
public class FirstDependency : IFirstDependency {}
public class DerivingDependency : FirstDependency {}

public class ForContext : SceneContext
{
      protected override void RegisterBindings()
      {
            Binder.Bind<ForClass>();
            Binder.Bind<OtherClass>();

            Binder.Bind<IFirstDependency>().To<FirstDependency>();
            Binder.Bind<IFirstDependency>()
                  .To<DerivingDependency>()
                  .For<ForClass>();
         }

      protected override void OnStart()
      {
            var forClass = InstanceProvider.GetInstance<ForClass>();
            var otherClass = InstanceProvider.GetInstance<OtherClass>();
      }
}

public class ForClass
{
      public ForClass()
      {
            Debug.Log(First.GetType());
      }

      [Inject]
      private IFirstDependency First;
}
```

```
public class OtherClass
{
      public OtherClass()
      {
            Debug.Log(First.GetType());
      }

      [Inject]
      private IFirstDependency First;
}
```

Figure 18. Otherwise conflicting dependencies are mapped with For constraint.

The code would result in following messages to appear on the console:

```
Debug.Log: DIT.TestClasses.DerivingDependency
Debug.Log: DIT.TestClasses.FirstDependency
```

Names are preferably used in cases where one class has multiple identical interfaces as dependencies (see 2.2), while the use of For is encouraged when different classes in the same context require a different implementation for identical interfaces.

Binding a dependency as singleton is desired when only one instance of the implementation is required across the context. Contextual local singletons solve the problem of distributing one reference across multiple scripts in hierarchy.

Weak bindings are useful in cases where one default binding is required or state of other bindings are not known or in control of the developer. When using dynamic binding without type safety, weak bindings provide a safe interface for binding without conflicts. This itself enables the possibility of creating virtual dependencies which can be overridden.

```
public interface IFirstDependency {}
public class FirstDependency : IFirstDependency {}
public class DerivingDependency : FirstDependency {}
public class MoreDerivingDependency : DerivingDependency {}

public class WeakContext : SceneContext
{
      protected override void RegisterBindings()
      {
            Binder.Bind<DefaultEntity>();

            Binder.Bind<IFirstDependency>().To<FirstDependency>().AsWeak();
            Binder.Bind<IFirstDependency>().To<MoreDerivingDependency>();
            Binder.Bind<IFirstDependency>().To<DerivingDependency>().AsWeak();
      }

      protected override void OnStart()
      {
            var entity = InstanceProvider.GetInstance<DefaultEntity>();
      }
}
```

```
public class DefaultEntity
{
      public DefaultEntity()
      {
            Debug.Log(First.GetType());
      }

      [Inject]
      private IFirstDependency First;
}
```

Figure 19. Bindings to FirstDependency and DerivingDependency are marked as weak.

When binding is declared as weak, it is always discarded in case of better alternatives.

```
for (int i = 0; i < hashes.Count; i++)
{
      if (bind.IsWeak)
      {
            if (container.Bindings.ContainsKey(hashes[i]))
            {
                  if (container.Bindings[hashes[i]].IsWeak)
                        container.Bindings.Remove(hashes[i]);
                  else
                        continue;
            }
      }
      container.Bindings.Add(hashes[i], bind);
}
```

Figure 20. Logic for resolving weak bindings

Next chapter will introduce injection process in DIT.

5.1.3 Injection

Injection in DIT works very much like in any other DI container by employing reflection to inspect and insert values during run time.

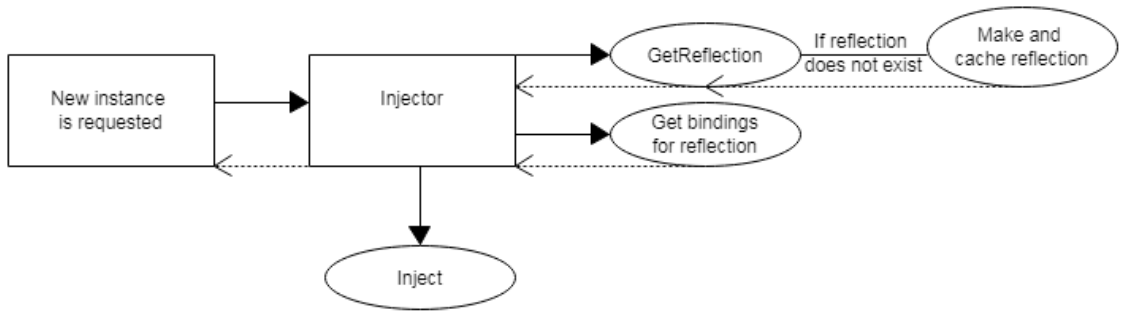


Image 6. Simplified diagram of injection process.

Because reflection is relatively expensive operation (see 3.3), few conventions can be followed to reduce performance overhead it causes.

- Caching the required type information and accessing the information only through cache
- Using reflection only during initialization of the program

By caching all type information, expensive reflection has to be done only once for each type. By delegating all reflection operations and caching to initialization stage of the program, all performance heavy operations are dealt with during start of the program which is more tolerable and commonly accepted than run time performance issues.

```
public class ReflectionCache: IReflectioncache
{

      private readonly Dictionary<Type, IReflectedClass>
                     Reflections = new Dictionary<Type, IReflectedClass>();

      private void ReflectClass(Type type)
      {
            var reflectedClass = new ReflectedClass(type);
            Reflections.Add(reflectedClass.Type, reflectedClass);

      }

      public IReflectedClass GetReflection(Type type, IContext context)
      {
            if (!Reflections.ContainsKey(type))
                  ReflectClass(type);
            var reflection = Reflections[type];
            reflection.ResolveFor(context);
            return reflection;
      }
}
```

Figure 21. Reflection cache validates if cached reflection for type exists and creates one if it does not. ResolveFor method resolves all special attributes for the calling context. AutoInject is one such attribute.

IReflectedClass interface provides access to cached reflections while the underlying implementation holds all the relevant data gained from reflecting the associated type.

DIT makes use of Injectable class to describe fields and properties of reflected types. The function of Injectable is to cache relevant type information, attributes, and setter and getter delegates.

DIT has one main attribute type associated with injection which can be seen in most of the examples. InjectAttribute class has metadata attribute AttributeUsage which restricts

the usage of the attribute to fields and properties. InjectAttribute can be used to mark dependent fields and properties which the class wants to be fulfilled.

In addition to InjectAttribute, more deriving type AutoInjectAttribute exists to automatically solve dependencies. This attribute allows the developer to solve concrete dependencies without explicitly declaring bindings for them, as well as interfaces which are marked with ConstructedAsAttribute.

ConstructedAsAttribute is used to mark classes or interfaces with default type they are constructed as when binding is not explicitly declared.

```
[ConstructedAs(typeof(FirstDependency))]
public interface IFirstDependency {}
public class FirstDependency : IFirstDependency{}
public class DerivingDependency : FirstDependency {}

public class NoBindingsContext : SceneContext
{
      protected override void RegisterBindings()
      {
            Binder.Bind<NoBindings>();
      }

      protected override void OnStart()
      {
            var entity = InstanceProvider.GetInstance<NoBindings>();
      }
}

public class NoBindings
{
      public NoBindings()
      {
            Debug.Log(First.GetType());
            Debug.Log(Deriving.GetType());
      }

      [AutoInject]
      private IFirstDependency First;

      [AutoInject]
      private DerivingDependency Deriving;
}
```

Figure 22. No bindings regarding IFirstDependency or DerivingDependency were registered.

Regardless of lacking binding registrations, this code would result in following messages to appear on the console:

```
Debug.Log: DIT.TestClasses.FirstDependency
Debug.Log: DIT.TestClasses.DerivingDependency
```

IInjector provides interface for injecting dependencies. Whenever injector receives a new instance to be injected, it will attempt to inject it recursively. By default, injector iterates through hierarchy of the calling context until it finds a suitable binding for the requested dependency or no binding is found. Multi-context injection allows injection based on collection of arbitrary contexts.

```
public interface IInjector
{

      object MultiContextInject(
            List<IContext> contexts,
            IConstructedObject constructedObject,
            bool partlyInject = false);

      object Inject(
            IContext context,
            IConstructedObject constructedObject,
            bool partlyInject = false);
}
```

Figure 23. Optional parameter partlyInject allows dependencies to be left uninjected if no binding is found.

Next chapter will explain what contexts are and how hierarchical contexts work in DIT.

5.1.4 Hierarchical Contexts

Contexts are the configuration containers in DIT, which are required for dependencies to be injected. Contexts handle initialization logic of the program and are designed to work in Unity scenes. They encapsulate and provide services and/or interfaces for binding registration, object lifecycle, injection, instance creation and reflection. Contexts work as centralized service hubs to ensure that classes used in production code are not required to inherit from any custom class. This allows non DIT specific code to be written when employing the framework, excluding inject attributes. This has advantages, as it eases transition if the user decides to stop using DIT in middle of the development.

To understand how contexts can be utilized in Unity development, few simple test cases are needed. Image 7 shows the premise for this example and Figure 24 lists all relevant scripts used in the example.

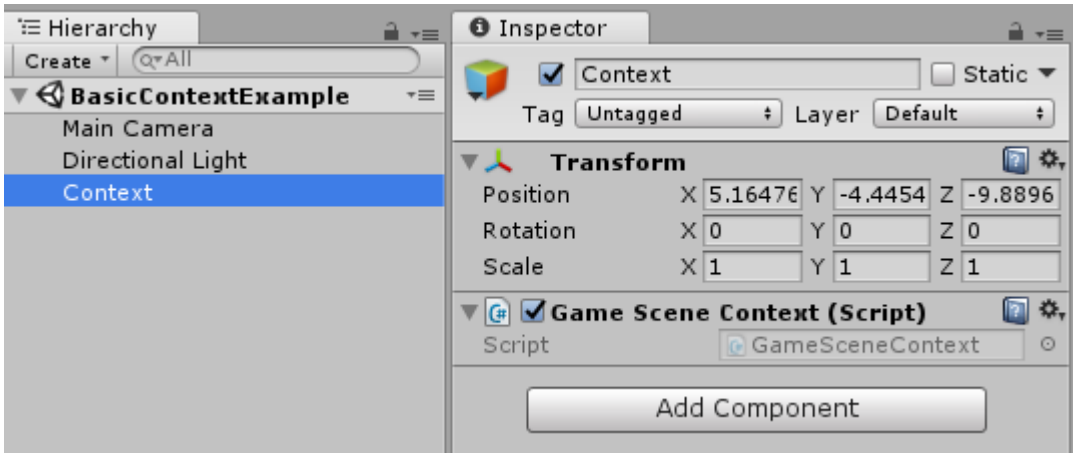


Image 7. Simple test premise in Unity hierarchy.

```
public class Hero : MonoBehaviour
{
    [Inject]
    private IWeapon Weapon { get; set; }

    [Inject]
    private IArmour Armour { get; set; }


    public void Awake()
    {
        Weapon.Use();
        Armour.Inspect();
    }
}

public class GameSceneContext : SceneContext
{
    protected override void RegisterBindings()
    {
        MonoBinder.Bind<Hero>().To<Hero>("Hero");

        Binder.Bind<IArmour>().To<Breastplate>();
        Binder.Bind<IWeapon>().To<Sword>();

        Debug.Log("GameSceneContext register");
    }

    protected override void OnAwake()
    {
        Debug.Log("GameContext OnAwake");
    }

    protected override void OnStart()
    {
        var hero = InstanceProvider.GetInstance<Hero>();
        Debug.Log("GameContext OnStart");
    }

}
```

Figure 24. Hero is Monobehaviour, therefore it requires Gameobject to exist. String "Hero" is given as parameter to define the name of the created Gameobject. MonoBinder is binder interface for MonoBehaviours.

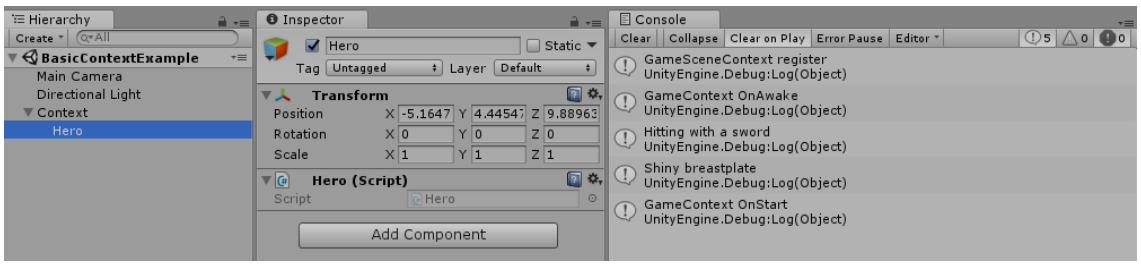


Image 8. Outcome of the test when entering play mode. Runtime instantiated MonoBehaviour objects are placed below the context which created them by default.

Because Gameobjects can be directly placed into scenes and hierarchy, DIT has been equipped to handle such situation.

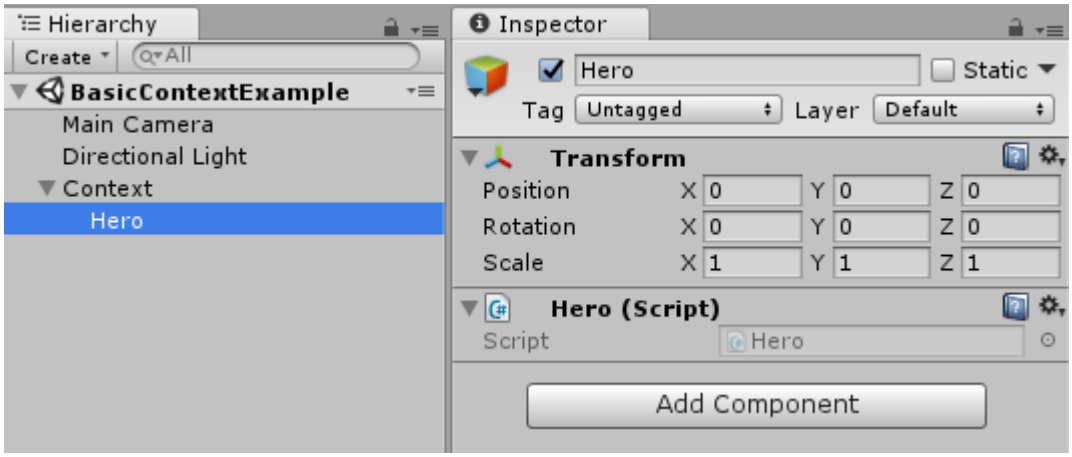


Image 9. Hero is placed into the scene in editor and not created through InstanceProvider.

Small modification is made to the context to demonstrate the difference:

```
public class GameSceneContext : SceneContext
{
    protected override void RegisterBindings()
    {
        Binder.Bind<IArmour>().To<Breastplate>();
        Binder.Bind<IWeapon>().To<Sword>();

        Debug.Log("GameSceneContext register");
    }

    protected override void OnAwake()
    {
        Debug.Log("GameContext OnAwake");
    }

    protected override void OnStart()
    {
        Debug.Log("GameContext OnStart");
    }

}
```

Figure 25. Creation of hero was removed from the context as the Gameobject with Hero script has already been placed into the scene.

When entering play mode, exactly the same outcome is reached as in Image 8.

Context and injector are built to understand how hierarchy works in Unity. This can be utilized by placing arbitrary number of contexts below one Gameobject. As explained in chapter 5.1.3, injector iterates through contexts before matching binding is found for the dependency. This allows the creation of entities with virtual dependencies and shared resources such as data models using the singleton pattern (see 3.4.1).

After understanding principles of contexts, DI, hierarchy and Gameobjects, more complex example which utilizes these capabilities can be demonstrated.

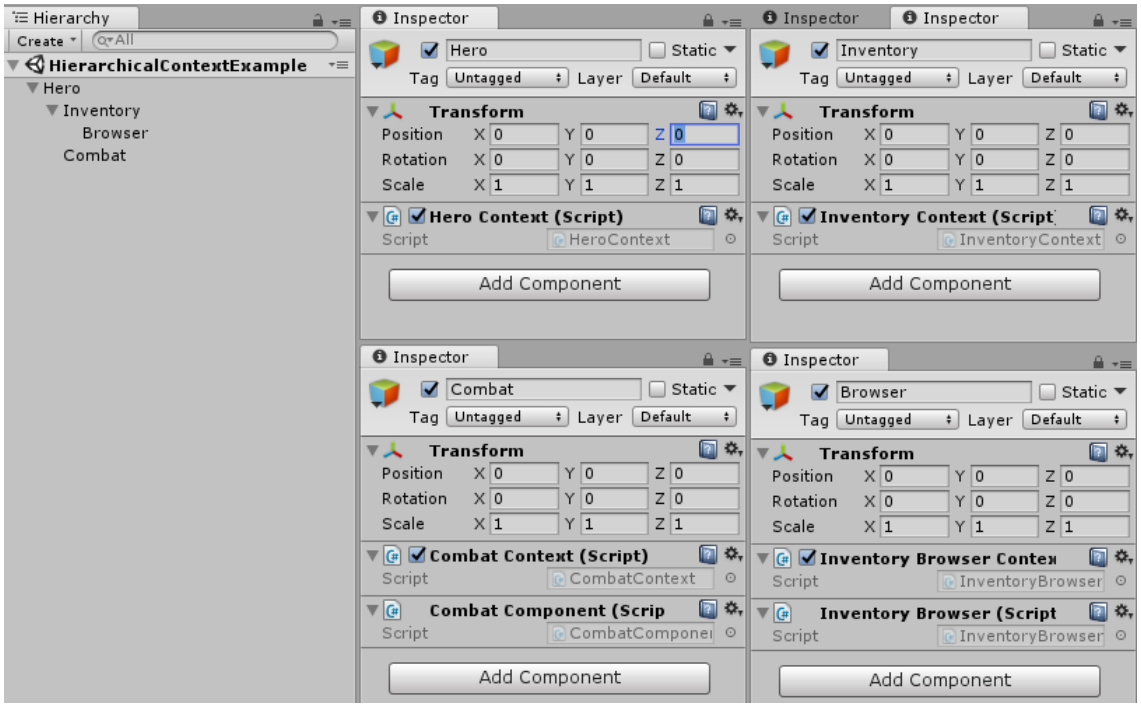


Image 10. Premise of the example. All Gameobjects seen in the hierarchy and their components are seen on the right.

Because DIT implements hierarchical dependency injection, dependencies can be declared higher in the hierarchy and still be properly injected to child Gameobjects components. This allows child contexts to be dependent on other contexts which adds modularity to the program.

```
public interface IItemModel
{
    string Name { get; }

    float Strength { get; }
}

public class AmuletModel : IItemModel
{
    public string Name
    {
        get { return "Amulet"; }
    }

    public float Strength
    {
        get { return 36.2f; }
    }
}
public class TrinketModel : IItemModel
{
    public string Name
    {
        get { return "Trinket"; }
    }

    public float Strength
    {
        get { return 23.8f; }
    }

}
```

Figure 26. Simple models for simulating items with few properties.

Items are mock objects used in this example to simulate how data models can contain information and DIT allows easy sharing of such data models through hierarchy and singleton pattern. This allows subtyping, meta level class structures and component system to be formed.

```
public interface IEquipmentViewModel
{
    float Strength { get; }
}

public interface IEquipmentModel : IEquipmentViewModel
{
    void Equip(IItemModel itemModel);
    void Unequip(IItemModel itemModel);
}

public class EquipmentModel : IEquipmentModel
{
    private List<IItemModel> EquippedItems = new List<IItemModel>();
    public void Equip(IItemModel itemModel)
    {
        EquippedItems.Add(itemModel);
        Debug.Log("Equipped " + itemModel.Name);
    }
```

```
    public void Unequip(IItemModel itemModel)
    {
        EquippedItems.Remove(itemModel);
        Debug.Log("Unequipped " + itemModel.Name);
    }

    public float Strength
    {
        get
        {
            float retval = 0;
            for (int i = 0; i < EquippedItems.Count; ++i)
                retval += EquippedItems[i].Strength;
            return retval;
        }
    }
}
```

Figure 27. Classes concerning equipment describe all equipped items and how they collectively affect combat stats.

EquipmentModel class can return the collective strength of all items currently equipped which can be used to calculate damage. The model contains information about equipped items which can be manipulated by other parts of the program.

```
public interface IInventoryViewModel
{
    List<IItemModel> Items { get; }
}

public interface IInventoryModel : IInventoryViewModel
{
    void Add(IItemModel itemModel);
    void Remove(IItemModel itemModel);
}

public class InventoryModel : IInventoryModel
{
    private List<IItemModel> ItemsField = new List<IItemModel>();
    public List<IItemModel> Items
    {
        get { return ItemsField; }
    }

    public void Add(IItemModel itemModel)
    {
        ItemsField.Add(itemModel);
    }

    public void Remove(IItemModel itemModel)
    {
        ItemsField.Remove(itemModel);
    }
}

public class InventoryBrowser : MonoBehaviour
{
    [Inject]
    private IInventoryViewModel InventoryModel;


    public void ListAllItems()
```

```
    {
        Debug.Log("Inventory contains the following items: ");
        for (int i = 0; i < InventoryModel.Items.Count; ++i)
            Debug.Log(InventoryModel.Items[i].Name);
        Debug.Log("----------End of inventory----------");
    }
}

public class InventoryBrowserContext : SceneContext
{
    protected override void RegisterDependentBindings()
    {
        Binder.Bind<IInventoryViewModel>()
           .ToValue(InstanceProvider.GetInstance<IInventoryModel>());
    }

    protected override void OnStart()
    {
        var browser = GetComponent<InventoryBrowser>();
        browser.ListAllItems();
    }
}

public class InventoryContext : SceneContext
{
    protected override void RegisterBindings()
    {
        Binder.Bind<IInventoryModel>().To<InventoryModel>().AsSingleton();
    }

    protected override void OnAwake()
    {
        var equipmentModel = InstanceProvider.GetInstance<IEquipmentModel>();
        var inventoryModel = InstanceProvider.GetInstance<IInventoryModel>();

        inventoryModel.Add(new AmuletModel());
        inventoryModel.Add(new TrinketModel());

        equipmentModel.Equip(inventoryModel.Items[0]);
        equipmentModel.Equip(inventoryModel.Items[1]);
    }

}
```

Figure 28. Inventory browser context is dependent on IInventoryModel and wishes to bind the shared data model to read only interface. OnAwake method is run after all dependencies have been injected.

Inventory contains collections of items and enables equipping and unequipping of items. RegisterDependentBindings method is always run after all RegisterBinding methods are properly executed in the hierarchy.

```
public class HeroContext : SceneContext
{
    protected override void RegisterBindings()
    {
        Binder.Bind<IEquipmentModel>().To<EquipmentModel>().AsSingleton();
    }
}

public class CombatContext : SceneContext
{
    protected override void RegisterDependentBindings()
```

```
    {
        Binder.Bind<IEquipmentViewModel>()
           .ToValue(InstanceProvider.GetInstance<IEquipmentModel>());
    }

    protected override void OnStart()
    {
        var combatComponent = GetComponent<CombatComponent>();
        combatComponent.Swing();
    }
}
```

Figure 29. CombatContext is dependent on IEquipmentModel and wishes to bind it to read only interface called IEquipmentViewModel.

Read only interface for data models does not add any functionality to the program itself but it restricts other parts of the program from modifying the shared resource. This is useful pattern if the shared data model is not supposed to be modified in that sub-context.

Contexts can be extended to create automated generic modules and complete entities without the need of boilerplate code by extending on the pattern seen in Image 10. Independent sub-contexts can be combined with any parent context which fulfills the required dependencies, making them highly reusable.

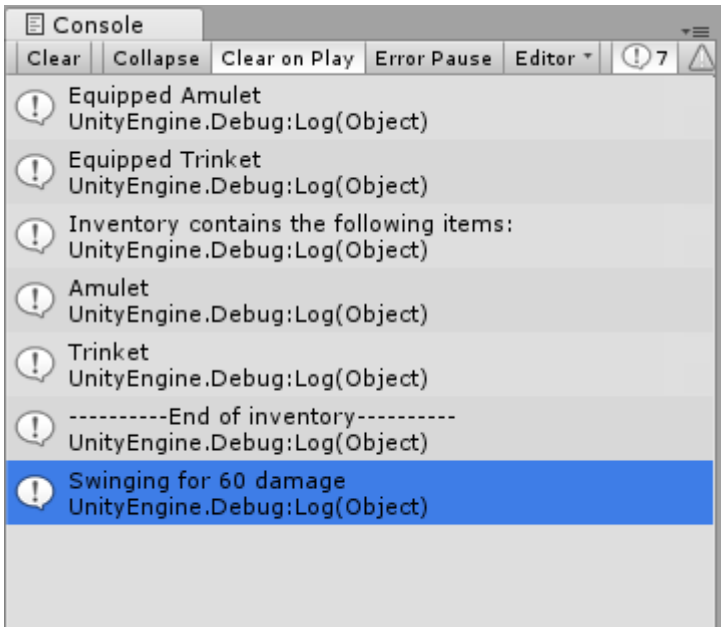


Image 11. Outcome when entering play mode.

Next chapter will explain how DIT implements object pooling.

5.2 Pooling

Pooling or object pooling is a design pattern used for object recycling. Object pooling can offer significant performance boosts to parts of the program where class instantiation rate is high and the amount of concurrently used instances is low. [3]

Example of such case where object pooling is needed is shooting projectiles. A character which can shoot 20 projectiles each second could cause huge performance issues if new projectile would have to be created each time a projectile would be needed. Object pool would create many projectiles during the initialization process of the program and provide an interface for the object dependent on projectiles to use when new projectiles are needed. These projectiles would then be returned back to the object pool when their lifecycle would end, allowing them to be recycled

DIT provides default implementation and binder interface for object pooling with type safety.

```
public class Pool<T> : PoolBase, IPool<T>
{
    [Inject]
    private IInstanceProvider InstanceProvider { get; set; }

    [Inject]
    private IPoolConfiguration PoolConfiguration { get; set; }

    private readonly Stack<T> InstanceStack = new Stack<T>();

    public Type PoolType { get; protected set; }

    public int PoolSize
    {
        get { return PoolConfiguration.PoolSize; }
        set { PoolConfiguration.PoolSize = value; }
    }

    public bool AutoInflate
    {
        get { return PoolConfiguration.AutoInflate; }
        set { PoolConfiguration.AutoInflate = value; }
    }

    public float InflationMultiplier
    {
        get { return PoolConfiguration.InflationMultiplier; }
        set { PoolConfiguration.InflationMultiplier = value; }
    }


    public Pool()
    {
        PoolType = typeof(T);
        CreateInstances(PoolSize);
    }
```

```
    public T GetInstance()
    {
        T retval = default(T);

        // If null for default(T)
        while (EqualityComparer<T>.Default.Equals(retval, default(T)))
        {
            if (InstanceStack.Count < 1)
                CreateInstances(1);

            retval = InstanceStack.Pop();
        }


        if (AutoInflate)
            Inflate();

        // During instance creation, it is made sure that T is IPoolable
        IPoolable poolable = retval as IPoolable;
        poolable.OnActivation();
        poolable.ReturnInstance = () =>
        { ReturnInstance(retval); };
        return retval;
    }

    public void ReturnInstance(T instance)
    {
        // During instance creation, it is made sure that T is IPoolable
        IPoolable poolable = instance as IPoolable;
        poolable.OnInactivation();
        poolable.ReturnInstance = null;
        InstanceStack.Push(instance);
    }

    private void Inflate()
    {
        int activeInstances = PoolSize - InstanceStack.Count;
        int instanceAmount = Mathf.RoundToInt(
                    activeInstances * InflationMultiplier) - PoolSize;
        CreateInstances(instanceAmount);
    }


    private void CreateInstances(int amount)
    {
        for (int i = 0; i < amount; i++)
        {
            T val = InstanceProvider.GetInstance<T>(this.GetType());
            if (!val.IsThis<IPoolable>())
                throw new NotPoolableException("Concrete type "
                     + val.GetType() + " does not implement "
                     + typeof(IPoolable));

            IPoolable instance = val as IPoolable;
            instance.OnInactivation();
            InstanceStack.Push(val);
            PoolSize++;
        }
    }
}
```

Figure 30. Default implementation for pooling. Objects placed in the pool have to implement IPoolable interface.

Pool makes sure that there's always fresh instances available for the requester. This is ensured by auto inflation which is defined during binding registration. If, however auto inflation is not active, pool will always create one new instance and returns it to the requester.

```
public interface IPoolable
{
    void OnActivation();

    void OnInactivation();

    ReturnInstance ReturnInstance { get; set; }
}
```

Figure 31. IPoolable interface required by the pool.

IPoolable requires the implementing class to specify how the object is handled when it is activated or deactivated and allows pooled objects to return themselves back to the pool. This is ensured during GetInstance method, where the pool creates a delegate for the IPoolable and stores it to the ReturnInstance variable which can be called to recycle the object.

Next chapter will introduce built in model–view–controller framework in DIT.

5.3 Model–view–controller

Model–view–controller (MVC) is a software architecture type, standard in web applications and often used in enterprise software. It is used to separate data (model), user interface (view), and operations of data transform (controller). MVC provides modular approach to software development by separating business logic from the user interface and providing modularity and reusable code. [16]

DIT implements MVC by separating these three concerns into 3 different classes.

- **Model** represents all the stored data of particular object, entity or module, allowing it to be transformed by controllers.
- **View** represents everything which can be soon or interacted by the user. View listens for changes in its model and acts according to the changes.
- **Command** transforms model data based on its internal implementation. They cannot hold permanent state on their own. Commands can be regarded as glorified method wrappers which allows more customization.

Contexts are used as configuration and constructor class for models and views and they are responsible for adding commands to events which views trigger when requested by the system.

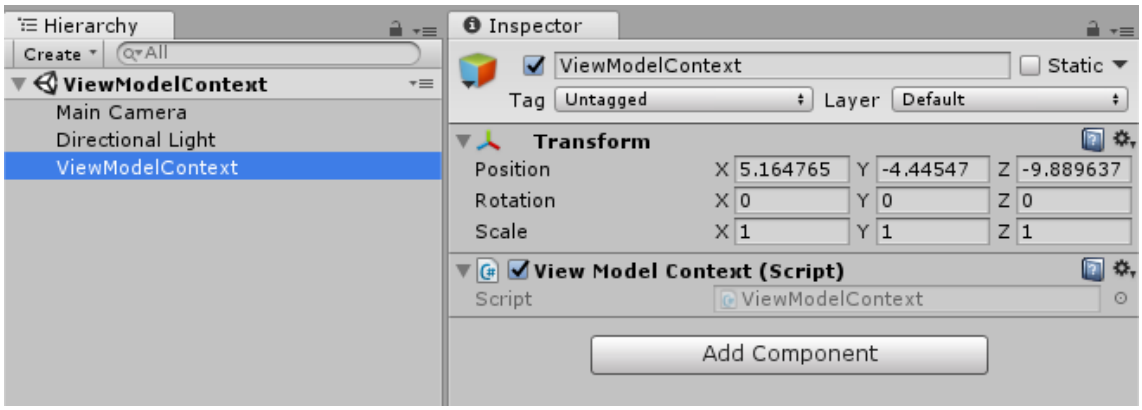


Image 12. Premise for the MVC test case.

```
public class DamageEvent : EventBase<CombatModel, int> { }

public class ViewModelContext : SceneContext<HealthView, CombatModel>
{

    protected override void ConfigureModel()
    {
        Model.Armor = 0.5f;
        Model.Health = 100;
    }

    protected override void ConfigureView()
    {
        View.DamageEvent.T1_T2_AddCommand<TakeDamage>();
    }

}

public class HealthView : View<CombatModel, DamageEvent>
{
    public DamageEvent DamageEvent
    {
        get { return FirstEvent; }
    }

    private void OnDamageTaken(int health)
    {
        Debug.Log("Health has been updated to " + health);
        if (health <= 0)
            Debug.Log("You died");
    }

    private void Start()
    {
        Model.ObservableHealth.AddListener(OnDamageTaken);
        Debug.Log("Someone hit you, executing DamageEvent");
        DamageEvent.Trigger(Model, 100);
        Debug.Log("Someone hit you again, executing DamageEvent");
        DamageEvent.Trigger(Model, 100);
    }
}
```

```
public class CombatModel : ObservableModel<int, float>
{
    public ObservableMember<int> ObservableHealth
    {
        get { return FirstMember; }
    }

    public int Health
    {
        get { return First; }
        set { First = value; }
    }

    public ObservableMember<float> ObservableArmor
    {
        get { return SecondMember; }
    }

    public float Armor
    {
        get { return Second; }
        set { Second = value; }
    }

}

public class TakeDamage : Command<CombatModel, int>
{

    public override void Execute(CombatModel model, int damage)
    {
        int totalDamage = (int)Mathf.Floor(damage * (1f - model.Armor));
        model.Health -= totalDamage;
    }
}
```

Figure 32. SceneContext can be instructed to create associated view and model which have auto wired dependencies.

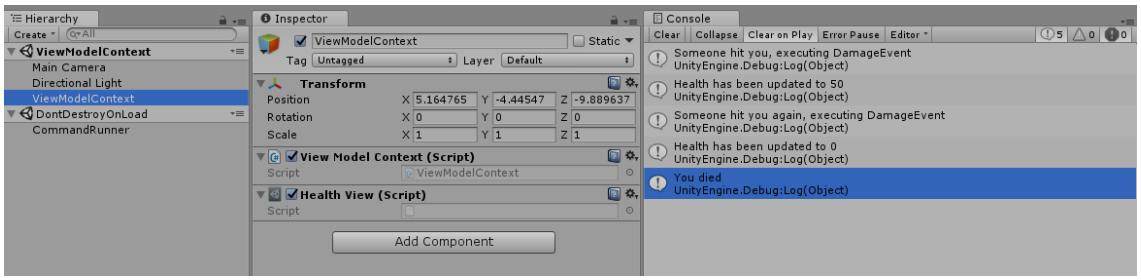


Image 13. The view is created during run time due to templated SceneContext. CommandRunner below DontDestroyOnLoad handles all coroutines invoked by commands.

Contexts have base class for creating view and model automatically with auto wired event and model dependencies. These dependencies are named based on their order and have to be renamed to clarify what each event, value or model does.

Models can contain values or other models. If models contain other models, these models are recursively bound and injected and all of these models can be accessed anywhere in the context by injection or InstanceProvider. Models have default implementation for observing value changes which are mainly used by the view to appropriately react to data transformations.

Events in DIT encapsulate view-view or user-view interaction with the possibility of adding cross cutting corners, such as analytics or logging without writing extra code.

Command can be ordered to listen to an event trigger, firing its Execute method in the process. Misplaced naming convention in ConfigureView method T1_T2_AddCommand<TakeDamage> enables commands with less parameters than the event to subscribe to it. The syntax is required because in C#, different where constraint is not counted as overloading a method. Events can have regular method subscribers in addition to commands. It is entirely possible to fire an event which causes another command in another context to fire. In this case, all dependencies injected to the command are looked up in both context hierarchies.

Because all business logic and data transformation are restricted (by the architecture convention) to commands, they can be used to implement a job system for multi-threading and many other useful patterns such as undo operation, automatic logging and analytics. This also makes testing very easy as everything can be tested in isolation.

MVC and dependency injection system provided by DIT tightly controls all architectural aspects of the program and restricts them to narrow sets of interfaces, therefore making it quite easy to add additional program wide features without changing production code.

The next chapter will summarize everything presented in this report.

6 Summary

DIT enables the use of dependency injection pattern in Unity engine by implementing contextual and hierarchical dependency relationships. This allows the user to create modular and highly reusable components which can work dependently or independently from other components used in the framework. Similar or identical entities can exist in the system and function completely differently based on the context(s) associated with

it. This helps with game development as one can substitute only configuration part of the program without modifying production code and still retain a working program. This also makes testing easier as non-related dependencies can be easily substituted with mock objects to simulate the system.

Model–view–controller system in DIT provides a common architectural interface for the user which makes component and entity design easier to approach. Because this architecture model generalizes all actions and simplifies the design process, automation can be built on top of the framework much more effortlessly. Such automation can be built in form of polymorphism and generic type arguments, using Unity engine's prefab system, Unity engine's asset bundle system, or custom form of representation which I am currently working on.

The ultimate purpose of DIT is to be general game development framework with useful tools and patterns, which allow the user to pick what they find useful and/or are comfortable with. The planned and implemented features reflect my current personal presences gained by exploring great amount of software architectural space and mentally building them to resolve complex problems. Biggest requirements set for DIT were that it should be modular, make architecture and entity design more effortless, allow high levels of automation, be extendable and it should have low performance overhead.

There are many small problems with the current architecture of DIT which makes it impossible to add new low-level features to it without inflexible "work around" code. Data-oriented design, entity component system and contextual-oriented programming are something which I intend to research further to help me understand performance issues better, and to learn more optimal architectural solutions. Test-driven development is also a thing I want to address in the further development of this framework.

References

1 Unity Technologies. 2017. Unity Manual for console window. https://docs.unity3d.com/Manual/Console.html

2 Unity Technologies. API reference document for MonoBehaviour. https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

3 Source Making. Web page explaining different software design patterns. https://sourcemaking.com/design_patterns/

4 Martin Fowler. 2004. Inversion of Control Containers and the Dependency Injection pattern. https://martinfowler.com/articles/injection.html

5 Francois-Nicola Demers and Jacques Malenfant. 1995. Reflection in logic, functional and object-oriented programming: A Short Comparative Study.

6 Microsoft. 2017. Microsoft .NET documentation. Reflection in the .NET framework. https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection

7 Microsoft. Microsoft Developer Network .NET Framework Class Library. Attribute Class. https://msdn.microsoft.com/en-us/library/system.attribute(v=vs.110).aspx

8 Erich Ganma, Richard Helm, Ralph Johnson, John Vlissides. 1994. Design Patterns: Elements of Reusable Object-Oriented Software.

9 Microsoft. Microsoft Developer Network Object Class. Object.GetHashCode Method(). https://msdn.microsoft.com/en-us/library/system.object.gethashcode(v=vs.110).aspx

10 Richard Carr. 2012. Multiton Design Pattern. Programming blog. http://www.blackwasp.co.uk/multiton.aspx

11 OODesign. Liskov's Substitution Principle. Website explaining Object-oriented design. http://www.oodesign.com/liskov-s-substitution-principle.html

12 Microsoft. 2015. C# Language reference. Keywords. https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/where-generic-type-constraint

13 Microsoft. Microsoft Developer Network. .NET Framework Class Library. Activator Class. https://msdn.microsoft.com/en-us/library/system.activator(v=vs.110).aspx

14 Microsoft. Microsoft Developer Network. .NET Framework Class Library. FormatterServices.GetUnitializedObject Method (Type). https://msdn.microsoft.com/en-

us/library/system.runtime.serialization.formatterservices.getuninitializedobject(v=vs.110).aspx

15 Surya Sasidhar. 2015. Stackoverflow programming forum. What is type-safe in .net? https://stackoverflow.com/questions/2437469/what-is-type-safe-in-net

16 Abhishek Bajpai. 2009. ResearchGate. Model View Controller Architecture on Embedded Systems. https://www.researchgate.net/publication/273626360_Model_View_Controller_Architecture_on_Embedded_Systems