▲ Hochschule Harz

BACHELORARBEIT

BA AI 27/2015

INVERSION OF CONTROL IN UNITY3D

Erstprüfer:	Prof. Daniel Ackermann
Zweitprüfer:	Prof. Ph. D. Jürgen K. Singer
Abgabedatum:	13.10.2015

VORGELEGT VON:

Ludwig Lauer

Albert Schweitzer Str. 8

31061 Alfeld (Leine)

m18711

Berlin, der 12. Oktober 2015

Hochschule Harz
Fachbereich Automatisierung und Informatik

THEMA UND AUFGABENSTELLUNG DER BACHELORARBEIT

BA AI 27/2015

LUDWIG LAUER

INVERSION OF CONTROL IN UNITY3D

Unity3D hat sich in den letzten Jahren als Spieleengine etabliert. Immer größere und ambitioniertere Projekte werden mit der Engine umgesetzt. Da Unity3D als Rapid Prototyping Tool für unabhängige Spieleentwickler konzipiert wurde, fällt die parallele Entwicklung in mittleren bis großen Teams schwer. Eine der Ursachen dieses Problems ist die enge Kopplung von Abhängigkeiten zwischen Objekten. Die Folgen sind Starrheit und Zerbrechlichkeit bei Veränderung des Programmcodes.

In der objektorientierten Programmierung gilt Inversion Of Control als Maßnahme um lose Kopplung zu erzielen. In dieser Arbeit soll untersucht werden, ob sich das Paradigma auch in der komponentenbasierten Architektur von Unity3D anwenden lässt. Dafür soll das Framework StrangeIoC herangezogen und genauer betrachtet werden. Mit Features wie Automated Dependency Injection, eigenem Eventsystem und einer MVC(S) Architektur sollen die Qualität, Testbarkeit, Skalierbarkeit und Wartbarkeit der Anwendung erhöht werden.

Ziel der Bachelorarbeit ist folglich das Erstellen einer lose gekoppelten Beispielanwendung in Unity3D unter Verwendung von StrangeIoC. Zudem soll untersucht werden, welche Praktiken die Ursache von enger Kopplung in Unity3D sind, welche Folgen diese haben und wie diese mit Inversion Of Control, Dependency Injection und StrangeIoC im speziellen, vermeidbar sind.

Die Bachelorarbeit beinhaltet folgende Teilaufgaben:

- Analyse von Ursachen und Folgen enger Kopplung zwischen Objekten in Unity3D
- Auseinandersetzung mit den Paradigmen Inversion Of Control und Dependency Injection
- Entwicklung einer Beispielanwendung mit StrangeIoC mit dem Ziel lose Kopplung unter Objekten umzusetzen

- Kritische Betrachtung von Entwicklungsaufwand, Qualität, Testbarkeit, Skalierbarkeit und Wartbarkeit im Vergleich mit anderen Architekturansätzen

Prof. Daniel Ackermann	Prof. Ph. D. Jürgen K. Singer
1. Prüfer	2. Prüfer

Eidesstattliche Erklärung

Ich erkläre, dass ich meine Bachelor-Arbeit „Inversion Of Control in Unity3D“ selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe und dass ich alle Stellen, die ich wörtlich oder sinngemäß aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe. Die Arbeit hat bisher in gleicher oder ähnlicher Form oder auszugsweise noch keiner Prüfungsbehörde vorgelegen.

Ich versichere, dass die eingereichte schriftliche Fassung der auf dem beigefügten Medium gespeicherten Fassung entspricht.

Berlin, der 12. Oktober 2015

Ludwig Lauer

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Unity3D hat sich in den letzten Jahren zu einer der bedeutendsten Spieleengines entwickelt. Mit über 45 % des globalen Marktanteils unter Spieleengines trägt Unity3D maßgeblich zu dem Independent Videogame Boom der letzten Jahre bei (vgl. Unity Technologies, 2015f). Unity3D ermöglicht die plattformunabhängige Erstellung von 2D- und 3D-Spielen. Dies hat besonders auf dem fragmentierten Markt der mobilen Spieleindustrie Anklang gefunden. Mit sich kontinuierlich verbessernder Performance und Rendertechniken sind stets ambitioniertere Projekte möglich. Dies bewegt nicht nur Indieentwickler und kleine Teams dazu, sich die Vorteile der Engine zunutze zu machen: Unity3D findet immer mehr Einzug in professionelle Spieleprojekte mit mittleren und großen Teams. Hier sind Tätigkeitsbereiche spezialisiert und klar definiert. Die Anforderungen an die Spieleengine beginnen sich zu wandeln: Parallele Entwicklung muss gewährleistet werden. Der Programmcode muss modular, skalierbar und wiederverwendbar sein. Es darf keinen Raum für Fehlkonfigurationen geben. Neue Geschäftsmodelle wie Free-to-Play bewirken, dass Spiele immer mehr zur Dienstleistung werden. Sie müssen über mehrere Jahre hinweg wartbar und erweiterbar bleiben.

Schaut man sich auf den zahlreichen Websites, Blogs und Foren über Unity3D um, scheinen diese Anforderungen Unitys Schwachstelle zu sein: Frustration und Beschwerden über entstehenden „Spaghetti“-Code, Kritik an der komponentenbasierten Architektur, fehlende Testbarkeit und Aussagen, dass Unity3D nur als Rapid Prototyping Tool tauglich ist und Kollaboration in großen Teams kaum möglich ist (vgl. Reddit Community, 2015; Suzdalnitski, 2015; Cann, 2014; Dąbrowski, 2014).

Oft sollen die Ursachen gerade die Funktionalitäten sein, die unerfahrenen Programmierern den einfachen Einstieg in Unity3D ermöglichen. Eine Ursache ist so gut wie immer vertreten: die Auflösung von Abhängigkeiten und die Kommunikation zwischen Objekten. Enge Kopplung von Abhängigkeiten zwischen Objekten führt zu Starrheit und Zerbrechlichkeit bei Veränderung des Programmcodes. In der objektorientierten Programmierung ist dieses Problem seit Langem bekannt. Es existieren zahlreiche Lösungen für die unterschiedlichsten Situationen und Entwicklungsumgebungen. Oft sind diese Lösungen Frameworks, die eng mit den Begriffen Inversion Of Control und Dependency Injection in Verbindung stehen.

Nun implementiert Unity3D keine klassische objektorientierte Hierarchie, sondern eine komponentenbasierte Architektur. Trotzdem haben auch für Unity eine Handvoll Inversion Of Control Frameworks das Licht der Welt erblickt.

1.2 Zielsetzung und Aufbau

In dieser Arbeit wird sich auf ein Problem beschränkt: die Kopplung von Abhängigkeiten zwischen Entitäten in Unity3D. Ziel ist es die Umsetzung von Spielen in Unity3D zu ermöglichen, ohne dessen Entitäten eng miteinander zu koppeln.

In Kapitel 2 wird zunächst die Problematik der Kopplung beschrieben. Voraussetzung dafür ist, zu erläutern, was Abhängigkeiten sind und warum diese entstehen. Weiter werden die Folgen von enger Kopplung erläutert. Dem Leser soll damit verdeutlicht werden, dass Kopplung direkten Einfluss auf die Testbarkeit und Erweiterbarkeit des Programmcodes hat.

Nachfolgend wird anhand eines Beispiels erklärt, wie lose Kopplung mittels Dependency Injection umzusetzen ist. In diesem Zusammenhang werden auch zugrunde liegende Prinzipien erläutert.

Infolgedessen wird in Kapitel 3 der Schritt zu Unity3D getätigt. Es wird untersucht, inwiefern Unity3D von einer herkömmlichen objektorientierten Hierarchie abweicht und ob dies hinderlich dabei ist, lose Kopplung umzusetzen. Es werden häufig verwendete Praktiken wie das Singletonpattern und Managerobjekte untersucht. Wie im Verlauf der Arbeit im Detail erklärt wird, entstehen Abhängigkeiten unter anderem dadurch, dass Objekte miteinander kommunizieren müssen. Unity3D stellt unterschiedliche Wege der Kommunikation bereit. Es wird nachgeprüft, ob diese Wege es bereits ermöglichen lose Kopplung umzusetzen. Auf Basis dieser Erkenntnisse wird ein Zwischenfazit gezogen, um die Notwendigkeit eines Inversion Of Control Frameworks für Unity3D zu verdeutlichen.

Im vierten Kapitel dieser Arbeit wird ein Inversion Of Control Container für Unity3D implementiert. Dieser Container soll Dependency Injection in MonoBehaviours automatisieren und somit die lose Kopplung unter MonoBehaviours ermöglichen. Die technische Umsetzung wird erläutert. Die mit Unity3D in Verbindung stehenden Hürden bei der Umsetzung werden überwunden und beschrieben. Im Anschluss wird ein Überblick über die vorhandenen Inversion Of Control Frameworks für Unity3D geliefert. Besonderheiten, Funktionsweisen und -umfänge der Frameworks werden beschrieben. Besondere Betrachtung wird StrangeIoC gewidmet: Die frameworkeigene MVC(S)-Architektur und die damit verbundenen Muster und Praktiken werden anhand von Beispielen beschrieben.

Auf Basis dieser Analysen wird in Kapitel 5 eine Aussage über die Tauglichkeit von Inversion Of Control Frameworks in der komponentenbasierten Architektur Unitys

getroffen. Im Speziellen wird ein Fazit über die MVC(S)-Architektur von StrangeIoC zur Entwicklung von Spielen in Unity3D gezogen. Es wird eine Entscheidung getroffen, für welche Anwendungsgebiete die Architektur besonders geeignet ist und für welche nicht. Für diese wird ein alternativer Architekturansatz in Aussicht gestellt.

1.3 Textliche Konventionen

In dieser Arbeit werden Fachbegriffe in englischer Sprache verwendet, sofern kein deutsches Gegenstück gebräuchlich ist.

2 Grundlagen

Geringe Kopplung unter Softwareentitäten (Modulen, Klassen, Methoden) wird als Indiz für gute Softwarearchitektur angesehen. Inversion Of Control und genauer Dependency Injection sind Werkzeuge, die dabei helfen sollen, lose Kopplung umzusetzen. In diesem Teil der Arbeit wird zunächst erläutert, was Kopplung zwischen Entitäten ist und wie diese zustande kommt. Es wird verdeutlicht, warum hohe Kohäsion unter minimaler Kopplung anzustreben ist. Die Vorteile loser Kopplung werden verdeutlicht. Inversion Of Control, Dependency Injection und die eng mit diesen Begriffen zusammenhängenden SOLID-Prinzipien werden beschrieben. Zum besseren Verständnis der vorgestellten Prinzipien, Muster und Praktiken wird lose Kopplung anhand eines Beispiels in einer Konsolenanwendung umgesetzt.

2.1 Kopplung und Kohäsion

Kopplung ist ein Maß, das die Anzahl und Stärke von Abhängigkeiten unter Softwareentitäten einer Anwendung beschreibt (vgl. IEEE Computer Society u. a., 2014, S. 2-3). Ist eine Softwareentität direkt von der Implementation einer anderen Entität abhängig, so sind diese eng miteinander gekoppelt. Ziel ist es, die Anzahl und Stärke von Abhängigkeiten zu minimieren und so lose Kopplung umzusetzen. Martin Fowler beschreibt in dem Artikel „Reducing Coupling“ Kopplung wie folgt:

> „If changing one module in a program requires changing another module, then coupling exists.“ (Fowler, 2001)

Eine Abhängigkeit zwischen Entitäten entsteht, sobald eine Entität von der anderen wissen muss, um zu funktionieren. Abhängigkeiten können unterschiedliche Stärken und Ausprägungen haben. Je größer das Wissen über eine andere Entität ist, desto stärker ist die Abhängigkeit zu ihr. Muss also eine Entität von der konkreten Implementation einer anderen Entität wissen, so existiert eine starke Abhängigkeit und folglich eine enge Kopplung. Wird sich hingegen auf eine Schnittstelle zur Kommunikation geeinigt, existiert lediglich eine Abhängigkeit zu eben dieser Schnittstelle. Die Entitäten selbst sind lose miteinander gekoppelt. Die tatsächliche Implementation einer Entität wird erst zur Laufzeit durch späte Bindung bekannt.

Kopplung steht im Kontrast zu dem Begriff der Kohäsion. Kohäsion beschreibt ein Maß für den inneren Zusammenhalt einer Softwareentität. Hohe Kohäsion liegt dann vor, wenn eine Entität nur Funktionalitäten aufweist, die einer einzelnen, wohldefinierten Aufgabe dienen. Mit anderen Worten: Eine Entität weist eine hohe Kohäsion auf, wenn ihre Elemente in einem engen Zusammenhang stehen. Ein qualitativ hochwertiges Softwaredesign weist demzufolge hohe Kohäsion unter minimaler Kopplung auf (vgl. Schatten u. a., 2010, S. 31). Im nächsten Abschnitt soll Inversion Of Control als Maßnahme präsentiert werden, um Kopplung zu minimieren.

2.2 Inversion of Control

Im ursprünglichen Sinne beschreibt Inversion Of Control nach Johnson u. Foote lediglich die Charakteristik eines Frameworks:

> „[...] methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity.“ (Johnson u. Foote, 1988)

Frameworks diktieren die Architektur und den Kontrollfluss einer Anwendung. Sie definieren einen wiederverwendbaren Teil eines Softwaredesigns. Der vom Nutzer des Frameworks geschriebene, anwendungsspezifische Code wird vom Framework aufgerufen (vgl. Gamma u. a., 1994, S. 27). Der Nutzer gibt also die Kontrolle über den Anwendungsfluss an das Framework. Die Kontrolle ist somit invertiert: Es ist nicht der Nutzer, der das Framework aufruft, sondern das Framework den Code des Nutzers. Diese Tatsache wird oft auch als Hollywood-Prinzip bezeichnet: "Don't call us, we'll call you" (vgl. Fowler, 2005).

Inversion Of Control ist das zentrale Merkmal, welches Frameworks von Bibliotheken unterscheidet. Allein Inversion Of Control ermöglicht die Definition einer wiederverwendbaren Softwarearchitektur, ohne dabei die konkrete Implementation einer Anwendung zu kennen. Das Framework ist lose mit der Implementation der eigentlichen Anwendung gekoppelt.

Es gibt zahlreiche Wege und Entwurfsmuster um Inversion Of Control in unterschiedlichen Aspekten einer Anwendung umzusetzen. Einige Beispiele hierfür sind die Entwurfsmuster Factory, Strategy und Template Method. In dieser Arbeit soll der Fokus auf die Umkehr der Kontrolle über die Auflösung von Abhängigkeiten liegen, mit dem Ziel lose Kopplung zu ermöglichen. Um diese spezielle Art von Inversion Of Control besser abzugrenzen, verwendet Fowler in seinem Artikel „Inversion of Control Containers and the Dependency Injection pattern“ den Begriff Dependency Injection (vgl. Fowler, 2004).

2.3 Dependency Injection

Seemann gibt in „Dependency Injection in .NET“ eine sehr vage Definition des Begriffs Dependency Injection:

> „Dependency Injection is a set of software design principles and patterns that enable us to develop loosely coupled code.“ (Seemann, 2011, S. 4)

Dependency Injection umfasst eine Reihe von Praktiken, die es ermöglichen lose Kopplung umzusetzen. Um welche Praktiken es sich genau handelt, wird in Abschnitt 2.5 anhand eines Beispiels erläutert. Der Kern von Dependency Injection lässt sich jedoch auf eine einzige Praktik reduzieren. James Shore beschreibt diese auf seiner Webseite wie folgt:

> „Dependency injection means giving an object its instance variables.“ (Shore, 2006)

Anstatt, dass sich das Objekt selbst um die Erstellung seiner Abhängigkeiten – seiner Instanzvariablen – kümmert, wird diese Aufgabe an einen Dritten übergeben. Die Verwendung des benötigten Objekts ist von der Konfiguration des Objekts getrennt.

Doch diese Praxis allein ist unzureichend, um lose Kopplung mittels Dependency Injection zu ermöglichen. Es müssen zudem eine Reihe von Prinzipien gewahrt werden. Diese wurden von Robert C. Martin geprägt und sind im Allgemeinen als SOLID-Prinzipien bekannt (vgl. Bender u. McWherter, 2011, S. 49). Diese werden in Abschnitt 2.4 näher beschrieben.

2.4 SOLID Prinzipien

SOLID ist ein Akronym, welches für die folgenden Prinzipien steht: das Single Responsibility Principle, das Open/Closed Principle, das Liskov Substitution Principle, das Interface Segregation Principle und das Dependency-Inversion Principle. Es handelt sich um eine Sammlung von Richtlinien im Umgang mit Abhängigkeiten, die eine saubere Softwarearchitektur ermöglichen sollen (vgl. Metz, 2009, 9:10 Min.). Im folgenden Abschnitt werden diese Prinzipen vorgestellt und ihre Relevanz bei der Umsetzung von Dependency Injection beschrieben.

2.4.1 Single Responsibility Principle

Das Single Responsibility Principle besagt, dass es niemals mehr als einen Grund geben sollte, um eine Klasse zu ändern. Darauf Folgend wird ein „Grund zur Änderung“ in dem Zusammenhang als eine Verantwortung definiert. Jedoch wurde „Grund zur

Änderung“ bewusst gewählt, da es auch legitim ist mehrere Verantwortlichkeiten in einer Klasse unterzubringen, wenn diese sich mit hoher Wahrscheinlichkeit gemeinsam ändern (vgl. Martin, 2003, S. 95-98).

Wird das Single Responsibility Principle gewahrt, so ist in der Regel davon auszugehen, dass Klassen hohe Kohäsion aufweisen. Hohe Kohäsion ist jedoch nicht immer auf das Single Responsibility Principle zurückzuführen. So kann eine Klasse hohe Kohäsion aufweisen, aber trotzdem mehrere Verantwortungen haben.

Dependency Injection wendet das Single Responsibility Principle an, indem es Klassen die Verantwortung über die Erstellung von Instanzen ihrer Abhängigkeiten entzieht (vgl. Seemann, 2011, S. 24).

2.4.2 Open/Closed Principle

Softwareentitäten (Klassen, Methoden, Funktionen) sollen offen für Erweiterung, aber geschlossen für Modifikation sein. Mit „Offen für Erweiterung“ ist gemeint, dass es möglich sein soll, die Entität in ihrem Umfang und Verhalten zu erweitern. „Geschlossen für Modifikation“ zum anderen bedeutet, dass zur Erweiterung keine Änderungen an der Entität an sich notwendig sind (vgl. Martin, 2003, S.99-100).

Abstraktion ermöglicht die Umsetzung des Open/Closed Principles. Wird von einer abstrakten Basisklasse geerbt, so ist es möglich das Verhalten der Abstraktion zu ändern, ohne die Abstraktion selbst zu manipulieren. Die Entwurfsmuster Strategy und Template Method sind Beispiele für das Open/Closed Principle.

Auch Dependency Injection setzt das Open/Closed Principle um, indem Abhängigkeiten lediglich in Form von Abstraktionen existieren. Damit ist dieses Prinzip eine wichtige Grundlage für lose Kopplung (vgl. Seemann, 2011, S. 106).

2.4.3 Liskov Substitution Principle

Das Liskov Substitution Principle beschreibt eine grundlegende Anforderung beim Einsatz von Vererbung: Untertypen einer Basisklasse sollen untereinander ersetzbar sein, ohne dabei das Verhalten des Programms selbst zu verändern. Das Programm selbst darf zu keinem Zeitpunkt einen Untertyp anders als einen anderen Untertypen behandeln (vgl. Martin, 2003, S. 111-112).

Dependency Injection ermöglicht es, dass Abhängigkeiten beliebig austauschbar sind, solange sie die nötige Abstraktion implementieren. Dies ist jedoch nur möglich, wenn das Liskov Substitution Prinzip gewahrt wird (vgl. Seemann, 2011, S. 9-10).

Dieses Prinzip kommt besonders beim Unittesting zum Vorschein: Abhängigkeiten eines Systems Under Test können durch Test Doubles ersetzt werden, ohne das Verhalten des Systems selbst zu verändern (vgl. Seemann, 2011, S.19).

2.4.4 Interface Segregation Principle

Anstatt umfangreiche Interfaces zu nutzen, sollten diese in mehrere kleine Interfaces aufgeteilt werden. Die Aufteilung soll nach Methodengruppen erfolgen, entsprechend wie sie von anderen Entitäten, die das Interface nutzen, gebraucht werden. So wird gewährleistet, dass andere Entitäten nur von Abstraktionen mit hoher Interfacekohäsion abhängig sind. Auch wenn diese Abstraktionen von einer einzigen nicht-kohäsiven Entität implementiert werden (vgl. Martin, 2003, S. 135).

Das Interface Segregation Principle scheint zunächst nicht mit Dependency Injection im Zusammenhang zu stehen. Es ist eher eine Richtlinie, die bei der Anwendung von Dependency Injection gewahrt werden sollte. Wird eine Abhängigkeit von einem einzigen großen Interface repräsentiert, lenkt dieses Interface in die Richtung einer konkreten Implementation. Die Folge ist, dass Entitäten schwieriger auszutauschen sind, weil sie Interfacemethoden implementieren, die sie eigentlich gar nicht benötigen, um eine Abhängigkeit zu erfüllen (vgl. Seemann, 2011, S. 284).

2.4.5 Dependency Inversion Principle

Das Dependency Inversion Principle besteht aus zwei Teilen:

- High-Level Module sollen nicht von Low-Level Modulen abhängig sein. Beide sollten von Abstraktionen abhängen.
- Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen (vgl. Martin, 2003, S. 127).

Zum einen beschreibt das Dependency Inversion Principle, dass die Architektur einer Anwendung in Schichten erfolgen sollte. Dabei soll die abstraktere, höher liegende Schicht ihre Abhängigkeiten in Form von Interfaces deklarieren. Die ihr untergeordnete Schicht wird basierend auf diesen Interfaces realisiert. Folglich sind die übergeordneten Schichten nicht von untergeordneten Schichten abhängig. Stattdessen sind untergeordnete Schichten von den abstrakten Interfaces in den übergeordneten Schichten abhängig (vgl. Martin, 2003, S. 128-129).

Abbildung 2.1 auf der nächsten Seite verdeutlicht diese Invertierung der Abhängigkeiten: Anstatt, dass das *Policy*-Modul – wie in der linken Bildhälfte gezeigt – von der Implementation des *Mechanism*-Moduls abhängt, besteht lediglich eine Abhängigkeit an ein *Mechanism*-Interface. Dieses Interface befindet sich innerhalb des höherliegenden

Policy-Moduls. Die tatsächliche Implementation erfolgt innerhalb des untergeordneten *Mechanism*-Moduls, wodurch eine Abhängigkeit an das *Policy*-Modul entsteht. Demzufolge haben Änderungen des *Mechanism*-Moduls keinen Einfluss auf das *Policy*-Modul.

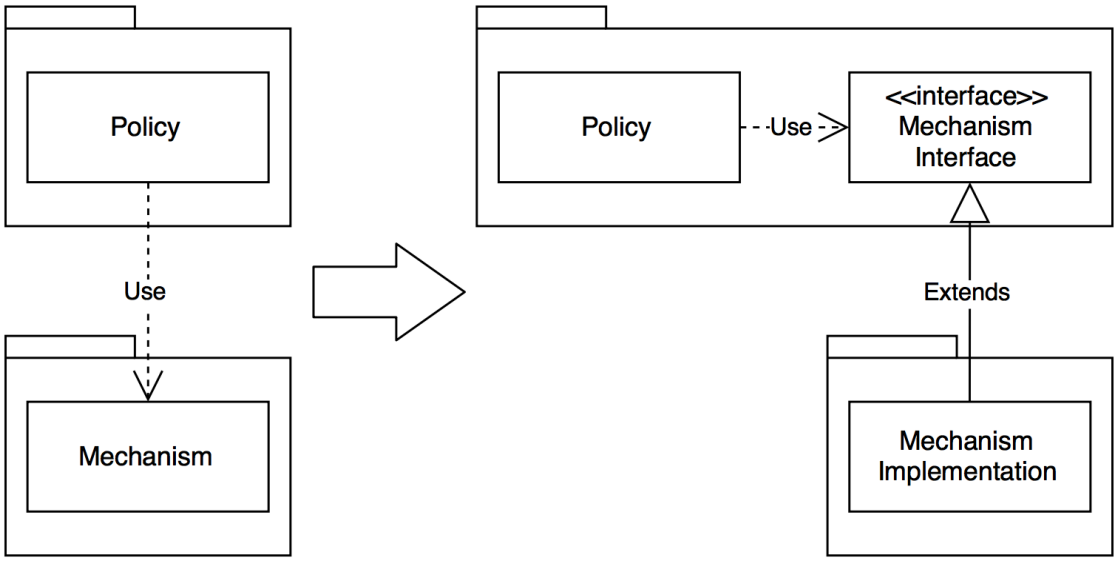

Abb. 2.1: Invertierung der Abhängigkeiten zwischen Schichten. Quelle: in Anlehnung an Martin, 2003, S.129

Zum anderen beschreibt Martin eine weitere, allgemeine Interpretation des Dependency Inversion Principles:

> „Depend on abstractions.“ (Martin, 2003, S. 129)

So sollen Klassen keine Referenzen zu anderen konkreten Klassen halten, von ihnen ableiten oder Methoden ihrer Basisklasse überschreiben. Diese Praxis wurde bereits von der Gang of Four in „Design Patterns - Elements of Reusable Object-Oriented Software“ mit den folgenden Worten beschrieben:

> „Program to an interface, not an implementation.“ (Gamma u. a., 1994, S. 17-18)

Die Deklaration von Abhängigkeiten in Form von Abstraktionen ist eine Grundlage, um lose Kopplung mittels Dependency Injection umzusetzen. Späte Bindung ermöglicht es, dass die konkrete Implementation auf diesem Wege erst zur Laufzeit bekannt sein muss.

2.5 Umsetzung von Dependency Injection

Um zu erläutern, wie die in den vorherigen Abschnitten erklärten Prinzipien bei der Umsetzung von Inversion Of Control mittels Dependency Injection zusammenspielen, wird ein Beispiel herangezogen. Anhand dieses Beispiels werden Entwurfsmuster die

im direkten Zusammenhang zu Dependency Injection stehen, beschrieben. Um das Beispiel so einfach wie möglich zu halten, wurde hierfür eine Konsolenanwendung in C# gewählt.

2.5.1 Ausgangssituation

Eine gewünschte Funktionalität eines zu entwickelnden Spiels ist es, auszugeben, dass ein verwundbares Objekt Schaden genommen hat. Die in Abbildung 2.2 dargestellte Klasse *DamageRenderer* setzt diese Funktionalität um. Alle qualitativen und funktionalen Anforderungen werden von der Klasse erfüllt. Sie kann von allen Objekten wiederverwendet werden, wenn diese Schaden nehmen. Die Klasse beschränkt ihre Verantwortung auf eine einzige Funktionalität: Sie formatiert die Information in ein lesbares Format und gibt diese in der Konsole aus.

```
public class DamageRenderer
{
    public void Render(string target, int damage)
    {
        var output = string.Format("{0} was attacked and took {1}
            damage.", target, damage);
            System.Console.WriteLine(output);
    }
}
```

Abb. 2.2: Ausgangssituation des Beispiels. Quelle: Eigene Abbildung

2.5.2 Neue Anforderungen

Im Verlauf der Entwicklung des Spieles wird deutlich, dass das Spiel in verschiedenen Sprachen veröffentlicht werden soll. Diese auch als Lokalisierung bekannte Anforderung ist eine gängige Anforderung für jegliche Art von Software. Es wird mit dem Refactoring begonnen und die Klasse *EnglishLocalization* extrahiert. In Abbildung 2.3 wird das Resultat des Refactorings gezeigt. Die Verantwortung über die Lokalisierung wurde in eine allein dafür vorgesehene Klasse verschoben.

```
public class DamageRenderer
{
    private readonly EnglishLocalization localization;

    public DamageRenderer()
    {
        localization = new EnglishLocalization();
    }

    public void Render(string target, int damage)
    {
        var output = string.Format(localization.GetText(), target,
            damage);
        System.Console.WriteLine(output);
    }
}

public class EnglishLocalization
{
    public string GetText()
    {
        return "{0} was attacked and took {1} damage.";
    }
}
```

Abb. 2.3: Extraction der englischen Lokalisierung aus der existierenden Klasse. Quelle: Eigene Abbildung

Die Klasse *DamageRenderer* funktioniert weiterhin wie im vorherig gezeigten Beispiel. Der Unterschied ist, dass *DamageRenderer* nun eine Abhängigkeit zu *EnglishLocalization* besitzt. Wie kann es nun ermöglicht werden, die englische Lokalisierung mit der Deutschen aus Abbildung 2.4 auszutauschen? Das Problem hierbei ist, dass *DamageRenderer* eine konkrete Abhängigkeit zu *EnglishLocalization* besitzt. Mit anderen Worten, sie sind eng miteinander gekoppelt. Die Folge ist, dass *EnglishLocalization* nicht – ohne *DamageRenderer* zu ändern – durch *GermanLocalization* ausgetauscht werden kann.

```
public class GermanLocalization
{
    public string GetText()
    {
        return "{0} wurde angegriffen und hat {1} Schaden genommen.";
    }
}
```

Abb. 2.4: Deutsche Lokalisierung. Quelle: Eigene Abbildung

2.5.3 Programming to an Interface

Um lose Kopplung umzusetzen, darf keine Abhängigkeit zu einer konkreten Implementation existieren. Stattdessen muss sich auf eine Abstraktion als Abhängigkeit verlassen werden. In dem Beispiel in Abbildung 2.5 wird dies umgesetzt, indem *English-* und *GermanLocalization* das Interface *ILocalizationService* implementieren. Dadurch, dass *DamageRenderer* nur noch eine Variable des Interfaces *ILocalizationService* hält, lässt sich die konkrete Implementation austauschen. Voraussetzung hierfür ist jedoch weiterhin, dass *DamageRenderer* geändert wird. Der Grund dafür ist, dass *EnglishLocalization* im Konstruktor von *DamageRenderer* instanziiert wird. Somit ist *DamageRenderer* weiterhin eng mit *EnglishLocalization* gekoppelt.

```
public class DamageRenderer
{
    private readonly ILocalizationService localization;

    public DamageRenderer()
    {
        localization = new EnglishLocalization();
    }

    public void Render(string target, int damage)
    {
        var output = string.Format(localization.GetText(), target,
            damage);
        System.Console.WriteLine(output);
    }
}

public interface ILocalizationService
{
    string GetText();
}

public class EnglishLocalization : ILocalizationService
{
    public string GetText()
    {
        return "{0} was attacked and took {1} damage.";
    }
}

public class GermanLocalization : ILocalizationService
{
    public string GetText()
    {
        return "{0} wurde angegriffen und hat {1} Schaden genommen.";
    }
}
```

Abb. 2.5: Beispiel von „Depend on abstractions“. Quelle: Eigene Abbildung

2.5.4 Injecting Dependencies

Um die Klassen lose miteinander zu koppeln, kommt an dieser Stelle der Einsatz von Dependency Injection ins Spiel: Anstatt, dass sich eine Klasse selbst um die Auflösung ihrer Abhängigkeiten kümmert, sollte sie von außen mit diesen versorgt werden. Die Verantwortung über die Auflösung der Abhängigkeiten wird abgegeben. Die Klasse selbst macht lediglich deutlich, dass Sie ein Objekt benötigt, welches die abstrakten Bedürfnisse eines Interfaces umsetzt. In diesem Fall handelt es sich um die Bedürfnisse eines *ILocalizationServices*. Dependency Injection lässt sich auf unterschiedlichen Wegen umsetzen. Die Gebräuchlichsten sind Constructor Injection und Property Injection.

Constructor Injection

Die Abhängigkeit wird über den Konstruktor bei der Instanziierung des Objektes bereitgestellt. In Abbildung 2.6 auf der nächsten Seite wird Constructor Injection in der *DamageRenderer*-Klasse angewandt. Eine Instanz von *ILocalizationService* muss bei Aufruf des Konstruktors übergeben werden.

Constructor Injection stellt sicher, dass die nötige Abhängigkeit auch tatsächlich verfügbar ist, indem eine Instanziierung des Objektes (ein Konstruktoraufruf) ohne diese nicht möglich ist. Damit wird erzwungen, dass der gesamte Abhängigkeitsgraph aufgebaut wird, bevor die Klasse genutzt werden kann. Eine Guard Clause stellt sicher, dass nicht *null* als Argument übergeben wird. Constructor Injection sollte in den meisten Fällen angewandt werden, denn in der Regel ist eine Abhängigkeit zwingend notwendig.

```
public class DamageRenderer
{
    private readonly ILocalizationService localization;

    public DamageRenderer(ILocalizationService localization)
    {
        if (localization == null)
        {
            throw new ArgumentNullException("localization");
        }

        this.localization = localization;
    }

    public void Render(string target, int damage)
    {
        var output = string.Format(localization.GetText(), target,
            damage);
        System.Console.WriteLine(output);
    }
}
```

Abb. 2.6: Beispiel von Constructor Injection. Quelle: Eigene Abbildung

Property Injection

Ist eine Abhängigkeit optional, kann die Klasse über ein Property mit einer Instanz dieser Abhängigkeit versorgt werden. Abbildung 2.7 auf der nächsten Seite zeigt die Umsetzung von Property Injection anhand des gewählten Beispiels. Die Klasse stellt ein öffentlich setzbares Property vom Typ der Abhängigkeit bereit. In dem gezeigten Beispiel ist Property Injection wenig angebracht, da die Abhängigkeit notwendig ist. Man könnte jedoch, falls kein *LocalizationService* bereitgestellt wird, beispielsweise im Getter des Properties eine Instanz von *EnglishLocalization* erstellen und diese so als Local Default verwenden.

Das Open/Closed Principle lässt sich somit auch mit Properties umsetzen: Properties bieten die Möglichkeit eine optionale Abhängigkeit zur Erweiterung der Funktionalität einer Klasse preiszugeben, ohne dabei die Klasse selbst zu modifizieren.

Ein Nachteil ist, dass andere Klassen zu jeder Zeit Veränderungen an dem Property vornehmen können. Aus diesem Grund wird Property Injection in manchen Kreisen kritisiert, da es die Datenkapselung bricht (vgl. Butler, 2013).

In einigen Fällen muss – aufgrund von Beschränkungen durch die Verwendung von Frameworks wie z. B. ASP.NET – auf Constructor Injection verzichtet werden. Das Framework schreibt das Vorhandensein eines Default Constructors vor, der zur Initialisierung genutzt wird. In diesen Fällen kann auch alternativ auf Property Injection zurückgegriffen werden.

```
public class DamageRenderer : IDamageRenderer
{
    public ILocalizationService Localization { private get; set; }

    public void Render(string target, int damage)
    {
        if (localization == null)
        {
            throw new ArgumentNullException("localization");
        }

        var output = string.Format(Localization.GetText(), target,
            damage);
        System.Console.WriteLine(output);
    }
}
```

Abb. 2.7: Beispiel von Property Injection. Quelle: Eigene Abbildung

2.5.5 Composition Root

In den vorangegangenen Abschnitten wurde verdeutlicht, dass Dependency Injection es verlangt, die Verantwortung über die Auflösung der Abhängigkeiten abzugeben. Es wurde jedoch nicht beschrieben, an wen diese Verantwortung übertragen werden soll. Die Antwort ergibt sich durch die konsequente Umsetzung von Dependency Injection: Abhängigkeiten werden in der Objekthierarchie nach oben gereicht bis dies nicht mehr möglich ist. Die Auflösung der Abhängigkeiten sollte demzufolge so nah wie möglich beim Einstiegspunkt der Applikation geschehen. Soweit es möglich ist, sollte die Initialisierungslogik aller Objekte an diesem zentralen Ort umgesetzt werden. Dieser zentrale Ort wird als Composition Root bezeichnet (vgl. Seemann, 2011, S. 76-77).

In einer Konsolenanwendung – wie dem gezeigten Beispiel – erfüllt die Main-Methode die Voraussetzungen für einen Composition Root: Sie ist der Einstiegspunkt in die Anwendung und zugleich der höchste Punkt in der Objekthierarchie. Abbildung 2.8 zeigt den Composition Root der Beispielanwendung.

```
public class CompositionRoot
{
    public static void Main(string[] args)
    {
        var localization = new GermanLocalization();
        var renderer = new DamageRenderer(localization);
        renderer.Render("Mario", 10);
    }
}
```

Abb. 2.8: Die Main-Methode erfüllt die Eigenschaften des Composition Roots in einer Konsolenanwendung. Quelle: Eigene Abbildung

Im Composition Root wird die Konfiguration der Anwendung vorgenommen. Es ist der einzige Ort, an dem die konkreten Instanzen von Objekten bekannt sein müssen. Er sollte unabhängig von dem Rest der Anwendung kompilierbar sein und sich deswegen in einem separaten Assembly befinden.

2.5.6 Feste und unbeständige Abhängigkeiten

Nicht jede Abhängigkeit muss lose mit der Anwendung gekoppelt werden. Tatsächlich ist dies für bestimmte, feste Abhängigkeiten unnötig. Wichtig ist jedoch unterscheiden zu können, wann eine Abhängigkeit als fest („stable“) und wann als unbeständig („volatile“) angesehen werden sollte.

Kann davon ausgegangen werden, dass ein bereits existierendes Modul niemals ausgetauscht oder verändert wird, so kann es als eine feste Abhängigkeit angesehen werden. Ein Beispiel hierfür sind die meisten Module der Base Class Library, da sie fester Bestandteil des .NET Frameworks sind. Ob ein Modul als feste Abhängigkeit angesehen werden kann, lässt sich anhand der folgenden Kriterien beurteilen:

- Die Klasse oder das Modul existiert bereits.
- Es ist zu erwarten, dass neue Versionen der Abhängigkeit keine Änderungen enthalten werden, die Auswirkungen auf den Programmcode haben.
- Das Modul beinhaltet ausschließlich deterministische Algorithmen.
- Es ist nicht zu erwarten, dass das Modul jemals durch ein anderes ausgetauscht wird.

Den Gegensatz zu festen Abhängigkeiten bilden Klassen und Module, bei denen es absehbar ist, dass sie sich verändern werden. Ist eine Abhängigkeit unbeständig, sollte sie lose mit dem Rest der Anwendung gekoppelt werden, um ihre Austauschbarkeit zu ermöglichen. Eine Abhängigkeit wird als unbeständig angesehen, wenn eines der folgenden Kriterien auf sie zutrifft:

- Durch die Abhängigkeit wird eine bestimmte Laufzeitumgebung vorausgesetzt.
- Die Abhängigkeit existiert noch nicht.
- Die Abhängigkeit nutzt eine Bibliothek, die nicht auf allen Systemen vorhanden ist.
- Die Abhängigkeit weist nichtdeterministisches Verhalten auf (vgl. Seemann, 2011, S. 23-24).

Ein Beispiel für eine in der Base Class Library vorhandene Klasse, die als unbeständige Abhängigkeit behandelt werden sollte, ist *System.Random*. Die Klasse weist nichtdeterministisches Verhalten auf, was sich besonders auf die Testbarkeit der Anwendung auswirkt.

2.6 Resultierende Vorteile und Möglichkeiten

In den vorherigen Abschnitten wurde ausführlich beschrieben, wie und warum lose Kopplung umzusetzen ist. Jedoch sind eventuell die Gründe für lose Kopplung, aufgrund des einfach gehaltenen Beispiels, noch nicht ganz ersichtlich geworden. Besonders große und komplexe Softwareprojekte profitieren maßgeblich von loser Kopplung. Welche Faktoren genau beeinflusst werden, wird im Folgenden erläutert.

2.6.1 Erweiterbarkeit

Während der Entwicklung einer Anwendung werden neue Anforderungen bekannt und existierende Anforderungen beginnen sich zu verändern. Trotzdem muss die Anwendung erweiterbar und wartbar bleiben. Mit zunehmender Größe der Anwendung, wird es umso problematischer sie zu erweitern. In sehr großen Softwareprojekten ist es schwer abzusehen, welche Auswirkungen auch nur die kleinste Änderung hat. Aus diesem Grund ist es notwendig, Entitäten so autonom wie möglich zu halten. Dadurch, dass die Abhängigkeiten einer Entität auf Interfaces reduziert werden, haben Änderungen der konkreten Implementation keinen Einfluss auf sie.

Lose Kopplung setzt das Open/Closed Principle in die Tat um: Module sind offen für Erweiterung und geschlossen für Veränderung. Maßgeblich verantwortlich hierfür ist das Konzept der späten Bindung, welches sich durch „programming to an interface“ konsequent umsetzen lässt. Die konkrete Implementation ist austauschbar und wird erst zur Laufzeit bekannt. Sie ist unabhängig vom Rest der Anwendung veränderbar.

Dependency Injection hilft dabei, Verantwortungen klarer aufzuteilen und zu definieren. Durch das Einhalten des Single Responsibility Principles wird es deutlicher an welchen Stellen Änderungen gemacht werden müssen. Oft genügt es eine konkrete Implementation durch eine andere auszutauschen, um ein gewünschtes Verhalten zu erzielen. Dependency Injection macht dies trivial.

2.6.2 Wiederverwendbarkeit

Einzelne Softwaremodule sind wiederverwendbar, da diese lose zu ihren Abhängigkeiten gekoppelt sind. Die eigentliche Implementation dieser Abhängigkeiten ist für das Modul nicht relevant. Ist in einer anderen Umgebung eine andere konkrete Implementation notwendig, so kann sie einfach ausgetauscht werden.

2.6.3 Parallele Entwicklung

Lose Kopplung ermöglicht die parallele Entwicklung einer Softwareanwendung, indem klare Schnittstellen zur Kommunikation definiert werden. Unter den kollaborierenden Entwicklern muss sich lediglich auf diese Schnittstellen geeinigt werden. Die konkreten Implementationen sind nicht mehr relevant und können unabhängig voneinander im Parallelen umgesetzt werden.

2.6.4 Testbarkeit

Wie lose Kopplung die Testbarkeit einer Anwendung drastisch erleichtert, lässt sich am besten anhand eines Beispiels erläutern: Abbildung 2.9 zeigt die Implementation eines simplen Spiels: Der Spieler soll das Resultat eines Würfelwurfs erraten. Mit der *Roll*-Methode der gezeigten *GuessANumber*-Klasse soll der Würfelwurf ausgeführt werden. Darauf folgend kann der Spieler – durch den Aufruf der *MakeGuess*-Methode – den Versuch vornehmen, das Ergebnis zu erraten.

```
public class GuessANumber
{
    private IRandom random;
    private int result;
    private const int max = 6;

    public GuessANumber(IRandom random)
    {
        this.random = random;
    }

    public GuessANumber Roll()
    {
        result = random.Range(1, max);
        return this;
    }

    public bool MakeGuess(int guess)
    {
        return result == guess;
    }
}

public interface IRandom
{
    int Range(int min, int max);
}
```

Abb. 2.9: Beispiel einer durch Dependency Injection testbaren Klasse. Quelle: Eigene Abbildung

Die *GuessANumber*-Klasse hat eine unbeständige („volatile“) Abhängigkeit zu *IRandom*.

Diese Entscheidung wurde bewusst getroffen, da das Generieren einer Zufallsnummer nicht deterministisch ist. Die Abhängigkeit zu *IRandom* wird über Constructor Injection aufgelöst. Eine Klasse, die das *IRandom*-Interface implementiert, könnte mithilfe von *UnityEngine.Random* oder *System.Random* umgesetzt werden.

Abbildung 2.10 zeigt einen Unittest, der sicherstellen soll, dass die *MakeGuess*-Methode korrekt feststellen kann, ob richtig geraten wurde. Um die *GuessANumber*-Klasse durch einen Unittest auf korrekte Funktionalität prüfen zu können, muss das Resultat deterministisch sein. Hierzu wird anstatt einer echten Implementation einer *Random*-Klasse ein Stub des *IRandom*-Interfaces erzeugt. Das Erzeugen eines Stubs kann durch Verwendung des Isolation-Frameworks NSubstitute automatisiert werden: Durch Aufruf der Methode *For<T>* der Klasse *Substitute* wird ein dynamischer Stub für *IRandom* erzeugt. Der Stub wird so konfiguriert, dass er für jeden Aufruf der *Range*-Methode – unabhängig von beiden Parametern – einen vorgegebenen Wert zurückgibt. Das Ergebnis eines Würfelwurfs durch die *Roll*-Methode wird somit diesem vorgegebenen Wert entsprechen. Damit ist zu erwarten, dass der Aufruf der *MakeGuess*-Methode mit dem vorgegebenen Wert, „true“ zurückgibt. Die tatsächliche Funktionalität der Methode kann nun durch ein Assert-Statement überprüft werden.

```
using NUnit.Framework;
using NSubstitute;

[TestFixture]
public class GuessANumberTests
{
    [Test]
    public void ShouldReturnTrueForRightGuess()
    {
        var expected = 6;

        var randomStub = Substitute.For<IRandom>();
        randomStub.Range(Arg.Any<int>(),
           Arg.Any<int>()).Returns(expected);

        var guessANumber = new GuessANumber(randomStub);
        guessANumber.Roll();

        Assert.True(guessANumber.MakeGuess(expected));
    }
}
```

Abb. 2.10: Testen der *GuessANumber*-Klasse. Quelle: Eigene Abbildung

Lose Kopplung ermöglicht, dass Module in Isolation betrachtet werden können. Durch Dependency Injection können zugrunde liegende Implementationen einer Abstraktion durch Fakes, Mocks und Stubs ausgetauscht werden. Es muss mit Bedacht auf die Testbarkeit einer Anwendung entschieden werden, ob eine Abhängigkeit fest oder unbeständig ist. Der sicherste Weg Testbarkeit zu gewährleisten, ist es Test Driven

Development in Kombination mit Dependency Injection zu praktizieren (vgl. Bender u. McWherter, 2011, S. 10).

3 Analyse der Architektur von Unity3D

In dem letzten Kapitel wurde beschrieben, was lose Kopplung ist und wie Inversion Of Control, und genauer Dependency Injection, dabei helfen, diese umzusetzen. Es wurde beschrieben, dass lose Kopplung maßgeblich zu der Qualität eines Softwareproduktes beiträgt. Dependency Injection wurde beispielhaft in einer Konsolenanwendung angewendet. In diesem Kapitel werden relevante Unterschiede zwischen der klassischen objektorientierten Programmierung bei der Entwicklung einer Konsolenanwendung und der komponentenbasierten Architektur von Unity3D beschrieben. Gute und schlechte Praktiken bei der Arbeit mit Unity3D werden erläutert. Unity3D stellt von sich aus Mittel bereit, um die Kommunikation zwischen Klassen zu ermöglichen. Es wird untersucht, ob diese es bereits ermöglichen lose Kopplung umzusetzen.

3.1 Komponentenbasierte Architektur von Unity3D

Jede Spieleengine implementiert eine Form eines Game Object Models. Ein Game Object Model beschreibt Hilfsmittel, die es ermöglichen Entitäten innerhalb der Spielwelt zu modellieren und zu simulieren (vgl. Gregory, 2014, S. 854). Meistens bestehen diese Hilfsmittel aus zwei Teilen: einer visuellen Repräsentation der Spielobjekte innerhalb des Editors der Engine und einer Laufzeitumgebung, die die Entwicklung neuer Spielobjekte ermöglicht. In jedem Fall wird ein Framework bereitgestellt, das die Konstruktion neuer Spielobjekte erlaubt (vgl. Gregory, 2014, S. 869).

In klassischen objektorientierten Implementationen eines Game Object Models werden Spielobjekte von einer Vererbungshierarchie abgeleitet. Es handelt sich um eine monolithische Klassenhierarchie: Alle Objekte werden von einer gemeinsamen Basisklasse namens *GameObject* abgeleitet. Diese Basisklasse enthält Funktionalitäten, die für alle Klassen notwendig sind. Ein Beispiel hierfür ist Serialisierung. Mit Wachstum des Projektes wächst auch die Hierarchie stets tiefer und weiter. Es folgt eine Reihe von Problemen: Klassen, die sich tief in der Vererbungshierarchie befinden, sind schwierig zu verstehen und zu verändern, da hierfür auch alle Basisklassen bekannt sein müssen.

Es kann nur ein Merkmal als Basis für eine Vererbungshierarchie gewählt werden.

Ein Merkmal kann beispielsweise die Beweglichkeit eines Objekts sein. Eine andere Charakteristik wäre die Fähigkeit mit anderen Objekten zu kollidieren. Möchte man ein Objekt mit beiden Merkmalen haben, so stößt man schnell an die Grenzen einer solchen Architektur: Die benötigten Funktionalitäten müssen in der Hierarchie nach oben verschoben werden und bewirken so, dass sie eventuell in Klassen vorhanden sind in denen sie nicht gebraucht werden. In Abbildung 3.1 wird das Problem veranschaulicht. Das Problem kann in einigen Sprachen auch durch Mehrfachvererbung gelöst werden, was aber zu einem Deadly Diamond führen kann: Ein Deadly Diamond entsteht indem eine Klasse von zwei Klassen erbt, die wiederum von der selben Basisklasse erben.

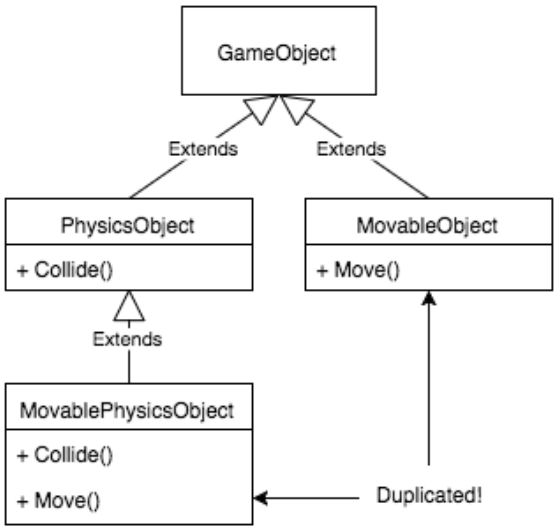

Abb. 3.1: Doppelte Logik aufgrund einer monolithischen Vererbungshierarchie. Quelle: in Anlehnung an Nystrom, 2014, S. 215

Um dem Auftreten dieser Probleme vorzubeugen, implementiert Unity3D ein komponentenbasiertes Game Object Model. Das Grundprinzip dieses Models ist Komposition der Vererbung vorzuziehen. Anstatt Objekte mittels Vererbung und des „is a“-Verhältnisses zu erweitern, erfolgt die Komposition eines Spielobjekts aus einzelnen Komponenten über das „has a“-Verhältnis. In Unity werden Spielobjekte von der Klasse *GameObject* abgebildet. Ein *GameObject* fungiert als Knotenpunkt für eine variable Anzahl von Komponenten und hat kaum bis gar keine weitere Funktionalität (vgl. Gregory, 2014, S. 887). Funktionalität wird erst durch das Anfügen von Komponenten hinzugefügt. Komponenten werden in Unity3D von der Klasse *MonoBehaviour* abgeleitet. Abbildung 3.2 auf der nächsten Seite zeigt eine vereinfachte Darstellung der komponentenbasierten Architektur Unitys. Die tatsächliche Implementation ist jedoch nicht bekannt, da es sich bei Unity3D um proprietäre Software handelt.

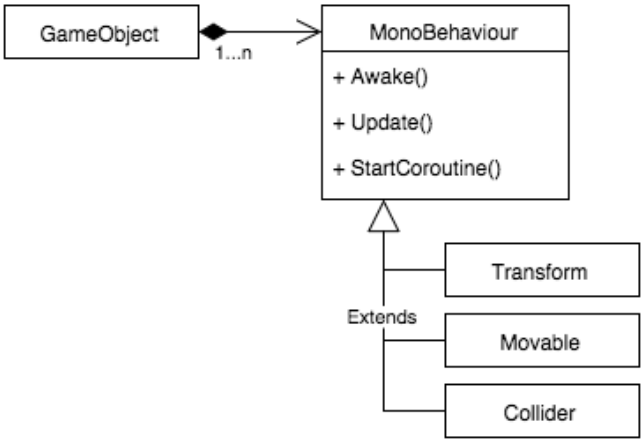

Abb. 3.2: Vereinfachte Darstellung der komponentenbasierten Architektur Unitys. Quelle: in Anlehnung an Gregory, 2014, S. 886

3.2 Programmierung und Konfiguration

MonoBehaviours sind die Schnittstelle, die es Programmierern ermöglicht die Engine mit individuellem Verhalten zu erweitern, um ein Spiel umsetzen zu können. Unity3D verwendet die .NET-kompatible Laufzeitumgebung Mono um die plattformunabhängige Entwicklung von Komponenten in C# und UnityScript zu ermöglichen. Dem Nutzer stehen folglich objektorientierte Sprachfeatures wie Polymorphismus und Vererbung innerhalb der Komponenten zur Verfügung. Komponenten können durch die Implementation von einer Reihe von vordefinierten Eventfunktionen Einfluss auf das Spielgeschehen nehmen.

Der Rest – also die Konfiguration und Komposition von Komponenten – erfolgt in einem visuellen Editor. Abbildung 3.3 auf der nächsten Seite zeigt das Benutzerinterface des Unity Editors. Der Editor erlaubt das Anlegen von Szenen, die den Lebensraum für Spielobjekte darstellen. Spielobjekte werden im sogenannten Hierarchiefenster erstellt und in einem Szenegraph angeordnet. Per Drag-and-Drop können Spielobjekten Komponenten hinzugefügt werden. Öffentliche Member einer Komponente können im Inspector konfiguriert werden. Wird eine Szene geladen, kümmert sich Unity3D, um die Instanziierung aller sich in der Szene befindlichen Spielobjekte und deren Komponenten.

Abb. 3.3: Das Benutzerinterface vom Unity Editor. Quelle: Unity Technologies, 2015c

Betrachtet man die in Abschnitt 2.2 auf Seite 5 gegebene Definition von Inversion Of Control, stellt man fest, dass diese auch auf Unity3D zutrifft. Die Kontrolle über die Komposition und Lebenszeit von Objekten ist bereits invertiert: Sie liegt in den Händen von Unity3D. Ein Indiz hierfür ist, dass Unity3D keine Konstruktoren in *MonoBehaviours* zulässt. Es ist also nicht möglich, Komponenten über einen Konstruktor mit ihren Abhängigkeiten zu versorgen. Aus diesem Grund stellt Unity3D alternative Wege zur Referenzierung und Kommunikation mit anderen Komponenten bereit. Diese werden im folgenden Abschnitt erläutert.

3.3 Kommunikation und Abhängigkeiten

In Unity3D soll jede Komponente eine einzige Funktionalität kapseln und sie wiederverwendbar machen. Mit anderen Worten: Komposition anstelle von Vererbung soll es ermöglichen Funktionalitäten aus unterschiedlichen Bereichen in wiederverwendbare Komponenten aufzuteilen, ohne sie dabei miteinander zu koppeln (vgl. Nystrom, 2014, S. 213). In der Realität ist dies jedoch nur begrenzt möglich: Komponenten sind Teile eines größeren Ganzen und müssen somit kommunizieren und interagieren. Komponenten haben folglich Abhängigkeiten zueinander.

Oft beschränkt sich die Kommunikation auf Komponenten desselben GameObjects. Es kommt jedoch auch vor, dass ein MonoBehaviour mit einem MonoBehaviour eines

anderen GameObjects kommunizieren muss. Folglich wird zwischen zwei Arten der Kommunikation unterschieden:

- Die Intra-Objekt-Kommunikation erfolgt unter MonoBehaviours desselben GameObjects oder Kindern von diesem.
- Die Inter-Objekt-Kommunikation beschreibt die Kommunikation zu anderen GameObjetcts, die ansonsten in keiner Relation zueinanderstehen.

In beiden Fällen kann die Kommunikation direkt oder indirekt erfolgen. Bei direkter Kommunikation ist es notwendig, dass entweder Sender oder Empfänger einander kennen. Eines der Objekte hat also eine Abhängigkeit zu dem anderen. Kommunikation ist indirekt, wenn sie über ein drittes Objekt – einem Vermittler – erfolgt. In den meisten Fällen wird eine Referenz auf ein weiteres Objekt benötigt und es existiert somit eine Abhängigkeit zu diesem.

Im folgenden Abschnitt wird untersucht, ob Unitys integrierte Möglichkeiten Abhängigkeiten aufzulösen es ermöglichen, lose Kopplung zwischen Komponenten umzusetzen.

3.3.1 Referenzieren von anderen MonoBehaviours

Unity3D bietet unterschiedliche Möglichkeiten, um Zugriff auf Spielobjekte und Komponenten zu erlangen. Es werden zahlreiche Game Object Queries für die Inter-Objekt-Kommunikation bereitgestellt. Um andere Komponenten referenzieren zu können, existieren sogenannte Component Queries. Des Weiteren ist auch eine Kommunikation über Events und Messages möglich.

SerializeField

SerializeField ist ein Attribut, das ermöglicht private Member einer Komponente zu serialisieren und somit im Inspektor zu setzen. Setzbare Datentypen sind primitive Datentypen wie *int*, *float*, *bool* und in Unity3D eingebaute Datentypen wie *Vector*, *Quaternion* und *Color*, sowie Strings, Enums und Structs. Des Weiteren können Referenzen zu Klassen und Objekten durch Drag-and-Drop im Editor gesetzt werden. Dabei ist zu beachten, dass diese Objekte vom Typ *UnityEngine.Object* ableiten müssen. Es können also ausschließlich Komponenten oder Spielobjekte referenziert werden. Aufgrund dieser Limitierung ist es nicht möglich, Interfaces im Editor zugänglich zu machen. Mit dem *SerializeField*-Attribut lässt sich somit keine lose Kopplung umsetzen.

Neben Variablen mit dem *SerializeField*-Attribut werden auch Variablen, die mit dem Zugriffsmodifizierer *public* gekennzeichnet wurden im Editor zugänglich gemacht. Dies ist jedoch ausdrücklich nicht zu empfehlen, da so die Datenkapselung des Objektes gebrochen wird.

Ist bereits vor Ausführung des Spiels bekannt, welche Referenzen benötigt werden, sollten diese, wenn möglich, über *SerializeField* aufgelöst werden. Um sicherzugehen, dass die Referenz im Editor aufgelöst wurde, ist es ratsam, einen Guard Clause in die *OnValidate*-Funktion der Komponente hinzuzufügen. In Abbildung 3.4 wird dies anhand eines einfachen Beispiels verdeutlicht.

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.Assertions;

public class SomeBehaviour : MonoBehaviour
{
    [SerializeField]
    private Image myImage;

    private void OnValidate()
    {
        Assert.IsNotNull(myImage);
    }
}
```

Abb. 3.4: SerializeField mit Guard Clause in OnValidate. Quelle: Eigene Abbildung

Wenn erst zur Laufzeit bestimmt werden kann, welches das benötigte Objekt ist, zum Beispiel, wenn sich die Abhängigkeit außerhalb eines instanziierten Prefabs befindet, so muss auf eine andere Methode der Referenzierung zurückgegriffen werden.

GetComponent

```
public Component GetComponent(Type type);
public T GetComponent();
```

Abb. 3.5: Signaturen der GetComponent-Methoden. Quelle: Eigene Abbildung

GetComponent ermöglicht es, auf andere Komponenten desselben GameObjects zuzugreifen. Abbildung 3.5 zeigt die Signaturen, der in der Klasse *MonoBehavior* zur Verfügung stehenden *GetComponent*-Methoden. Ist keine Komponente des gewünschten Typs vorhanden, wird *null* zurückgegeben. Dies kann verhindert werden, indem man dem Script das Attribut *RequireComponent* hinzufügt. *RequireComponent* bewirkt, dass die benötigte Komponente automatisch an das GameObject angefügt wird. Um auf Komponenten von anderen GameObjects in der über- und untergeordneten Hierarchieebene zugreifen zu können, werden zusätzlich die in Abbildung 3.6 gezeigten Methoden bereitgestellt.

```
public Component GetComponentInChildren(Type t);
public Component GetComponentInParent(Type t);
```

Abb. 3.6: Signaturen der GetComponentInChildren-Methoden. Quelle: Eigene Abbildung

Alle *GetComponent*-Methoden spezifizieren einen Typparameter *T* als Suchkriterium (siehe Abb. 3.6). Da *T* nicht weiter – zum Beispiel auf *UnityEngine.Object* – beschränkt wird, ist es möglich, anstatt konkreten Komponenten auch Komponenten anhand eines Interfaces zu finden. Enge Kopplung zu einer konkreten Implementation einer Komponente entsteht so nicht. Dennoch ist *GetComponent* keine optimale Lösung: Die Komponente selbst kümmert sich um die Auflösung ihrer Abhängigkeiten. *GetComponent* funktioniert somit entgegengesetzt zum Hollywood-Prinzip. Es ist von außen nicht sichtbar, dass die Komponente Abhängigkeiten hat. Ohne die Implementation der Komponente zu kennen, kann keine Aussage darüber getroffen werden, welche Abhängigkeiten notwendig sind, damit sie funktioniert. Die Komponente kann nur in Verbindung mit anderen Komponenten als GameObject getestet werden.

Beim Verwenden von *GetComponent* wird implizit eine Anordnung in der Objekthierarchie festgelegt. Wird zum Beispiel eine benötigte Komponente in ein Child-GameObject verschoben, ist das Spiel ohne Änderung im Programmcode nicht mehr lauffähig. Aus diesem Grund sollte GetComponent nur in Ausnahmefällen, wenn beispielsweise eine Komponente zur Laufzeit hinzugefügt wird, verwendet werden.

Stringbasierte Find-Funktionen

```
public static GameObject Find(string name);
public static GameObject FindWithTag(string tag);
```

Abb. 3.7: Signaturen der stringbasierten Find-Funktionen. Quelle: Eigene Abbildung

Unitys *Find*-Funktionen sind klassische Game Object Queries zur Inter-Objekt Kommunikation. Das gewünschte Spielobjekt wird auf Basis eines Strings als Suchkriterium zurückgegeben (siehe Abb. 3.7). Dieser String kann entweder der Name oder ein Tag sein. Nachdem das gesuchte Objekt gefunden wurde, muss die gewünschte Komponente mittels *GetComponent* ausfindig gemacht werden. Die Funktionen unterscheiden sich im Verhalten, falls kein passendes GameObject gefunden wird: Wird kein Objekt mit einem passenden Namen gefunden, gibt *Find null* zurück. Wird kein Objekt mit einem passenden Tag gefunden, wirft *FindWithTag* eine *UnityException*.

Generell ist von der Nutzung beider Funktionen abzuraten, da diese nicht gewährleisten, dass das gesuchte GameObject tatsächlich auffindbar ist. Es wird sich darauf verlassen,

dass ein GameObject mit bestimmtem Namen oder Tag in der Szene existiert und die gewünschten Komponenten an diesem vorhanden sind. Es gibt somit viel Raum für Fehlkonfiguration.

FindObjectOfType

```
public static Object FindObjectOfType(Type type);
public static T FindObjectOfType<T>() where T : Object;
```

Abb. 3.8: Signaturen der FindObjectOfType-Methoden. Quelle: Eigene Abbildung

Im Gegensatz zu den anderen *Find* Functions ist *FindObjectOfType* typsicher (siehe Abb. 3.8). Es wird gewährt, dass eine in der Szene aktive Instanz eines bestimmten Typs erhalten wird. Anzumerken ist, dass der gewünschte Typ von *UnityEngine.Object* abgeleitet sein muss. Somit funktioniert auch *FindObjectOfType* nicht mit Interfaces. Die generische Variante von *FindObjectOfType* ist anderen *Find*-Funktionen vorzuziehen, da sie direkt die Instanz der gesuchten Komponente vom Typ *T* als Rückgabewert liefert. Es ist also nicht nötig *GetComponent* zu nutzen.

In der Dokumentation der Funktion wird angemerkt, dass *FindObjectOfType* sehr langsam ist. Sie sollte daher nicht in jedem Frame aufgerufen werden. Als Alternative wird das Singleton Pattern vorgeschlagen (vgl. Unity Technologies, 2015b).

Singleton und Statics

Singletons stellen sicher, dass es nur eine Instanz einer Klasse gibt, auf welche ein globaler Zugriffspunkt existiert. Singletons werden hier mit aufgeführt, weil sie in Unity3D alternativ zu *FindObjectOfType* genutzt werden können, mit der Beschränkung, dass sie auf eine Instanz limitiert sind. Abbildung 3.9 auf der nächsten Seite zeigt eine mögliche Implementation eines Singletons, welches von *MonoBehaviour* ableitet. Die Singletonkomponente muss lediglich einem Spielobjekt in der Szene hinzugefügt werden, um es global verfügbar zu machen. Der Zugriff auf die Singletonkomponente erfolgt über die statische Variable *Instance*.

```
using UnityEngine;

public class Singleton : MonoBehaviour
{
    public static Singleton Instance { get; private set; }

    public void Awake ()
    {
        if (Instance == null)
        {
            Instance = this;
            DontDestroyOnLoad (this);
        }
        else
        {
            Destroy (gameObject);
        }
    }
}
```

Abb. 3.9: Implementation des Singleton Patterns in Form eines MonoBehaviours. Quelle: Eigene Abbildung

Die abgebildete Implementation des Singletons erlaubt es, Funktionalitäten eines MonoBehaviours, wie dem Update Loop, zu nutzen. Sollten diese Funktionalitäten nicht notwendig sein, kann auch ein klassischer Singleton nach der Gang Of Four verwendet werden (siehe Gamma u. a., 1994, S. 127). Ist gar keine Vererbung notwendig, gibt es keinen Grund ein Singleton zu implementieren. Stattdessen kann eine statische Klasse verwendet werden.

An dieser Stelle sei anzumerken, dass Singletons aus vielen Gründen als Anti-Pattern angesehen werden. Eine der Ursachen dafür ist, dass sie einen globalen Zustand einführen. Gleiches gilt für statische Klassen. Globaler Zustand führt dazu, dass Klassen nicht in Isolation testbar sind. Wird Zustand von einem Testfall auf einen anderen übertragen, kann dieser das Verhalten des Tests manipulieren.

Eine Klasse die einen Singleton verwendet, verschleiert ihre Abhängigkeit zu diesem. Von außerhalb ist nicht ersichtlich, dass die Klasse überhaupt eine Abhängigkeit hat. Damit ist der Singleton eng mit der Klasse gekoppelt und nicht austauschbar. In Unittests kann eine solche Abhängigkeit nicht durch ein Test Double ausgetauscht werden.

Singletons verletzen das Single Responsibility Principle: Zusätzlich zu ihrer Anwendungslogik erhält eine Klasse die Verantwortung, sicherzustellen, dass lediglich eine einzige Instanz von ihr existiert (vgl. Densmore, 2004).

3.3.2 Events und Messages

Alternativ kann die Kommunikation zwischen Objekten auch über Events erfolgen. Events können im Unterschied zu Methodenaufrufen keinen oder mehrere Empfänger haben. Grundlage für Events ist das Observer Pattern. Die Klasse, die das Event auslöst, wird als Sender und die Klassen, die das Event behandeln als Empfänger, bezeichnet. Ein typischer Anwendungsfall für Events sind UI-Elemente wie Buttons, Slider und Toggles. Nichtsdestotrotz kann mit Events das gleiche Ergebnis wie mit Referenzierung und Methodenaufruf erzielt werden. Unity3D und C# bieten mehrere Möglichkeiten um die Kommunikation mittels Events umzusetzen. In dieser Arbeit wird die Implementation von C# Events nicht aufgeführt, da sie ohne Mediation oder Statics nicht das Problem der Referenzierung lösen. Wird ein gewöhnliches C# Event als Kommunikationskanal gewählt, so muss der Empfänger über einen der im letzten Abschnitt beschriebenen Wege Zugriff auf den Sender erhalten. Um diese Limitierung zu umgehen, bietet Unity3D ab der Version 4.6 eine eigene Implementierung von Events.

UnityEvent (Unity3D 4.6+)

UnityEvents ermöglichen das Setzen von Callbacks über das Editorfenster. Auf diese Weise sind Sender und Empfänger komplett voneinander entkoppelt. Darüber hinaus bieten UnityEvents zusätzlich nahezu identische Funktionalität wie klassische C# Events.

Die Verwendung eines nicht-generischen UnityEvents ohne Parameter wird in Abbildung 3.10 auf der nächsten Seite gezeigt. Die *Publisher* Komponente löst das Event in der *Start*-Methode aus. Ein Empfänger benötigt lediglich eine öffentliche Methode mit der passenden Signatur.

```
using UnityEngine;
using UnityEngine.Events;

public class EventPublisher : MonoBehaviour
{
    public UnityEvent myEvent;

    private void Start()
    {
        if (myEvent != null)
        {
            myEvent.Invoke();
        }
    }
}

public class EventSubscriber : MonoBehaviour
{
    public void Receive()
    {
        Debug.Log("Recieved");
    }
}
```

Abb. 3.10: Beispiel eines Senders und Empfängers unter Verwendung eines UnityEvents. Quelle: Eigene Abbildung

Auffällig ist, dass bis jetzt noch keine Verbindung zwischen Sender und Empfänger existiert. Diese kann im Editor gesetzt werden (siehe Abbildung 3.11 auf der nächsten Seite). Somit wird es ohne weiteren Programmcode möglich, zu bestimmen wer auf ein Event reagieren soll.

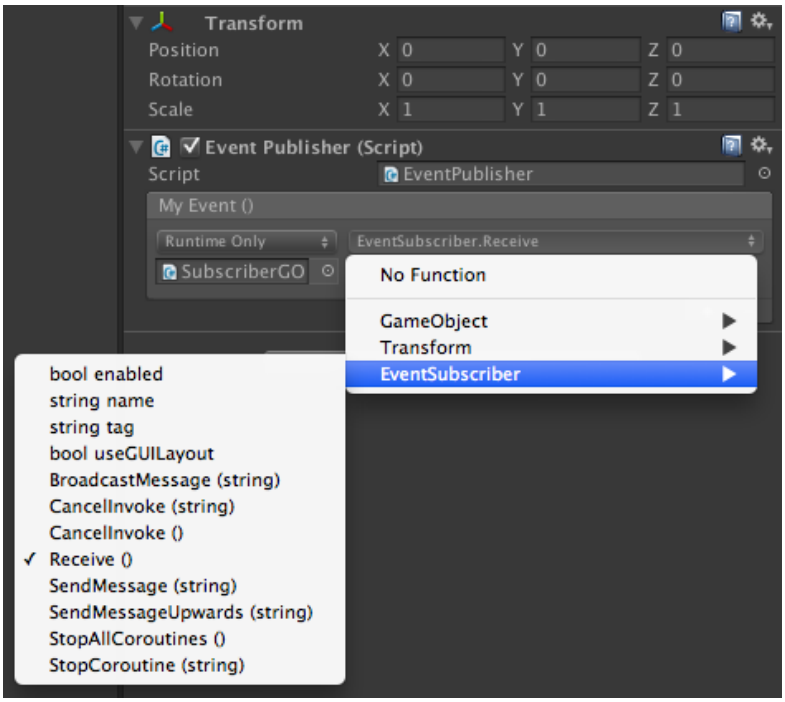

Abb. 3.11: Hinzufügen einer Callback-Methode für einen UnityEvent im Editor. Quelle: Eigene Abbildung

UnityEvents bieten eine Möglichkeit, Sender von Empfänger komplett voneinander zu entkoppeln. Allerdings besteht ebenfalls ein hohes Risiko einer Fehlkonfiguration. Es ist nicht im Programmcode ersichtlich, ob Empfänger existieren und wer diese sind. Genauso wenig wird deutlich, ob eine als öffentlich gekennzeichnete Methode als Callback genutzt wird oder nicht. Oft werden Methoden und Events, auf die keine Referenzen im Programmcode existieren, einfach gelöscht.

Events sind dafür geeignet optionale Abhängigkeiten aufzulösen. Ist eine Abhängigkeit jedoch zwingend erforderlich, sollte zu einer anderen Möglichkeit gegriffen werden.

Senden von Messages

```
public void SendMessage(string methodName, SendMessageOptions
    options);
public void BroadcastMessage(string methodName, SendMessageOptions
    options);
public void SendMessageUpwards(string methodName, SendMessageOptions
    options);
```

Abb. 3.12: Methoden der Klasse MonoBehaviour zur Message-Kommunikation. Quelle: Eigene Abbildung

MonoBehaviours bieten die Funktionalität, untereinander mittels Messages zu kommunizieren (siehe Abb. 3.12). *SendMessage* ermöglicht die Kommunikation zwischen

Komponenten innerhalb eines Spielobjekts. *BroadcastMessage* sendet eine Message an das aufrufende Spielobjekt und dessen Kindobjekte. *SendMessageUpwards* sendet die Nachricht an alle Spielobjekte, die sich dem Spielobjekt übergeordnet in der Szene-Hierarchie befinden.

Abbildung 3.13 zeigt die Verwendung von *SendMessage* anhand von zwei Komponenten, die sich am gleichen Spielobjekt befinden. Der Sender einer Message spezifiziert die Callbackmethode mittels eines Stringparameters. Dieser String repräsentiert den Namen der gewünschten Callbackmethode. Der Zugriffsmodifizierer der Callbackmethode ist dabei nicht relevant. Sie kann, wie im Beispiel gezeigt, auch als *private* deklariert werden. Durch den Aufzählungstyp *SendMessageOptions* ist es möglich zu spezifizieren, ob ein Empfänger zwingend erforderlich ist. Wenn dieser als erforderlich gekennzeichnet wurde und dennoch kein Empfänger gefunden wird, wirft *SendMessage* eine Ausnahme zur Laufzeit.

```
using UnityEngine;

public class MessagePublisher : MonoBehaviour
{
    private void Start()
    {
        SendMessage("Receive");
    }
}

public class MessageSubscriber : MonoBehaviour
{
    private void Receive()
    {
        Debug.Log("Received");
    }
}
```

Abb. 3.13: Verwendung von SendMessage. Quelle: Eigene Abbildung

SendMessage verwendet „magic strings“ zur Identifikation der Callbackmethode. Refactorings der Methodennamen gefährden die Funktionalität der Anwendung. Auch hier ist es nicht ersichtlich, ob eine ansonsten nicht verwendete private Methode als Callback genutzt wird. Nach dem YAGNI-Prinzip werden nicht verwendete private Methoden einfach gelöscht. *SendMessage* legt implizit eine Anordnung in der Hierarchie fest, indem Aufrufe spezifisch an Parent- oder Childelemente erfolgen müssen.

Aus diesen Gründen ist der Nutzung von SendMessage generell abzuraten. Besonders, da mit Unity Version 4.6 ein neues Messaging System als Ersatz für SendMessage vorgestellt wurde (vgl. Unity Technologies, 2015d).

UnityEngine.EventSystems (Unity3D 4.6+)

```
public static bool Execute(GameObject target,
    EventSystems.BaseEventData eventData, EventFunction<T> functor);
public static GameObject ExecuteHierarchy(GameObject root,
    EventSystems.BaseEventData eventData, EventFunction<T>
    callbackFunction);
```

Abb. 3.14: Methodensignaturen des Eventsystems. Quelle: Eigene Abbildung

Das Eventsystem wurde in Verbindung mit uGUI in Unity3D 4.6 eingeführt. Es wird dazu verwendet, um Events bei Nutzereingaben zu senden. Man kann es jedoch auch für benutzerdefinierte Events verwenden. Anstelle eines Strings wird ein Interface verwendet, um eine Komponente zu kennzeichnen, die eine Callbackfunktion für ein bestimmtes Event implementiert. Events können mittels *ExecuteHierarchy* in der Objekthierarchie nach oben, jedoch nicht nach unten, gesendet werden (siehe Abb. 3.14).

Ein Beispiel zur Verwendung des Eventsystems wird in Abbildung 3.15 gezeigt. Der Sender löst ein Event durch den Aufruf der *Execute*-Methode der Klasse *ExecuteEvents* in der *Start*-Methode aus. Die Angabe der Callbackmethode erfolgt als Parameter in Form einer Lambdafunktion. Es wird die *Recieve*-Methode des Empfängers aufgerufen. Hierzu muss der Empfänger das *IEventTarget*-Interface implementieren.

```
using UnityEngine;
using UnityEngine.EventSystems;

public class EventPublisher : MonoBehaviour
{
    private void Start()
    {
        ExecuteEvents.Execute<IEventTarget>(this.gameObject, null,
            (target, data) => target.Receive());
    }
}

public interface IEventTarget : IEventSystemHandler
{
    void Receive();
}

public class EventSubscriber : MonoBehaviour, IEventTarget
{
    public void Receive()
    {
        Debug.Log("Received");
    }
}
```

Abb. 3.15: Verwenden von UnityEngine.EventSystems. Quelle: Eigene Abbildung

Das neue Eventsystem ist ein typsicherer Ansatz Events an Komponenten in der Szenehierarchie zu senden. Sender und Empfänger sind lose miteinander gekoppelt. Bei der Verwendung entsteht jedoch eine Abhängigkeit zu der statischen *ExecuteEvents*-Klasse. Testbarkeit ist somit nur innerhalb von Unity3D gewährleistet. Ohne das Spielobjekt der Empfängerkomponente zu kennen, kann jedoch nur an das eigene oder an direkte Parent-Spielobjekte gesendet werden.

3.4 Wertung und Konsequenzen

Unitys komponentenbasierte Architektur hilft dabei, Klassen mit lediglich einer Verantwortung zu erstellen. Spielobjekte lassen sich durch Komposition aus Komponenten in der Szenehierarchie arrangieren. Um koordinieren zu können, müssen Komponenten miteinander kommunizieren. Unity3D stellt dafür unterschiedliche Wege zur Kommunikation mit anderen Komponenten bereit. Durch Verwendung dieser Kommunikationsmittel entstehen Abhängigkeiten und damit eine Form von Kopplung. Einige Methoden ermöglichen zwar lose Kopplung, aber nicht durch Inversion Of Control. Eines haben jedoch alle Kommunikationsmittel gemeinsam: Die Kommunizierenden müssen Komponenten oder Spielobjekte sein. Dies führt dazu, dass Objekte, um überhaupt kommunizieren zu können, von *MonoBehaviour* ableiten und an Spielobjekte angehängt werden.

Der Architektur von Unity3D fehlt ein Mittel zum Umgang mit geteilter Logik, die keinem einzelnen Spielobjekt zugeordnet werden kann. Es wird sich in der Regel auf einem von zwei Wegen Abhilfe geschaffen:

- Durch die Erstellung von sogenannten Manager- und Controllerobjekten, die an ansonsten „leere“ Spielobjekte angehängt werden.
- Durch Singletons und statischen Klassen, die globalen Zugriff ermöglichen.

Solche Klassen haben oft keine eindeutig definierten Aufgaben. Sie wachsen mit der Zeit und werden somit zu „God Objects“. Managerobjekte verletzen das Single-Responsibility Principle somit gleich auf mehreren Ebenen: Sie haben oft mehr als nur eine Aufgabe. Sie vermischen Darstellungs- mit Anwendungslogik, indem sie als Spielobjekte in der Szene existieren. Des Weiteren verletzen insbesondere Singletons das Single Responsibility Principle, indem sie ihre Erstellung mit anderer Anwendungslogik mischen.

Durch die Referenzierung von Singletons, statischen Klassen oder Managern mittels „Find“-Methoden wird der Rest der Anwendung eng mit diesen gekoppelt. Es wird unmöglich einzelne Komponenten zu testen und in anderen Projekten wiederzuverwenden. Es besteht folglich auch in Unity3D das Bedürfnis nach einer Alternative, die es ermöglicht lose Kopplung umzusetzen, ohne dabei die Kommunikation auf Komponenten oder

Spielobjekte zu beschränken. Im nächsten Kapitel wird beschrieben, wie Dependency Injection in Unity3D umgesetzt werden kann, um diese Ansprüche zu erfüllen.

4 Dependency Injection in Unity3D

Aus dem Fazit des letzten Kapitels geht hervor, dass keiner der in Unity3D zur Verfügung stehenden Kommunikationsmechanismen den gewünschten Ansprüchen entspricht. Das Ziel lose Kopplung in Unity3D umzusetzen, besteht weiterhin. Weiter wurde im letzten Kapitel festgestellt, dass Unity3D Programmierer in vieler Hinsicht einschränkt. Kommunikation ist auf die in der Szene existierenden Spielobjekte und deren Komponenten beschränkt. Es wurde festgestellt, dass die Kontrolle über die Komposition von Objekten bereits invertiert ist. In diesem Kapitel wird beschrieben, wie es dennoch ermöglicht wurde, Dependency Injection in Unity3D umzusetzen.

4.1 Automating Dependency Injection

In Abschnitt 2.5.5 auf Seite 15 wurde veranschaulicht, wie Dependency Injection „per Hand“ angewendet werden kann. Benötigte Instanzen werden manuell im Composition Root instanziiert. In dem gezeigten Beispiel war dies noch sehr überschaubar. In einer echten Anwendung entstehen jedoch schnell komplexe Abhängigkeitshierarchien, die eine manuelle Dependency Injection unübersichtlich und fehleranfällig machen. Aus diesem Grund wird sich oft dazu entschlossen, die Auflösung von Abhängigkeiten zu automatisieren.

In Unity3D ist diese Automatisierung zwangsläufig erforderlich. Die Kontrolle über die Erstellung von Komponenten liegt in den Händen von Unity3D. Es ist nicht möglich, Komponenten über den Konstruktor mit ihren Abhängigkeiten zu versorgen. Es existiert kein klarer Einstiegspunkt in die Anwendung, der es ermöglichen würde, Komponenten von außen mit ihren Abhängigkeiten zu versorgen. Beim Laden einer Szene werden alle sich in ihr befindlichen Spielobjekte instanziiert.

Die Kontrolle über die Komposition von Objekten soll an einen sogenannten Inversion Of Control Container abgegeben werden. Im Composition Root wird vom Nutzer lediglich die Konfiguration des Containers vorgenommen. Die zu instanziierenden Typen werden an diesem Ort beim Container registriert. Der Container kümmert sich um alles Notwendige: Er versorgt alle Klassen mit ihren registrierten konkreten Abhängigkeiten und verwaltet die Lebenszeit aller Objekte. Im Folgenden wird eine Möglichkeit der Implementation eines Inversion Of Control Containers in Unity3D beschrieben. Dabei wird

besonders darauf eingegangen, wo aufgrund von Unitys Architektur Einschränkungen gemacht werden müssen.

4.2 Umsetzung eines Inversion Of Control Containers für Unity3D

Der folgende Abschnitt beschreibt die Umsetzung eines Inversion Of Control Containers für Unity. Als Name für das Framework wurde der Einfachheit halber „Container“ gewählt. Anhand der Implementation des Frameworks werden die mit Inversion Of Control Containern verbundenen Konzepte erläutert. Weiter werden Hürden und Einschränkungen, die aus Unitys Architektur hervorgehen, beschrieben. Es werden Lösungen und Abhilfemaßnahmen für diese präsentiert.

4.2.1 Definieren des Composition Roots

In Unity3D steht kein klarer Einstiegspunkt in die Anwendung zur Verfügung. Es existiert kein Zugriff auf eine *Main*-Methode, in der der Kontrollfluss der Anwendung beginnt. Der frühste Zeitpunkt, an dem der Nutzer Einfluss auf den Kontrollfluss der Anwendung nehmen kann, ist die *Awake*-Methode eines *MonoBehaviours*. Zu diesem Zeitpunkt wurden bereits die in der Szene platzierten Spielobjekte mitsamt ihren Komponenten instanziiert. Soweit nicht anders spezifiziert, erfolgt die Aufrufreihenfolge der *Awake*-Methoden der Komponenten zufällig nach der Reihenfolge, in der sie geladen werden. Es ist möglich in den Script Execution Order Settings anzugeben, dass ein bestimmtes Script früher oder später geladen werden soll (vgl. Unity Technologies, 2015e).

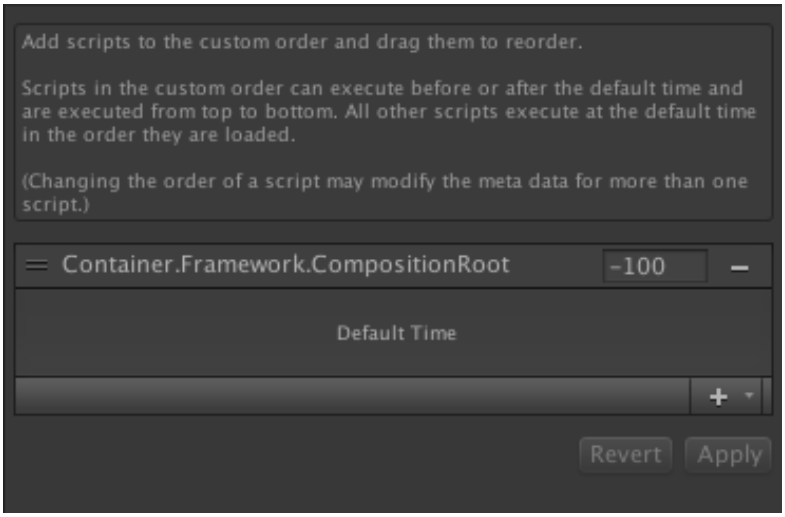

Abb. 4.1: Script Execution Order Settings mit Composition Root Komponente an oberster Stelle. Quelle: Eigene Abbildung

Zur Definition des Composition Roots wird zunächst ein einfaches MonoBehaviour erstellt und einem leeren GameObject in der Szene hinzugefügt. Um sicherzustellen, dass dieses MonoBehaviour auch das erste Script ist, welches bei Anwendungsstart ausgeführt wird, sollte es in den Execution Order Settings an oberster Stelle stehen. Abbildung 4.1 zeigt die korrekte Konfiguration eines Composition Roots in den Script Execution Order Settings. Damit ist gewährleistet, dass der Composition Root Zugriff auf alle sich in der Szene befindlichen Komponenten hat, bevor diese ausgeführt werden. Im Composition Root kann nun die Konfiguration des Inversion Of Control Containers vorgenommen werden und Komponenten mit ihren Abhängigkeiten versorgt werden.

Wie in Abbildung 4.2 zu sehen ist, wurde der Composition Root im „Containter“-Framework als abstrakte Klasse definiert. Die Aufgabe der Klasse ist es zunächst eine Instanz eines *Containers* zu erstellen. Der *Container* bildet das Herzstück des Frameworks. Durch ihn können abstrakte Basisklassen an ihre konkreten Instanzen „gebunden“ werden. Diese Typregistrierung soll in der Methode *SetupBindings* vorgenommen werden. Umgehend danach wird die *Init*-Methode aufgerufen. In ihr kann weitere Initialisierungslogik vorgenommen werden. Beide Methoden sind abstrakt, da die Typregistrierung für jede Applikation individuell erfolgen muss.

```
namespace Container.Framework
{
    public abstract class CompositionRoot : UnityEngine.MonoBehaviour
    {
        protected IContainer container;

        protected virtual void Awake()
        {
            container = new Container();
            SetupBindings();
            Init();
        }

        protected abstract void SetupBindings();

        protected abstract void Init();
    }
}
```

Abb. 4.2: Die Klasse CompositionRoot.cs. Quelle: Eigene Abbildung

Damit ist die Umsetzung des Composition Roots abgeschlossen. Es fehlt jedoch noch die Implementation des Containers an sich. Im Folgenden wird beschrieben, wie dieser im „Container“-Framework umgesetzt wurde.

4.2.2 Aufgaben eines Inversion Of Control Containers

Bevor mit der Implementierung des Containers begonnen werden kann, müssen zunächst dessen grundlegende Aufgaben erläutert werden. Die Hauptaufgabe eines Inversion Of Control Containers ist die Erstellung von Objekten. Hierfür muss dem Container bekannt sein, wie der Objekt Graph der Anwendung aufzulösen ist. Dem Nutzer des Containers muss es folglich ermöglicht werden, ihn zu konfigurieren: Er muss auswählen können, welche konkreten Implementationen einer Abstraktion „injiziert“ werden sollen. Die *Register*-Methoden bilden diese Funktionalität ab.

Mit der Aufgabe der Erstellung von Objekten geht eine weitere Aufgabe einher: die Kontrolle über die Lebensdauer der Objekte. Der Container muss bestimmen, wann eine Instanz eines Objektes erstellt werden soll und wann diese nicht mehr gebraucht wird. Hierfür ist es für den Container wichtig den Sichtbarkeitsbereich („scope“) eines Objektes zu kennen. Im „Container“-Framework wird zwischen zwei unterschiedlichen Sichtbarkeitsbereichen unterschieden: Transient und Singleton.

Register Transient

Wird eine Klasse beim Container mit dem Sichtbarkeitsbereich Transient registriert, so wird dieser bei jeder Anfrage zur Auflösung eine neue Instanz des Typs liefern. Instanzen können somit unabhängig voneinander Zustand halten. Aus diesem Grund ist Transient die sicherste Variante der Sichtbarkeitsbereiche. Sollte es beispielsweise Zweifel an der Threadsicherheit einer Klasse geben, kann diese als Transient registriert werden. Die Möglichkeit Klassen als Transient zu registrieren, soll vom Container durch die Methode *RegisterTransient* implementiert werden.

Register Singleton

Im Gegensatz zu Transient wird bei dem Sichtbarkeitsbereich Singleton für jede Anfrage zur Auflösung eines Typs dieselbe Instanz geliefert. Zur Registrierung eines Objektes mit Singleton Sichtbarkeitsbereich soll die Methode *RegisterSingleton* dienen. Genauso wie beim Singleton Entwurfsmuster kann die Instanz entweder „eager“ oder „lazy“ instanziiert werden. Erfolgt die Instanziierung direkt bei der Registrierung, ist diese „eager“, erfolgt sie erst bei Injektion, ist sie „lazy“.

Der Singleton Sichtbarkeitsbereich sollte nicht mit dem Singleton Entwurfsmuster verwechselt werden: Ein Container liefert immer die selbe Instanz eines Typs, jedoch verwendet er im Gegensatz zum Singleton Entwurfsmuster hierzu keine statische Membervariable. Der Sichtbarkeitsbereich eines als Singleton registrierten Typs ist der des

Containers an dem er registriert wurde. Durch die Definition von mehr als einem Composition Roots, ist es möglich mehrere Container zu verwenden. Somit ergibt sich auch die Möglichkeit, dass verschiedene Container eine unterschiedliche Instanz liefern.

Resolve

Analog zu den *Register*-Methoden liefert die Methode *Resolve* eine zuvor registrierte Instanz eines Typs. Mit dieser Instanz soll auch ihr gesamter Abhängigkeitsgraph instanziiert werden. In den meisten Fällen genügt ein einmaliger Aufruf der *Resolve*-Methode für das Root-Objekt des Object Graphs, um die komplette Anwendung zu initialisieren.

Release

Üblicherweise gehört zu der Verwaltung der Lebensdauer eines Objekts zu entscheiden, was mit Objekten geschieht, die sich außerhalb des Sichtbarkeitsbereiches befinden. Mit dem Aufruf einer *Release*-Methode soll der Speicherbereich eines nicht mehr benötigten Objekts korrekt freigegeben werden. In .NET ist dies jedoch in den meisten Fällen nicht nötig, da ein Objekt, welches nicht mehr referenziert wird, vom Garbage-Collector aus dem Speicher entfernt wird. Aus diesem Grund wurde beim „Container“-Framework auf die Implementation einer *Release*-Methode verzichtet.

Das Interface des Containers

```
public interface IContainer
{
    void RegisterTransient<TInter, TClass>() where TClass : class,
        TInter;
    void RegisterSingleton<TInter, TClass>(TClass instance = null)
        where TClass : class, TInter;
    T Resolve<T>();
}
```

Abb. 4.3: Minimales IContainer-Interface. Quelle: Eigene Abbildung

Aus den beschriebenen Aufgaben ergibt sich das in Abbildung 4.3 gezeigte Interface des zu implementierenden Dependency Injection Containers. Im nächsten Abschnitt wird die konkrete Implementation der Klasse *Container* beschrieben.

4.2.3 Implementation des Containers

Die Implementation beginnt mit der Erstellung der Klasse *Container*, die das IContainer Interface implementiert. Um eine grundlegende Typregistrierung vornehmen zu können, müssen die Zuordnungen abgespeichert werden.

Registrieren und Speichern von Typzuordnungen

```
public class Container : IContainer
{
    private readonly IDictionary<Type, Type> transientMap = new
        Dictionary<Type, Type>();
    private readonly IDictionary<Type, object> singletonMap = new
        Dictionary<Type, object>();

    public void RegisterTransient<TInter, TClass>() where TClass :
        class, TInter
    {
        transientMap[typeof(TInter)] = typeof(TClass);
    }

    public void RegisterSingleton<TInter, TClass>(TClass instance =
        null) where TClass : class, TInter
    {
        //Eager instantiation of singleton objects
        if (instance == null)
        {
            instance = (TClass)Instantiate(typeof(TClass));
        }

        singletonMap[typeof(TInter)] = instance;
    }

    private object Instantiate(Type type)
    {
        //...
    }

    //...
}
```

Abb. 4.4: Implementation der Register-Methoden in Container.cs. Quelle: Eigene Abbildung

In Abbildung 4.4 wird die Implementation der beiden *Register*-Methoden in *Container.cs* gezeigt. Um die Registrierung abzuspeichern, werden zwei *Dictionaries* als Datenstrukturen verwendet. Hierzu werden in beiden Methoden die Typen einer Abstraktion als Schlüssel verwendet. Der Unterschied zwischen Transient und Singleton besteht in der Speicherung der Datenwerte: Bei der Registrierung als Transient wird der Typ der konkreten Klasse als Wert in der *transientMap* abgespeichert. Bei der Registrierung

eines Singletons wird dahingegen eine konkrete Instanz der Klasse gespeichert. Diese Instanz kann als Parameter übergeben werden. Aus der Abbildung geht hervor, dass es nicht zwingend notwendig ist eine Instanz bereitzustellen. Für diesen Fall wurde sich dazu entschieden, direkt eine „eager Instantiation“ vorzunehmen. Auf die Implementation der *Instantiate*-Methode wird zu einem späteren Zeitpunkt innerhalb dieses Abschnitts eingegangen.

Implementation der Resolve-Methode

Wie in Abbildung 4.5 zu sehen ist, delegiert die generische *Resolve*-Methode ihren Aufruf an eine nicht-generische Version. Diese prüft zunächst, ob eine Instanz in der Singleton-Registrierung vorliegt. Bei Vorhandensein wird diese zurückgegeben. Ist der Typ jedoch als Transient registriert, wird eine neue Instanz des in der *transientMap* registrierten, konkreten Typs erstellt und zurückgegeben.

```
public T Resolve<T>()
{
    var type = typeof(T);
    return (T)Resolve(type);
}

private object Resolve(Type type)
{
    object instance;

    if (singletonMap.ContainsKey(type))
    {
        instance = singletonMap[type];
    }
    else if (transientMap.ContainsKey(type))
    {
        instance = Instantiate(transientMap[type]);
    }
    else
    {
        throw new Exception("Couldn't resolve binding for " + type);
    }

    return instance;
}
```

Abb. 4.5: Implementation der Resolve-Methode. Quelle: Eigene Abbildung

Instanziieren von Klassen basierend auf Typinformationen

Die tatsächliche Erstellung von Objekten übernimmt die in Abbildung 4.6 gezeigte *Instantiate*-Methode der Klasse *Container*. Unter Vorgabe eines konkreten Typs wird eine Instanz dieses Typs geliefert. Die Methode delegiert diese Verantwortung an eine

Methode namens *CreateInstance* der Klasse *Activator*. Es sind hierfür noch weitere Informationen über die Klasse notwendig. Dies ist zum einen der zu verwendende Konstruktor der Klasse mitsamt seinen Parametern und zum Anderen Properties, in die Abhängigkeiten injiziert werden sollen. Über die *GetConstructor*-Methode erhält der Container Informationen über den Konstruktor, den er zur Erstellung des Objektes nutzen soll. Dieser kann vom Nutzer des Frameworks spezifiziert werden. Um den Ablauf genauer beschreiben zu können, ist es Voraussetzung zwei Sprachfeatures von C#, die diese Funktionalität ermöglichen, vorzustellen: Attributes und Reflection.

```
private object Instantiate(Type type)
{
    object instance;

    var constructor = GetConstructor(type);
    if (constructor != null)
    {
        var paramInfos = constructor.GetParameters();
        var paramInstances = ResolveParameters(paramInfos);
        instance = Activator.CreateInstance(type, paramInstances);
    }
    else
    {
        //No constructor defined, use the default constructor
        instance = Activator.CreateInstance(type);
    }

    InjectProperties(instance);
    return instance;
}
```

Abb. 4.6: Instantiate-Methode in Container.cs. Quelle: Eigene Abbildung

Attributes

Mittels Attributes lassen sich Metadaten für Programmcodeelemente definieren. Diese Metadaten sind Informationen, die von der Anwendung zur Laufzeit ausgewertet werden können. Es können praktisch alle Elemente des Programmcodes mit Metadaten versehen werden: Assemblys, Klassen, Methoden, Eigenschaften usw. (vgl. Microsoft Corporation, 2015a). Ein Beispiel hierfür ist das bereits in Abschnitt 3.3.1 auf Seite 25 gezeigte *SerializeField*-Attribut. Die Abbildung 4.7 zeigt eine Membervariable, welcher das *SerializeField*-Attribut zugewiesen wurde.

```
[SerializeField]
private Text sampleWithAttribute;
```

Abb. 4.7: Beispiel für die Verwendung eines Attributes. Quelle: Eigene Abbildung

Für den Inversion Of Control Container sind zwei Metadaten einer Klasse relevant: Ihm muss bekannt sein, welcher Konstruktor zur Instanziierung eines Objektes verwendet werden soll und in welche Properties Abhängigkeiten injiziert werden sollen. Es ist ausreichend ein einziges benutzerdefiniertes Attribut zu erstellen, um beide Funktionalitäten abzubilden: Bei der späteren Auswertung des Attributs kann geprüft werden, ob sich dieses an einem Konstruktor oder einem Property befindet und dementsprechend gehandelt werden. In Abbildung 4.8 wird die Erstellung des hierfür vorgesehenen *InjectAttribute*-Attributes dargestellt.

```
[AttributeUsage(AttributeTargets.Property |
   AttributeTargets.Constructor)]
public class InjectAttribute : Attribute
{
}
```

Abb. 4.8: Definition des Inject-Attributs. Quelle: Eigene Abbildung

Es wird zunächst eine Klasse definiert, die von der Klasse *Attribute* ableitet. Um die Gültigkeit des Attributs festzulegen, wird die Klasse selbst mit einem *AttributeUsage*-Attribut gekennzeichnet. Im Fall des *Inject*-Attributs wird die Gültigkeit auf Properties und Konstruktoren beschränkt. Bei der Verwendung entspricht der Attributname dem Klassennamen des Attributs (vgl. Microsoft Corporation, 2015c). Die Endung Attribute kann weggelassen werden. Aus Abbildung 4.9 können die gültigen Verwendungsmöglichkeiten des *InjectAttribute*-Attributs entnommen werden.

```
public class SampleClass
{
    //Alternativ [InjectAttribute]
    [Inject]
    public ISomeProperty property { get; set; }

    [Inject]
    public SampleClass(ISomeDependency first, IAnotherDependency
       second)
    {
        //...
    }
}
```

Abb. 4.9: Verwendungsmöglichkeiten des Inject-Attributes. Quelle: Eigene Abbildung

Reflection

Der Zugriff auf die mit dem *Inject*-Attribut gekennzeichneten Attribute erfolgt mittels Reflection. Reflection ermöglicht das Auslesen und Auswerten von Metadaten eines Assemblys oder Typs zur Laufzeit. Metadaten werden über die *System.Type*-Klasse

abgebildet (vgl. Microsoft Corporation, 2015b). Diese Klasse wurde bereits an einigen Stellen zur Implementation des Containers verwendet. Es wurde also genau genommen bereits Reflection eingesetzt.

Bei der Implementation des Containers wird an zwei Stellen Zugriff auf die Typinformationen des zu instanziierenden Objekts benötigt: Als Erstes muss mittels Reflection herausgefunden werden, welcher Konstruktor zur Instanziierung genutzt werden soll. Diese Funktionalität wird von der Methode *GetConstructor* in Abbildung 4.10 implementiert.

```
private ConstructorInfo GetConstructor(Type type)
{
    var constructors = type.GetConstructors();
    ConstructorInfo constructor;

    if (constructors.Length == 1)
    {
        constructor = constructors[0];
    }
    else
    {
        constructor = constructors.Single(x =>
            Attribute.IsDefined(x, typeof(InjectAttribute)));
    }

    return constructor;
}
```

Abb. 4.10: Zugriff auf Typinformationen mittels Reflektion in der Methode GetConstructor. Quelle: Eigene Abbildung

Die Methode *GetConstructors* der Klasse *Type* liefert Informationen zu allen definierten Konstruktoren einer Klasse. Um den gewünschten, mit dem *Inject*-Attribut gekennzeichneten Konstruktor zu erhalten, wird über die vorhandenen Konstruktoren iteriert und dieser mithilfe der statischen Methode *IsDefined* der Klasse *Attribute* identifiziert. Existiert nur ein einziger Konstruktor für einen Typ, muss dieser nicht explizit mit dem *Inject*-Attribut gekennzeichnet werden, um vom Container zu Instanziierung genutzt zu werden.

Betrachtet man die *Instantiate*-Methode in Abbildung 4.6 auf Seite 44 ein weiteres Mal, stellt man fest, dass bevor die tatsächliche Instanziierung über den Konstruktor vorgenommen wird, zunächst alle Parameter instanziiert werden müssen. Diese Aufgabe übernimmt die Methode *ResolveParameters* in Abbildung 4.11 auf der nächsten Seite. Es wird über die Parameterinformationen des Konstruktors iteriert und alle Parameter der Reihe nach instanziiert. Dies geschieht über einen rekursiven Aufruf der *Resolve*-Methode. Somit ist es Voraussetzung, dass alle Typen der Konstruktorparameter zuvor bei dem Container registriert wurden.

```
private object[] ResolveParameters(ParameterInfo[] parameterInfos)
{
    var parameters = new object[parameterInfos.Length];

    for (int i = 0; i < parameters.Length; i++)
    {
        var parameterInfo = parameterInfos[i];
        parameters[i] = Resolve(parameterInfo.ParameterType);
    }

    return parameters;
}
```

Abb. 4.11: Auflösen der Konstruktorparameter durch die ResolveParameters-Methode. Quelle: Eigene Abbildung

Nach der Instanziierung eines Typs ist es notwendig, die Klasse mit Abhängigkeiten, die in Form von Properties vorliegen, zu versorgen. Die Methode *InjectProperties*, gezeigt in der Abbildung 4.12, bildet diese Funktionalität des Containers ab. Es werden Informationen zu Properties mit dem *Inject*-Attribut über die Methode *GetProperties* der Klasse *Type* geliefert. Im nächsten Schritt wird über die *PropertyInfos* iteriert und mittels Aufruf der *Resolve*-Methode eine Instanz des Properties geholt. Diese Instanz wird danach dem Property des Objektes mithilfe *SetValue*-Methode zugewiesen.

```
public void InjectProperties(object instance)
{
    var type = instance.GetType();
    var propertyInfos = type.GetProperties().Where(prop =>
       Attribute.IsDefined(prop,
       typeof(InjectAttribute))).GetEnumerator();

    while (propertyInfos.MoveNext())
    {
        var propertyInfo = propertyInfos.Current;
        var propertyInstance = Resolve(propertyInfo.PropertyType);
        propertyInfo.SetValue(instance, propertyInstance, null);
    }
}
```

Abb. 4.12: Auflösen von Properties durch die InjectProperties-Methode. Quelle: Eigene Abbildung

Damit ist die grundlegende Implementation des Containers abgeschlossen. Zur optimalen Zusammenarbeit mit Unity3D sollten jedoch noch einige Erweiterungen vorgenommen werden. Diese Erweiterungen werden im folgenden Abschnitt beschrieben.

4.2.4 Erweiterungen des Containers für die Zusammenarbeit mit Unity3D

Zur Gewährleistung besserer Zusammenarbeit des Containers mit Unity3D, sollten noch Erweiterungen an diesem vorgenommen werden. Es gibt beispielsweise derzeit keine Möglichkeit Abhängigkeiten in MonoBehaviours zu injizieren. Die Umsetzungen dieser Erweiterungen werden in diesem Abschnitt präsentiert. Es wird verdeutlicht, wo Hürden und Einschränkungen bei der Umsetzung von Dependency Injection in Unity3D vorhanden sind.

Injections in MonoBehaviours

Wie bereits in den vorigen Abschnitten festgestellt wurde, übernimmt Unity3D selbst die Verantwortung über die Erstellung von MonoBehaviours. Es existiert folglich kein Konstruktor über den Abhängigkeiten aufgelöst werden können. Es muss sich auf Property Injection beschränkt werden. Dabei ist zu berücksichtigen, dass MonoBehaviours bereits in der Szene existieren, bevor die *Awake*-Methode des Composition Roots aufgerufen wird. Des Weiteren können GameObjects und MonoBehaviours auch zur Laufzeit instanziiert werden. In beiden Fällen muss es ermöglicht werden, dass diese ihre Abhängigkeiten erhalten. Im Folgenden werden drei Ansätze hierfür erläutert: das Verwenden des Containers als Service Locator, Factories und ExecuteEvents als MonoBehaviour-Extension.

Service Locator

Eine mögliche Lösung für das Problem ist die Verwendung des Containers als Service Locator. Bei einem Service Locator handelt es sich um einen statischen Wrapper für den Container. So wird in Abbildung 4.13 eine mögliche Implementation des Service Locator Patterns unter Verwendung der *Container*-Klasse gezeigt.

```
public static class ServiceLocator
{
    private static readonly IContainer container = new Container();

    public static void RegisterTransient<TInter, TClass>() where
        TClass : class, TInter
    {
        container.RegisterTransient<TInter, TClass>();
    }

    public static void RegisterSingleton<TInter, TClass>(TClass
        instance = null) where TClass : class, TInter
    {
        container.RegisterSingleton<TInter, TClass>(instance);
    }

    public static T Resolve<T>()
    {
        return container.Resolve<T>();
    }
}
```

Abb. 4.13: Implementation des Service Locators. Quelle: Eigene Abbildung

Die Konfiguration des Service Locators würde, wie die des Containers, in der zuvor definierten *CompositionRoot*-Klasse stattfinden. Der Unterschied zwischen einem Service Locator und einem Inversion Of Control Framework liegt nicht in der Implementation, sondern in der Verwendung. Die Verwendung der *Resolve*-Methode des Containers beschränkt sich üblicherweise auf einen einzigen Aufruf im Composition Root und setzt somit die rekursive Auflösung des gesamten Object Graphs in Gang. Wird ein Service Locator verwendet, können Abhängigkeiten von überall im Programm durch Aufruf der *Resolve*-Methode angefordert werden. MonoBehaviours würden also in der *Awake*-Methode durch Anfrage beim Service Locator ihre Abhängigkeiten erhalten.

Zunächst wird der Eindruck vermittelt, dass keine Vorteile bei der Verwendung des Service Locators verloren gehen: Abhängigkeiten sind lose gekoppelt, die Programmierung im Parallelen ist gewährleistet und Testbarkeit ist durch Konfiguration des Service Locators in der Testumgebung möglich. Trotzdem wird der Service Locator oft als Anti-Pattern angesehen. Klassen haben eine unnötige konkrete Abhängigkeit an den Service Locator. Die Klasse wird explizit für eine Verwendung mit dem Service Locator implementiert. Von außen ist es nicht sichtbar, dass die Klasse andere Abhängigkeiten hat. Es kann keine Aussage darüber getroffen werden, welche Abhängigkeiten notwendig sind, damit eine Klasse funktioniert.

Nichtsdestotrotz ermöglicht der Service Locator die Auflösung von Abhängigkeiten eines MonoBehaviours. Die Verwendung des Service Locators kann auf MonoBehaviours reduziert werden und in diesem Fall als notwendig angesehen werden. Das Problem ist jedoch, dass der Service Locator, aufgrund der globalen Sichtbarkeit als statische

Klasse, von unwissenden Programmierern auch in anderen Bereichen genutzt werden kann. Im Folgenden sollen zwei weitere Möglichkeiten vorgestellt werden, mit denen sich Abhängigkeiten eines MonoBehaviours auflösen lassen. Alle haben ihre Vor- und Nachteile. Der Service Locator ist die einfachste, aber zugleich auch die am wenigsten elegante Lösung des Problems.

„Inject“-Event als MonoBehaviour-Extension

Die Grundidee einer anderen Methode ist, dem Container über ein Event Bescheid zu geben, dass ein MonoBehaviour in der Szene erstellt wurde und dessen Abhängigkeiten aufgelöst werden müssen. Hierzu wird die Klasse MonoBehaviour mit der in Abbildung 4.14 gezeigten Extension erweitert.

```
public static class MonoInjectionExtension
{
    public static void Inject(this MonoBehaviour script)
    {
        ExecuteEvents.ExecuteHierarchy<IMonoInjectionHandler>
            (script.gameObject, null, (target, data) =>
               target.InjectDependencies(script));
    }
}
```

Abb. 4.14: Erweiterungsmethode für MonoBehaviours. Quelle: Eigene Abbildung

Mittels der *ExecuteHierarchy*-Methode der Klasse *ExecuteEvents* wird ein Event in der Hierarchie nach oben gesendet. Das Event soll vom Composition Root entgegengenommen werden, damit dieser die Auflösung der Abhängigkeiten an den Container delegieren kann. Dazu muss *CompositionRoot* das für diesen Fall erstellte *IMonoInjectionHandler*-Interface implementieren. Die Abbildung 4.15 zeigt das Interface *IMonoInjectionHandler* und dessen Implementation vom Composition Root.

```
public interface IMonoInjectionHandler : IEventSystemHandler
{
    void InjectDependencies(MonoBehaviour script);
}

public abstract class CompositionRoot : IMonoInjectionHandler
{
    //...

    public void InjectDependencies(MonoBehaviour script)
    {
        container.InjectProperties(script);
    }
}
```

Abb. 4.15: Implementation von IMonoInjectionHandler. Quelle: Eigene Abbildung

Bei dieser Methode ist zu beachten, dass alle Spielobjekte im Szenengraph Children des Composition Roots sein müssen. Jedes MonoBehaviour muss im *Start*-Event explizit die *Inject*-Erweiterungsmethode aufrufen, damit die Abhängigkeiten vom Container aufgelöst werden. Dies könnte bereits im *Awake*-Event geschehen, da durch das Festlegen der Script Execution Order in Abschnitt 4.2.1 gewährleistet wird, dass der Composition Root den Container bereits initialisiert hat. Instanziiert man jedoch ein Spielobjekt zur Laufzeit, so muss der Composition Root unmittelbar darauf als Parentelement festgelegt werden. Es ist nicht möglich ein Spielobjekt direkt als Child eines anderen Spielobjekts zu instanziieren. Zum Zeitpunkt des Aufrufs der *Awake*-Methode hat noch kein Parenting stattgefunden, auch wenn dies unmittelbar nach der Instanziierung geschieht. Demzufolge sollte die *Inject*-Methode im *Start*-Event des MonoBehaviours aufgerufen werden.

Bei der Verwendung der beschriebenen Methode ist es nötig, dass der Nutzer des Containers sich strikt an die beschriebenen Anweisungen hält. Ein Aufruf der *Inject*-Methode muss im *Start*-Event erfolgen, da ansonsten keine Abhängigkeiten injiziert werden. Alle Spielobjekte müssen Children des Composition Roots sein. Nachfolgend soll eine letzte Möglichkeit vorgestellt werden, die dem Nutzer weniger Verantwortung abverlangt.

Auflösung des Szenegraphs und Factories

Bei dieser Methode wird die Auflösung von Abhängigkeiten von MonoBehaviours in zwei Teile unterteilt. Durch den ersten Teil werden bereits in der Szene existierende MonoBehaviours mit ihren Abhängigkeiten versorgt. Dies wird bewerkstelligt indem dem Composition Root die *ResolveScene*-Methode hinzugefügt wird. Diese Methode wird im *Awake*-Event unmittelbar nach der Konfiguration des Containers aufgerufen. Die Änderungen an der *CompositionRoot*-Klasse werden in Abbildung 4.16 dargestellt.

```
public abstract class CompositionRoot
{
    //...

    protected override void Awake()
    {
        container = new Binder();
        SetupBindings();
        ResolveScene();
        Init();
    }

    private void ResolveScene()
    {
        foreach (var script in
           Object.FindObjectsOfType(typeof(MonoBehaviour)))
        {
            container.InjectProperties(script);
        }
    }
}
```

Abb. 4.16: Hinzugefügte ResolveScene-Methode in CompositionRoot.cs. Quelle: Eigene Abbildung

Um die Abhängigkeiten der existierenden MonoBehaviours aufzulösen, wird über alle MonoBehaviours in der Szene iteriert und dem Container mitgeteilt alle ihre Properties zu instanziieren. Durch das Festlegen der Script Execution Order in Abschnitt 4.2.1 auf Seite 38, wird gewährleistet, dass alle sich in der Szene befindlichen MonoBehaviours ihre Abhängigkeiten bereits vor dem Aufruf der *Awake*-Methode erhalten.

Weiter muss der Fall berücksichtigt werden, dass Spielobjekte dynamisch zur Laufzeit instanziiert werden können. Unmittelbar nach der Instanziierung müssen die Abhängigkeiten aller MonoBehaviours des Spielobjektes vom Container aufgelöst werden. Damit dies sichergestellt ist und an einem zentralen Ort geschieht, wird hierfür eine Factory implementiert. Über diese Factory werden alle Spielobjekte zur Laufzeit erstellt. Das heißt, dass es im Programmcode keine weiteren Aufrufe der *Instantiate*-Methode der Klasse *Object* geben darf. Der einzige Aufruf der *Instantiate*-Methode findet in der *Create*-Methode der *GameObjectFactory*-Klasse wie in Abbildung 4.17 dargestellt, statt.

```
public class GameObjectFactory : IGameObjectFactory
{
    //...

    public GameObject Create(GameObject original)
    {
        var instance = Object.Instantiate(original);

        foreach (var script in
            instance.GetComponents<MonoBehaviour>())
        {
            binder.InjectProperties(script);
        }

        return instance;
    }

    //...
}
```

Abb. 4.17: Ausschnitt GameObjectFactory.cs. Quelle: Eigene Abbildung

Direkt nach dem Erstellen eines MonoBehaviours über die Factory, werden dessen Properties vom Container instanziiert. Es ist sichergestellt, dass MonoBehaviours zum Start-Event über alle Abhängigkeiten verfügen. Die *GameObjectFactory-Klasse* bietet eine weitere Überladung der *Create*-Methode, die aus Platzgründen nicht dargestellt wurde. Mit ihr lassen sich Prefabs anhand ihres Namens im Resource-Ordner instanziieren. Spielobjekte, die aus dem Resource-Ordner geladen wurden, werden in einem Cache zwischengespeichert.

Bei Verwendung dieser Methode muss der Nutzer des „Container“-Frameworks keine frameworkspezifischen Operationen innerhalb eines MonoBehaviours vornehmen. Dem Nutzer muss lediglich bewusst sein, dass er die Klasse *GameObjectFactory* zur Instanziierung von Spielobjekten zur Laufzeit verwenden muss. Es ist nicht nötig, dass Spielobjekte Children des Composition Roots sind, um ihre Abhängigkeiten zu erhalten. Aus diesem Grund ist aus Sicht des Autors der vorliegenden Arbeit diese Methode den Anderen vorzuziehen.

Injecting MonoBehaviours

Die bis jetzt gezeigten Methoden beschreiben, wie Abhängigkeiten in MonoBehaviours injiziert werden können. An dieser Stelle wird beschrieben, wie mit Abhängigkeiten zu anderen MonoBehaviours umgegangen werden soll. Zunächst sollte überprüft werden, ob es tatsächlich notwendig ist, dass die benötigte Klasse von MonoBehaviour erbt. Durch strikte Trennung der Belange („Seperation Of Concerns“) kann dies meist auf die Darstellungslogik reduziert werden. Die benötigten Komponenten sind oft Unity-Eigene

Klassen wie *Button*, *Label* oder *RigidBody* am selben Spielobjekt. In diesen Fällen können sie mithilfe des *SerializeField*-Attributs im Editor gesetzt werden.

Liegen dennoch Gründe vor, dass eine Abhängigkeit zu einem MonoBehaviour benötigt wird, könnte dieses theoretisch beim Container registriert werden. Vorausgesetzt es implementiert ein Interface, welches die benötigte Funktionalität abbildet. Das Spielobjekt des MonoBehaviours kann mittels *Find*-Methode vom Composition Root aus ausfindig gemacht und im Anschluss beim Container registriert werden. Die Registration ist somit jedoch auf eine einzige Instanz eines Typs beschränkt. Hiervon wird jedoch ausdrücklich abgeraten. Anhand eines Beispiels soll im Folgenden ein alternativer Ansatz beschrieben werden.

Die Grundidee dieses Ansatzes ist es, nicht ein MonoBehaviour, sondern einen Wrapper für dieses MonoBehaviour beim Container zu registrieren. Eine Funktionalität eines MonoBehaviours, welche auch ausserhalb von diesem benötigt werden könnte, ist das Ausführen von Coroutines (vgl. Unity Technologies, 2015a). In Abbildung 4.18 wird die Verwendung dieses Ansatzes bei der Umsetzung der Funktionalität Coroutines auszuführen, gezeigt.

```
public interface ICoroutineRunner
{
    Coroutine StartCoroutine(IEnumerator routine);
}

public class CoroutineRunnerBehaviour : MonoBehaviour,
   ICoroutineRunner
{
    private void Awake()
    {
        DontDestroyOnLoad(transform.gameObject);
    }
}

public class CoroutineRunner : ICoroutineRunner
{
    private readonly IGameObjectFactory gameObjectFactory;
    private readonly ICoroutineRunner coroutineRunnerBehaviour;

    public CoroutineRunner(IGameObjectFactory gameObjectFactory)
    {
        this.gameObjectFactory = gameObjectFactory;

        var go =
           this.gameObjectFactory.Create("CoroutineRunnerPrefab");
        coroutineRunnerBehaviour =
           go.GetComponent<ICoroutineRunner>();
    }

    Coroutine StartCoroutine(IEnumerator routine)
    {
        return coroutineRunnerBehaviour.StartCoroutine(routine);
    }
}
```

Abb. 4.18: Implementation des CoroutineRunners. Quelle: Eigene Abbildung

Die benötigte Funktionalität wird zunächst durch die *ICoroutineRunner*-Schnittstelle definiert. Die Implementation erfolgt von der Klasse *CoroutineRunnerBehaviour,* indem sie von *MonoBehaviour* erbt. Weiter wird die Schnittstelle auch von der Klasse *CoroutineRunner* implementiert. Im Konstruktor erstellt sie, durch den Aufruf der *Create*-Methode der *GameObjectFactory* ein Prefab, welches die *CoroutineRunnerBehaviour*-Komponente besitzt. Den Aufruf der *StartCoroutine*-Methode delegiert die Klasse an diese *CoroutineRunnerBehaviour*-Komponente. Die Klasse *CoroutineRunner* kann ohne Weiteres im Composition Root beim Container registriert werden:

container.RegisterTransient<ICoroutineRunner, CoroutineRunner>();

Die Klasse kann nun in alle anderen Module injiziert und genutzt werden. Damit ist eine grundlegende Zusammenarbeit zwischen Unity3D und dem Inversion Of Control Container gewährleistet. Um die gezeigten Beispiele auf das Wesentliche zu reduzieren, wurde auf die Darstellung und Implementierung einiger Funktionalitäten verzichtet.

Diese werden jedoch im folgenden Abschnitt kurz erwähnt und beschrieben.

4.2.5 Weitere Ansatzpunkte zur Erweiterung

Bei der dargestellten Implementierung des Inversion Of Control Containers handelt es sich um ein „Proof Of Concept“. Die Funktionsweise, sowie nötige Sprachfeatures sollten erläutert und eine Zusammenarbeit mit Unity3D gewährleistet werden. Auf eine ausführliche Beschreibung einiger Details wurde verzichtet. So wird beispielsweise das „eager“- und „lazy-loading“ von Singletons in der tatsächlichen Implementierung des „Container“-Frameworks unterstützt. Die Implementation ist auf der beigelegten CD zu dieser Arbeit zu finden.

Überdies hinaus gibt es zahlreiche Ansatzpunkte zur Erweiterung des Frameworks:

- Die Verwendung von Reflection führt zu starken Performanceeinbußen. Obwohl sich die Aufrufe der Reflection-Bibliotheken zum Großteil auf den Programmstart beschränken, sollten diese reduziert werden. Die Ergebnisse der Reflection eines Typs sollten zwischengespeichert („cached“) und vom Container wiederverwendet werden.
- Das Verhalten bei zirkulären Abhängigkeiten ist nicht bestimmt. Im besten Fall sollten diese gar nicht erst zugelassen werden. Es sollte eine Ausnahme zur Laufzeit geworfen werden. Eine zirkuläre Abhängigkeit existiert, wenn eine Klasse *A* eine Klasse *B* referenziert und diese eine Klasse *C* referenziert, die wiederum eine Referenz auf Klasse *A* hat.
- Die API zur Konfiguration/Registrierung des Containers kann verbessert und erweitert werden. Es können „fluent interfaces“ zur Erstellung einer domänenspezifischen Sprache verwendet werden. Eine weitere Konfigurationsmöglichkeit könnte über JSON oder XML geboten werden. Der Vorteil einer solchen externen Konfiguration ist, dass das Verhalten der Anwendung ohne erneuter Kompilierung geändert werden kann.
- Bis jetzt wurde die Verwendung des Containers auf eine einzige Szene beschränkt. Auf das Verhalten bei der Arbeit mit mehreren Szenen wurde nicht eingegangen. Jede Szene könnte einen eigenen Composition Root haben, welcher die nötigen Abhängigkeiten der Szenen individuell auflöst. Alternativ könnte ein einziger Container in der „Haupt“-Szene die Abhängigkeiten in allen Szenen auflösen. Dies ist sinnvoll, wenn alle Szenen gleiche Abhängigkeiten haben. Werden Szenen additiv geladen, wäre auch die Erstellung einer Containerhierarchie eine Möglichkeit. Beim Root-Container werden Abhängigkeiten, die über mehrere Szenen hinweg benötigt werden registriert, während Child-Container spezifische Abhängigkeiten der additiv geladenen Szenen auflösen.

- Der Container sollte für eine ordnungsgemäße Entsorgung von Objekten sorgen. Im Speziellen sollte sichergestellt werden, dass die *Dispose*-Methode von Klassen, die *IDisposable* implementieren, aufgerufen wird.
- Es ist derzeit nicht möglich, Logik unmittelbar nach der Injection auszuführen. In Abbildung 4.18 wird diese Beschränkung umgangen, indem weitere Logik im Konstruktor ausgeführt wird. Die Gefahr hierbei ist jedoch, dass Properties zu diesem Zeitpunkt noch nicht inijziert wurden. Desweiteren sollte das Single Responsibility Principle gewahrt werden, indem im Konstruktor lediglich Initialisierungslogik ausgeführt wird. Aus diesem Grund sollte ein weiteres Attribut namens *PostConstruct* vorgestellt werden: Methoden, die mit dem *PostConstruct*-Attribut versehen wurden, sollen unmittelbar nach dem Injection-Prozess vom Container aufgerufen werden.

Aus diesen Gründen wird von dem Autor dieser Arbeit davon abgeraten, den präsentierten Container zur Entwicklung eines Spiels zu verwenden. In dem folgenden Abschnitt werden ausgereiftere Inversion Of Control Frameworks für Unity3D vorgestellt und auf ihre Tauglichkeit bei der Entwicklung von Spielen untersucht.

4.3 Übersicht Inversion Of Control Frameworks für Unity3D

In den vorherigen Abschnitten dieser Arbeit wurden Prinzipien, Praktiken und Funktionsweise von Dependency Injection und Inversion Of Control Containern beschrieben. Dieser Abschnitt soll eine Übersicht über die für Unity3D verfügbaren Inversion Of Control Container geben. Ihre Funktionsumfänge und Besonderheiten sollen beschrieben werden. Oft bieten diese Frameworks Funktionalitäten, die weit über automatisierte Dependency Injection hinausgehen.

Die vorzustellenden Container wurden speziell für die Verwendung mit Unity3D entwickelt. Ein Grund hierfür ist, dass – wie im vorherigen Abschnitt ausführlich beschrieben – einige Erweiterungen notwendig sind, um die Verwendung eines Containers in Unity3D zu ermöglichen. Ein weiterer Grund ist, dass Unitys Mono Laufzeitumgebung lediglich Kompatibilität zum .NET Framework 3.5 aufweist. Die meisten existierenden Inversion Of Control Container für .NET verwenden .NET 4.0 oder höher.

Das Grundkonzept für die Implementation eines Inversion Of Control Containers für Unity3D lieferte Sebastiano Mandalà im September 2012 in einer Serie von Blogeinträgen. Auf dieser Basis haben sich mit der Zeit eine Reihe von Frameworks entwickelt. Im Folgenden sollen die Frameworks Zenject, adic und StrangeIoC vorgestellt werden (siehe Mandalà, 2012).

Zenject

Zenject ist ein Inversion Of Control Container, der sich aus einem direkten Fork des Konzepts von Sebastiano Mandalà entwickelt hat. Zusätzlich zu den grundlegenden Funktionalitäten eines Inversion Of Control Containers bietet das Framework eine Vielzahl von Registrierungsoptionen, wie zum Beispiel die Registrierung von Prefabs und GameObjects oder Injection in spezifische Methoden und Getter. Darüber hinaus bietet das Framework eine Möglichkeit zur Validierung und Visualisierung des Object Graphs: Es kann überprüft werden, dass alle zu injizierenden Typen im Object Graph einer Szene beim Container registriert sind. Es ist möglich, eine an UML angelegte graphische Darstellung des Object Graphs der Anwendung zu generieren (vgl. Modest Tree Media, 2015).

Adic

Adic ist ein weiterer Dependency Injection Container für Unity3D, der sich vor allem durch seine Einfachheit in Umsetzung und Verwendung auszeichnet. Dennoch bietet er die vollständige Funktionalität eines Inversion Of Control Containers und noch einige weitere Optionen. Diese sind unter anderem: Conditional Bindings, die es ermöglichen eine Registrierung eines Typs unter bestimmten Bedingungen vorzunehmen. Eine mögliche Bedingung ist beispielsweise, dass Injection eines Typs nur über Konstruktoren oder in bestimmte Instanzen erfolgen soll. Zudem implementiert Adic ein Command- und Eventsystem, das sehr ähnlich zu dem von StrangeIoC ist (vgl. Martins, 2015). Diese werden im folgenden Abschnitt genauer erläutert.

StrangeIoC

StrangeIoC ist ein umfangreiches Inversion Of Control Framework, das weit mehr als nur einen Dependency Injection Container bietet. Das Open-Source Projekt verfügt über eine aktive Community, die bemüht ist, das Framework kontinuierlich zu verbessern und zu erweitern. StrangeIoC ist in großen Teilen von Robotlegs, einem Application Architecture Framework für ActionScript, beeinflusst. Aus diesem wurde auch das wohl augenscheinlichste Merkmal des Frameworks übernommen: Eine MVC(S)-Architektur, mit der sich ein breites Spektrum von Anwendungen umsetzen lassen soll. Die Architektur vereint eine umfangreiche Anzahl von Konzepten, welche in ihrer Gesamtheit eine saubere Trennung von Belangen ermöglichen sollen. Diese Konzepte und ihr Zusammenwirken als MVC(S)-Architektur sollen im folgenden Abschnitt erläutert werden (vgl. Third Motion, Inc., 2015a).

4.4 MVC(S)-Architektur in Unity3D mithilfe von StrangeIoC

Model-View-Controller-Service, oder kurz MVC(S), ist ein Architekturmuster, welches die Struktur einer Anwendung in vier eigenständige Module unterteilt: Model, View, Controller und Service. Es erweitert die klassische MVC-Architektur um eine Serviceschicht, unter welche der Zugriff auf externe Daten und Dienste, wie Webserver, Datenbanksysteme oder Zugriffe auf das Dateisystem, fallen.

MVC(S) soll einen konsistenten Ansatz zur Strukturierung einer Anwendung liefern. Genauso wie bei der klassischen MVC-Architektur ist die Grundlage hierfür Präsentations- von Anwendungslogik zu trennen. Diese Trennung erfolgt durch die Definition folgender Schichten: Die Präsentationsschicht (View-Layer) ist für die Darstellung der Benutzeroberfläche und der Verarbeitung von Interaktionen mit dem Benutzer zuständig. Die Modellschicht (Model-Layer) speichert den aktuellen Zustand der Anwendung, welcher durch das Ausführen von Operationen durch den Nutzer und externer Quellen, resultiert. Die Steuerungsschicht (Controller-Layer) stellt Mechanismen zur Interaktion und Koordination zwischen den anderen Schichten zur Verfügung.

Zusätzlich zu dem Vorteil der Strukturierung hilft die MVC(S)-Architektur von StrangeIoC dabei, die Kopplung zwischen den einzelnen Schichten zu minimieren. Damit sollen einzelne Programmabschnitte testbar und die Austauschbarkeit der Darstellungs- und Serviceschicht gewährleistet werden. Abhängigkeiten zu externen APIs können in der Serviceschicht gekapselt und somit austauschbar gemacht werden. Somit ist es beispielsweise ohne große Umstände möglich, den Zugriff auf Daten aus dem lokalen Dateisystem durch den Remotezugriff auf eine Datenbank auszutauschen.

Die Umsetzung der MVC(S)-Architektur weicht stark von der des klassischen MVC Entwurfsmusters ab. StrangeIoC verwendet eine Reihe von eigenen Konzepten und Mustern, um die MVC(S)-Architektur umzusetzen (vgl. Third Motion, Inc., 2015b).

4.4.1 Umsetzung von MVC(S) in StrangeIoC

In diesem Abschnitt wird erklärt, wie die in StrangeIoC zur Verfügung stehenden Konzepte in der MVC(S)-Architektur zusammenspielen. Eine schlüssige Zusammenfassung der Architektur liefert ein Schaubild aus der Dokumentation (siehe Abbildung A.1 im Anhang). Wie in der Grafik beschrieben wird, ist der Einstiegspunkt in die Anwendung der ContextView. Von ihm wird der eigentliche Context instanziiert.

Context und ContextView

Der Context ist ein von den restlichen Schichten der Architektur unabhängiges Modul. Es handelt sich um einen Composition Root wie in Abschnitt 4.2.1 auf Seite 38 definiert, mit dem Unterschied, dass es möglich ist, mehr als einen Context zu definieren. Jeder Context wird an einen ContextView, ein MonoBehaviour in einer bestimmten Szene, angehängt. Im Context wird, wie im Composition Root, die Registrierung von Typen beim Container vorgenommen. Der Context definiert den Sichtbarkeitsbereich aller bei seinem Container registrierten Typen. Zudem ist es möglich Registrierungen über mehrere Contexte hinweg vorzunehmen. StrangeIoC bezeichnet den Container als Binder und die Registrierung als Binding. Der Grund hierfür ist, dass nicht nur zu injizierende Abhängigkeiten registriert werden können. Es können auch – wie später noch genauer erläutert wird – Commands an Signals und Views an Mediators gebunden werden.

```
public class SampleContext : MVCSContext
{
    //...

    protected override void mapBindings()
    {
        base.mapBindings();

        injectionBinder.Bind<IBoardLayoutModel>()
            .To<BoardLayoutModel>().ToSingleton();
        injectionBinder.Bind<IMovementRules>().To<MovementRules>();
        injectionBinder.Bind<FieldClickedSignal>().ToSingleton();

        commandBinder.Bind<StartSignal>().InSequence()
            .To<LoadBoardLayoutCommand>()
            .To<CreateBoardFieldViewsCommand>();
        commandBinder.Bind<FieldClickedSignal>()
            .To<ToggleSelectionCommand>();

        mediationBinder.Bind<FieldView>().To<FieldMediator>();
    }

    public override IContext Start()
    {
        base.Start();
        var startSignal = injectionBinder.GetInstance<StartSignal>();
        startSignal.Dispatch();
        return this;
    }
}
```

Abb. 4.19: Beispiel eines Contexts für ein Brettspiel. Quelle: Eigene Abbildung

Ein Beispiel eines Contexts wird in Abbildung 4.19 gezeigt: In der *mapBindings*-Methode erfolgen die Typregistrierungen bei den jeweiligen Bindern. Abhängigkeiten werden

beim *InjectionBinder* registriert, Commands werden über den *CommandBinder* an Signale gebunden und Views an Mediatoren durch den *MediationBinder*. Diese Registrierungen werden immer komplexer mit dem Wachstum der Anwendung. Aus diesem Grund können sie auch in einzelne Klassen ausgelagert werden. Ist die Registrierung abgeschlossen, wird die Anwendung durch das Auslösen eines Signals in der *Start*-Methode gestartet. An dieses Startsignal wurden zuvor einige Commands gebunden, die nun ausgeführt werden.

Signals

Signals stellen einen typsicheren Weg zur Kommunikation zwischen den einzelnen Architekturschichten dar. Signals können Commands auslösen, aber auch wie herkömmliche Events abonniert werden. Es handelt sich um eine Umsetzung des Observer-Musters, das intern *Action*-Delegaten anstatt Interfaces verwendet. Das Auslösen eines Signals erfolgt über die *Dispatch*-Methode. Wurde das Signal beim CommandBinder registriert, so fügt sich dieser als Listener hinzu. Beim Auslösen des Signals instanziiert der CommandBinder Instanzen der hinzugefügten Commands und führt diese aus. Damit sind Sender und Empfänger eines Signals komplett voneinander entkoppelt.

```
//Deklaration
public class FieldClickedSignal : Signal<int>{};

//Verwendung
[Inject]
public FieldClickedSignal clickSignal { get; set; }

clickSignal.Dispatch(12);
```

Abb. 4.20: Beispiel der Deklaration und Verwendung eines Signals. Quelle: Eigene Abbildung

Abbildung 4.20 zeigt die Deklaration und Verwendung eines Signals. Signals können bis zu vier Parameter haben, dessen Typen als Generika bei der Deklaration des Signals definiert werden. Diese Parameter werden bei Aufruf in Properties des Commands injiziert. Signals sind typsicher: Fehlerhaftes Aufrufen oder Abonnieren führt zu einer Ausnahme zur Laufzeit.

Commands

Commands sind zustandslose Controller-Objekte, die ein einzelnes Verhalten kapseln. Commands werden von Signals ausgelöst: Ein Aufruf der *Execute*-Methode erfolgt automatisch. Im Normalfall haben sie eine sehr kurze Lebenszeit: Ein Signal löst

die Instanziierung und Ausführung des Commands aus. Direkt danach erfolgt die Entsorgung. Commands bilden die Logik der Anwendung ab, indem sie Manipulationen an Models vornehmen, Spielobjekte instanziieren oder mit Services kommunizieren.

```
class ToggleSelectionCommand : Command
{
    [Inject]
    public int index { get; set; }

    [Inject]
    public IService service { get; set; }

    public override void Execute()
    {
        Retain();
        service.SendRequestToServer(response => {
            //...
            Release();
        });
    }
}
```

Abb. 4.21: Beispiel für einen Command. Quelle: Eigene Abbildung

In der Abbildung 4.21 wird der an das Signal aus dem vorigen Beispiel gebundene Command, dargestellt. Der beim Auslösen des Signals übergebene Parameter wird bei Instanziierung des Commands in das Property „index“ injiziert. Commands wird es durch Aufruf der *Retain*-Methode ermöglicht, seine Zerstörung zu verzögern. Die Ausführung der Anwendung wird nicht eingeschränkt, da der Aufruf eines Commands asynchron erfolgt. Nützlich ist das Erhalten eines Commands beispielsweise bei der Client-Server-Kommunikation: In dem gezeigten Beispiel in Abbildung 4.21 wird der Command so lange am Leben erhalten, bis dieser eine Antwort vom Server erhält. Dies geschieht durch den Aufruf der *Release*-Methode im Callback der Anfrage.

Commands bieten eine effiziente Möglichkeit Anwendungsschichten voneinander zu entkoppeln. Dies lässt sich damit begründen, dass sie sich nur durch Signale ausführen lassen: Für Models, Services und Mediatoren gibt es keinen Grund überhaupt von ihnen zu wissen. Models und Mediatoren brauchen lediglich Signale auszulösen, um die Ausführung der Anwendung in Gang zu setzen.

Views

Wie bereits erwähnt sind Views für die Darstellung der Benutzeroberfläche und der Verarbeitung von Interaktionen mit dem Benutzer verantwortlich. Dabei ist der Begriff „View“ nicht unbedingt wörtlich zu nehmen: Views übernehmen jegliche Ein- und Ausgabelogik, wie Rendering, Darstellungslogik von User Interfaces, Audioausgabe und

eventuelle Interaktion mit der Physikengine. Es handelt sich um Aufgaben, für die eine Interaktion mit Unity3D notwendig ist. Aus diesem Grund sind Views MonoBehaviours. Sie beinhalten jedoch keinerlei Anwendungslogik und kommunizieren lediglich zu ihren Mediatoren über Events. Ansonsten sind sie komplett autonome Komponenten, die ausschließlich ihren eigenen Zustand manipulieren können. Deswegen sollten keine Injections in MonoBehaviours vorgenommen werden. Die Absicht hierbei ist, das Unity-Framework soweit wie möglich aus der Anwendungslogik rauszuhalten, um eine enge Kopplung zu diesem zu verhindern.

Mediation

Das Mediator-Muster beschreibt einen Weg der Interaktion zwischen Objekten über einen Vermittler (vgl. Gamma u. a., 1994, S. 273). In StrangeIoC vermitteln Mediatoren zwischen View- und Anwendungslogik. Mediatoren interagieren im Auftrag ihres Views mit dem Rest des Frameworks und anders herum. Dies geschieht, indem der Mediator die Events seines Views abbonniert und daraufhin entsprechende Signals auslöst. Umgekehrt abonniert der Mediator auch die Events von Services oder Models und führt daraufin Operationen an seinem View aus.

Durch die Abbildung B.1 auf Seite 78 im Anhang wird veranschaulicht, wie der Mediations-Prozess in StrangeIoC vonstattengeht: Sobald ein View in eine Szene geladen wird, sendet dieser automatisch in der Awake-Methode eine Message an den ContextView. Dafür muss zwangsläufig jedes View ein Childobject eines ContextViews sein. Über den Context des ContextViews wird der MediationBinder aufgerufen. Dieser instanziiert einen zuvor an den View gebundenen Mediator und versorgt ihn mit seinen Abhängigkeiten mithilfe des Injectors.

```
public class SampleMediator : Mediator
{
    [Inject]
    public SampleView view { get; set; }

    [Inject]
    public IBoardLayoutModel layout { get; set; }

    [Inject]
    public FieldClickedSignal clickSignal { get; set; }

    public override void OnRegister()
    {
        view.OnBoardClicked.AddListener(OnClick);
        layout.OnLayoutChanged.AddListener(OnLayoutChange);
    }

    public override void OnRemove()
    {
        view.OnBoardClicked.RemoveListener(OnClick);
        layout.OnLayoutChanged.RemoveListener(OnLayoutChange);
    }

    private void OnClick(Vector2 clickPosition)
    {
        var index = ClickPositionToFieldIndex(clickPosition);
        clickSignal.Dispatch(index);
    }

    private void OnLayoutChange()
    {
        view.Refresh();
    }

    //...
}
```

Abb. 4.22: Beispiel der Implementation eines Mediators. Quelle: Eigene Abbildung

Ein Beispiel für eine mögliche Implementation eines Mediators wird in Abbildung 4.22 aufgeführt. Der Mediator reagiert auf das *OnBoardClicked*-Event des Views, indem er die Klickposition in einen verwendbaren Index konvertiert und daraufhin ein *FieldClickedSignal* mit dem Index als Parameter auslöst. Weiter abonniert der Mediator das *OnLayoutChanged*-Event des *IBoardLayoutModels*. In Reaktion auf dieses Event ruft der Mediator die *Refresh*-Methode des von ihm vermittelten Views auf. Das An- und Abmelden an die Events erfolgt in den *OnRegister*- und *OnRemove*-Callbacks. *OnRegister* wird direkt nach der Injektion von Abhängigkeiten aufgerufen. Der Aufruf von *OnRemove* erfolgt beim Zerstören des Views.

Wie durch das Beispiel deutlich wird, agieren Mediatoren lediglich als Brücke zwischen Darstellungs- und Anwendungslogik. Sie sollten somit leichtgewichtig sein und möglichst keine eigene Logik beinhalten. Sie dienen einzig dazu, Anwendungslogik aus den

Views herauszuhalten. Dafür gibt es zahlreiche Gründe: Darstellungslogik tendiert dazu mit der Zeit unüberschaubar zu werden, da an ihr oft Änderungen in letzter Minute vorgenommen werden. Weiter verwendet die Darstellungslogik eines Views die API von Unity3D, die durch statische Klassen und fehlendem Interfacesupport zu enger Kopplung und damit zu eingeschränkter Testbarkeit führt.

Models und Services

Die Aufgaben eines Models in StrangeIoC unterscheiden sich nicht von denen eines Models in einer klassischen MVC-Architektur. Models kapseln Zustand und Daten der Anwendung und ermöglichen den Zugriff auf diese über wohldefinierte Schnittstellen. Andere Klassen können über diese Schnittstellen Anfragen an das Model senden. Models selber senden keine Anfragen: Wird ein Model geändert, lässt es dies den Rest der Anwendung wissen, indem es ein Event auslöst.

Services stehen eng mit Models in Verbindung. Dennoch sollten sie getrennt voneinander existieren. Ein Service beschafft – oft durch die Kommunikation zur Außenwelt – Daten. Ein Model speichert diese Daten. Services kommunizieren mit externen Ressourcen einer Anwendung. Beispiele für solche Ressourcen sind unter anderem: Webserver und -Services, das Dateisystem und Datenbanken. Der Zugriff aus Services erfolgt über Commands, wie bereits in Abbildung auf Seite 62 gezeigt.

4.4.2 Beispielanwendung

Die im letzten Abschnitt gezeigten Programmcode-Ausschnitte wurden aus einer für diese Arbeit programmierten Beispielanwendung übernommen. Damit der Fokus auf die zu erläuternden Konzepte nicht verloren geht, wurden die Klassen für die Beispiele teilweise gekürzt und abgeändert. Die tatsächliche Implementation der Beispielanwendung ist auf der sich im Anhang dieser Arbeit befindlichen CD zu finden. Es handelt sich um eine Implementation des Brettspiels Peg Solitaire.

4.4.3 Beurteilung der Anwendbarkeit von MVC(S) in der Spieleentwicklung

Wie aus den vorgestellten Konzepten ersichtlich wird, wurde die MVC(S)-Architektur von StrangeIoC speziell mit der Absicht entworfen, Kopplung zu minimieren. Durch die Architektur ist es möglich Daten, Präsentation und Anwendungslogik eines Videospiels weitestgehend voneinander zu entkoppeln. Dennoch kann keine eindeutige Empfehlung zur generellen Verwendung der Architektur in der Spieleentwicklung gegeben werden.

StrangeIoCs Architektur hat ihren Ursprung in der Entwicklung von Rich Internet Applications (vgl. Hooks u. Fallow, 2011, S. IX). Es werden weitaus weniger zeitkritische Operationen als in Videospielen ausgeführt. In besonders performancekritischen Momenten innerhalb des Game Loops sollte deshalb auf eine hohe Anzahl an Injections und Signals verzichtet werden (vgl. Third Motion, Inc., 2015b). Eine Empfehlung ist, MVC(S) als Anwendungsarchitektur auf einer höheren Ebene anzusehen: Es handelt sich um ein Gerüst, das auf die meisten Teile des Programms anwendbar ist. So eignet sich die Architektur beispielsweise dazu, Benutzeroberflächen, Dateizugriffe und Client-Server-Kommunikation umzusetzen. Ob die Architektur auch innerhalb des Game Loop angewendet werden kann, muss für ein Spiel im Speziellen entschieden werden. Sie kann dennoch für die restlichen Programmabschnitte des Spiels verwendet werden.

Die komponentenbasierte Architektur von Unity3D berücksichtigt StrangeIoC nur teilweise: Es ist möglich einzelne Komponenten zu mediieren. Dies würde jedoch zu einer stark erhöhten Komplexität der Anwendung führen. In der Praxis werden Spielobjekte deshalb als ein Ganzes – einem View – angesehen.

5 Kritische Betrachtung & Fazit

Die Spieleengine Unity3D erfreut sich immer größerer Beliebtheit unter Spieleentwicklern. Gründe dafür sind Plattformunabhängigkeit, ausgereifte Rendertechniken, Benutzerfreundlichkeit und der einfache Einstieg in die Entwicklungsumgebung. Mit Wachstum eines Softwareprojektes und Teams werden weitere nichtfunktionale Anforderungen wie Wartbarkeit, Erweiterbarkeit, Wiederverwendbarkeit, Testbarkeit und die Möglichkeit zur parallelen Entwicklung deutlich. Im zweiten Kapitel dieser Arbeit wurde beschrieben, dass sich Kopplung zwischen Entitäten maßgeblich auf diese Anforderungen auswirkt. Kopplung beschreibt die Anzahl und Stärke von Abhängigkeiten unter Softwareentitäten. Eine Abhängigkeit entsteht, sobald eine Entität mit einer anderen kommunizieren muss, um zu funktionieren. Damit wurde das Ziel dieser Arbeit deutlich: Es soll untersucht werden, wie Kopplung zwischen Entitäten bei der Entwicklung von Videospielen in Unity3D minimiert werden kann.

Hierzu wurde zunächst Inversion Of Control als grundlegendes Paradigma präsentiert. Allein Inversion Of Control ermöglicht die Kommunikation zwischen Entitäten, ohne sie dabei eng miteinander zu koppeln. Mit Dependency Injection wurde eine konkrete Umsetzung von Inversion Of Control durch die Umkehr der Kontrolle über die Auflösung von Abhängigkeiten präsentiert. Es handelt sich um eine Reihe von Praktiken und Mustern, die es unter Wahrung der SOLID-Prinzipien ermöglichen, lose Kopplung zu erreichen. Die Vorteile loser Kopplung wurden durch die Umsetzung von Dependency Injection anhand eines Beispiels innerhalb einer Konsolenanwendung deutlich gemacht: Dadurch, dass Abhängigkeiten lediglich in Form von Interfaces vorliegen, ist es nicht notwendig, dass die eigentliche Implementierung überhaupt bekannt sein muss. Wie diese Implementierung tatsächlich aussieht, ergibt sich erst zur Laufzeit durch späte Bindung. Entitäten sind unabhängig voneinander austauschbar und erweiterbar. Durch diese Unabhängigkeit ist eine parallele Entwicklung gewährleistet. Lose Kopplung macht einzelne Entitäten testbar, indem sie in Isolation betrachtet werden können.

5.1 Kopplung in Unity3D

Nachdem die Vorteile und eine mögliche Umsetzung von loser Kopplung durch Inversion Of Control deutlich gemacht wurden, wurde Unity3D auf die Anwendung des Paradigmas geprüft. Zunächst wurde untersucht, inwieweit sich Unitys komponentenbasierte

Architektur von einer klassischen objektorientierten Vererbungshierarchie unterscheidet. Es wurde festgestellt, dass in Unity3D bereits Inversion Of Control stattfindet. Die Kontrolle über die Komposition und Lebenszeit von Objekten ist längst invertiert. Aus diesem Grund können Abhängigkeiten nicht auf herkömmlichen Wegen aufgelöst werden. Als Alternative stellt Unity3D eine Vielzahl von Kommunikationsmechanismen bereit. Diese wurden auf ihre Fähigkeit lose Kopplung umzusetzen untersucht. Es wurde festgestellt, dass diese Kommunikationsmechanismen bestimmten Beschränkungen unterliegen und einige Probleme mit sich bringen. Somit ist es zwar mit einigen Mechanismen, wie z. B. *GetComponent,* möglich lose Kopplung umzusetzen, jedoch ist die Kommunikation auf Objekte innerhalb der Szenenhierarchie beschränkt. Der Architektur von Unity3D fehlt ein Mittel zum Umgang mit geteilter Logik, die keinem einzelnen Spielobjekt zugeordnet werden kann. Das Resultat ist die häufige Verwendung von Anti-Patterns wie Managern, God Objects und Singletons, die zu enger Kopplung führen. Aus diesem Grund wurde die Entscheidung getroffen, Dependency Injection auch in Unity3D umzusetzen, um es so zu ermöglichen lose Kopplung zu realisieren, ohne dabei die Kommunikation auf Komponenten oder Spielobjekte zu beschränken.

5.1.1 Dependency Injection

Aufgrund der bereits beschriebenen Einschränkungen durch Unitys Architektur, ist es nicht möglich Dependency Injection manuell durchzuführen. Aus diesem Grund wurde sich dazu entschieden Dependency Injection durch Implementation eines Inversion Of Control Frameworks zu automatisieren. Dabei gab es einige Probleme zu bewältigen: Es musste zunächst ein „künstlicher“ Composition Root in Form eines MonoBehaviours definiert werden. Weiter wurden Möglichkeiten zu Injektion in MonoBehaviours beschrieben. Da alle Varianten mit Nachteilen behaftet sind, wurde sich nicht explizit für eine entschieden. Anzumerken ist, dass bei allen Methoden Property Injection verwendet werden muss. Bei der Implementation des Frameworks wurde deutlich, dass ein Dependency Injection Container von Attributes und Reflection Gebrauch macht. Da Reflection sehr performanceintensiv ist, sollte in zeitkritischen Momenten auf Dependency Injection verzichtet werden. Eine Funktionalität, die deswegen noch zum Framework hinzugefügt werden sollte, ist das Caching von Reflectionaufrufen. Für eine tatsächliche Verwendung bei der Entwicklung eines Videospiels sollte somit ein ausgereifteres Framework verwendet werden. In dieser Arbeit wurde ein Überblick über die für Unity3D entwickelten Inversion Of Control Frameworks geliefert, um eine Entscheidungshilfe zu bieten.

5.1.2 StrangeIoC

Dependency Injection allein führt noch nicht zu einer guten Softwarearchitektur. Um Kopplung weiter zu reduzieren, muss Inversion Of Control konsequent umgesetzt werden. StrangeIoC, eines der vorgestellten Inversion Of Control Frameworks, bietet hierzu zusätzlich eine individuelle MVC(S)-Architektur. Unterschiedliche Belange einer Anwendung werden in die Architekturschichten Model, View, Controler und Service eingeteilt. Durch die Vereinheitlichung der Kommunikation mittels Signals wird lose Kopplung zwischen den Architekturschichten ermöglicht. StrangeIoC liefert mit der MVC(S)-Architektur einen konsistenten Ansatz zur Strukturierung einer Anwendung. Mit diesem gernerellen Architekturansatz lässt sich ein breites Spektrum von Anwendungen umsetzen. Es gibt jedoch auch Fälle, für die die MVC(S)-Architektur weniger gut geeignet ist. Dies sind unter anderem Videospiele, die viel zeitkritische Logik beinhalten oder stark von der komponentenbasierten Architektur Unitys Gebrauch machen.

5.2 Ausblick und Alternativen

Zusammenfassend kann gesagt werden, dass Dependency Injection auch in Unitys komponentenbasierter Architektur seine Existenzberechtigung hat. Im Gegensatz zu integrierten Kommunikationsmechanismen ist die Kommunikation nicht auf Spielobjekte und Komponenten in der Szene beschränkt. Durch Dependency Injection wird es ermöglicht, lose Kopplung in Unity3D umzusetzen. Inversion Of Control Container und deren Integration in Unity gehen jedoch nicht nur mit Vorteilen einher: Injections können einen negativen Einfluss auf Performance haben. Da der Composition Root selbst ein Spielobjekt in der Szene sein muss, kann er streng genommen mit dem Manager Anti-Pattern in Verbindung gebracht werden. Es handelt sich um eine Einschränkung, die direkt auf die komponentenbasierte Architektur von Unity3D zurückzuführen ist.

Ferner ist es jedoch die Verantwortung der Engine, ein Kommunikationsmittel bereitzustellen, das lose Kopplung zwischen Entitäten ermöglicht. Ein Ansatz hierfür wäre, dass Unity3D von sich aus einen Composition Root oder sogar eine integrierte Form von Dependency Injection zur Verfügung stellt. Eine weitere Alternative wäre der Schritt zu einer Entity Component System Architektur. In einer solchen Architektur beinhalten Komponenten lediglich Daten und müssen somit nicht miteinander kommunizieren. Jegliches Verhalten wird von Systemen abgebildet. Die Engine sorgt dafür, dass ein System Referenzen zu relevanten Komponenten aller Entities erhält. Somit machen auch Entity Component Systems Gebrauch von Inversion Of Control. Indem der Zustand einer Anwendung ausschließlich in Komponenten gehalten wird, gibt es keinen Grund für Systeme untereinander zu kommunizieren. In einer Entity Component System Architektur existiert Kopplung somit lediglich in Form von Daten und Events von Komponenten,

die von mehreren Systemen genutzt werden (vgl. Mandalà, 2015).

Die Ansätze schließen sich nicht gegenseitig aus: Dependency Injection kann auch in einer Entity Component System Architektur angewendet werden, um Abhängigkeiten von Systemen lose mit diesen zu koppeln. Es könnte sogar ein Inversion Of Control Container zur Implementation einer Entity Component System Architektur genutzt werden.

Glossar

Dependency Inversion Principle	High-Level Klassen sollen nicht von Low-Level Klassen abhängig sein, sondern beide von Interfaces. Interfaces sollen nicht von Details abhängig sein, sondern Details von Interfaces
Interface Segregation Principle	Interfaces sollen nur die Funktionalität wiederspiegeln die ihre Klienten erwarten.
JSON	Die JavaScript Object Notation ist ein textbasiertes Datenformat zum Datenaustausch zwischen Anwendungen.
Liskov Subsitituion Principle	Abgeleite Klassen sollen sich so verhalten, wie der Basistyp es von ihnen erwartet.
Mock	Ein Test Double, welches prüfen kann, ob tatsächlich Interaktion mit diesem stattgefunden hat.
MVC(S)	Kurzform für Model, View, Controler, Service. Siehe Kapitel 4.4
Open/Closed Principle	Eine Klasse soll offen für Erweiterung, jedoch geschlossen für Modifikation sein.
Prefab	Mithilfe eines Prefabs lassen sich Vorlagen bzw. Prototypen von GameObjects definieren, um zur Laufzeit Instanzen von diesem erzeugen zu können.
Single Responsibility Principle	Eine Klasse soll nur eine Verantwortlichkeit haben.
Strategy Pattern	Das Strategy Pattern ist ein Entwurfsmuster, welches Algorithmen zur Laufzeit austauschbar macht, indem diese von einer gemeinsamen Abstraktion ableiten.
Stub	Ein Test Double, welches bei Aufruf einen vorgegebenen Wert liefert.

System Under Test	Ein System Under Test bezeichnet eine zu testende Entität in einem Testszenario.
Template Method	Die Template Method (Schablonenmethode) ist ein Entwurfsmuster, mit dem sich Teile eines in einer Basisklasse definierten Algorithmus durch Vererbung neu definieren lassen.
Test Double	Test Double ist ein Oberbegriff für Klassen, die in Unittests als Ersatz für reale Abhängigkeiten eingesetzt werden. (Siehe Mock, Stub)
XML	Die Extensible Markup Language ist eine textbasierte Auszeichnungssprache fuŕ hierarchisch strukturierte Daten.
YAGNI	Akronym für “You Aren’t Gonna Need It”. Einem Programm soll keine überflüssige Logik hinzugefügt werden.

Literaturverzeichnis

[Bender u. McWherter 2011] BENDER, James ; MCWHERTER, Jeff: *Professional Test Driven Development with C# - Developing Real World Applications with TDD*. 1. Aufl. New York, NY : John Wiley & Sons, 2011 http://eu.wiley.com/WileyCDA/WileyTitle/productCd-047064320X.html. – ISBN 978–1–118–10210–7

[Butler 2013] BUTLER, Tom: *Constructor Injection vs Setter Injection*. https://r.je/constructor-injection-vs-setter-injection.html. Version: 2013. – [Online; Stand 06. Oktober 2015]

[Cann 2014] CANN, Michael: *Taming Unity*. http://www.mikecann.co.uk/games/taming-unity/. Version: 2014. – [Online; Stand 06. Oktober 2015]

[Dąbrowski 2014] DĄBROWSKI, Kamil: *NamekDev - Why I hate Unity3D popularity*. http://www.namekdev.net/2014/06/why-i-hate-unity3d-popularity/. Version: 2014. – [Online; Stand 06. Oktober 2015]

[Densmore 2004] DENSMORE, Scott: *Why Singletons are Evil*. http://blogs.msdn.com/b/scottdensmore/archive/2004/05/25/140827.aspx. Version: 2004. – [Online; Stand 06. Oktober 2015]

[Fowler 2001] FOWLER, Martin: Reducing Coupling. In: *IEEE Software* 18 (2001), 102 - 104. http://dx.doi.org/10.1109/ms.2001.936226. – DOI 10.1109/ms.2001.936226. – ISSN 0740–7459

[Fowler 2004] FOWLER, Martin: *Inversion of Control Containers and the Dependency Injection pattern*. http://www.martinfowler.com/articles/injection.html. Version: 2004. – [Online; Stand 15. September 2015]

[Fowler 2005] FOWLER, Martin: *Martin Fowler's Bliki: Inversion of control*. http://martinfowler.com/bliki/InversionOfControl.html. Version: 2005. – [Online; Stand 15. September 2015]

[Gamma u. a. 1994] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns - Elements of Reusable Object-Oriented Software*. 1. Aufl. Amsterdam : Pearson Education, 1994 http://www.pearsoned.co.uk/bookshop/detail.asp?item=171742/. – ISBN 978–0–321–70069–8

[Gregory 2014] GREGORY, Jason: *Game Engine Architecture, Second Edition*. 2nd edition. Revised. Boca Raton, FL : CRC Press, 2014 https://www.crcpress.com/Game-Engine-Architecture-Second-Edition/Gregory/9781466560017/. – ISBN 978–1–466–56001–7

[Hooks u. Fallow 2011] HOOKS, Joel ; FALLOW, Lindsey: *ActionScript Developer's Guide to Robotlegs*. 1. Aufl. Sebastopol, CA : O'Reilly Media, Inc., 2011 http://shop.oreilly.com/product/0636920021216.do/. – ISBN 978–1–449–30890–2

[IEEE Computer Society u. a. 2014] IEEE COMPUTER SOCIETY ; BOURQUE, Pierre ; FAIRLEY, Richard E.: *Swebok - Guide to the Software Engineering Body of Knowledge*. 3. Aufl. Washington, DC : IEEE Computer Society Press, 2014 http://www.computer.org/web/swebok/. – ISBN 978–0–769–55166–1

[Johnson u. Foote 1988] JOHNSON, Ralph E. ; FOOTE, Brian: Designing Reusable Classes. In: *Journal of Object-Oriented Programming* 1 (1988), Nr. 2, S. 22–35

[Mandalà 2012] MANDALÀ, Sebastiano: *Seba's Lab - IoC Container for Unity3D*. http://www.sebaslab.com/ioc-container-for-unity3d-part-1/. Version: 2012. – [Online; Stand 06. Oktober 2015]

[Mandalà 2015] MANDALÀ, Sebastiano: *Seba's Lab - Entity Component System Design*. http://goo.gl/xaZrqZ. Version: 2015. – [Online; Stand 06. Oktober 2015]

[Martin 2003] MARTIN, Robert C.: *Agile Software Development - Principles, Patterns, and Practices*. 1. Aufl. Amsterdam : Pearson Education, 2003. – ISBN 978–0–135–97444–5

[Martins 2015] MARTINS, André: *adic - Another Dependency Injection Container for Unity 3D and beyond*. https://github.com/intentor/adic. Version: 2015. – [Online; Stand 06. Oktober 2015]

[Metz 2009] METZ, Sandi: *GORUCO 2009 - SOLID Object-Oriented Design*. http://confreaks.tv/videos/goruco2009-solid-object-oriented-design. Version: 2009. – [Online; Stand 06. Oktober 2015]

[Microsoft Corporation 2015a] MICROSOFT CORPORATION: *Microsoft Developer Network - Attributes*. https://msdn.microsoft.com/en-us/library/z0w1kczw.aspx. Version: 2015. – [Online; Stand 06. Oktober 2015]

[Microsoft Corporation 2015b] MICROSOFT CORPORATION: *Microsoft Developer Network - Reflection*. https://msdn.microsoft.com/en-us/library/ms173183.aspx. Version: 2015. – [Online; Stand 06. Oktober 2015]

[Microsoft Corporation 2015c] MICROSOFT CORPORATION: *Microsoft Developer Network - Viewing Type Information*. https://msdn.microsoft.com/en-us/library/t0cs7xez.aspx. Version: 2015. – [Online; Stand 06. Oktober 2015]

[Modest Tree Media 2015] MODEST TREE MEDIA: *Zenject - Dependency Injection Framework for Unity3D*. https://github.com/modesttree/Zenject. Version: 2015. – [Online; Stand 06. Oktober 2015]

[Nystrom 2014] NYSTROM, Robert: *Game Programming Patterns*. 1. Aufl. United States : Genever Benning, 2014 http://www.bookdepository.com/Game-Programming-Patterns-Robert-Nystrom/9780990582908/. – ISBN 978–0–990–58291–5

[Reddit Community 2015] REDDIT COMMUNITY: *Reddit - Coming from a .Net programming background Unity's API feels like a horror-show*. https://np.reddit.com/r/gamedev/comments/3hr8al/coming_from_a_net_programming_background_unitys/. Version: 2015. – [Online; Stand 06. Oktober 2015]

[Schatten u. a. 2010] SCHATTEN, Alexander ; BIFFL, Stefan ; DEMOLSKY, Markus ; GOSTISCHA-FRANTA, Erik ; ÖSTREICHER, Thomas ; WINKLER, Dietmar: *Best Practice Software-Engineering - Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen*. 1. Aufl. Berlin : Springer-Verlag, 2010. – ISBN 978–3–827–42487–7

[Seemann 2011] SEEMANN, Mark: *Dependency Injection in .NET*. 1. Aufl. Birmingham : Manning, 2011 https://www.manning.com/books/dependency-injection-in-dot-net/. – ISBN 978–1–935–18250–4

[Shore 2006] SHORE, James: *Dependency Injection Demystified*. http://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html. Version: 2006. – [Online; Stand 15. September 2015]

[Suzdalnitski 2015] SUZDALNITSKI, Ilya: *Unity Ninjas - Unity3D Game Code Architecture*. http://www.unityninjas.com/code-architecture/unity3d-game-code-architecture/. Version: 2015. – [Online; Stand 06. Oktober 2015]

[Third Motion, Inc. 2015a] THIRD MOTION, INC.: *StrangeIoC - Main Page*. http://strangeioc.github.io/strangeioc/. Version: 2015. – [Online; Stand 06. Oktober 2015]

[Third Motion, Inc. 2015b] THIRD MOTION, INC.: *StrangeIoC - The Big, Strange How-To*. http://strangeioc.github.io/strangeioc/TheBigStrangeHowTo.html. Version: 2015. – [Online; Stand 06. Oktober 2015]

[Third Motion, Inc. 2015c] THIRD MOTION, INC.: *StrangeIoC - Views and Mediators Deep Dive*. https://strangeioc.wordpress.com/2015/06/06/bsht-views-and-mediators-deep-dive-ready-for-review/. Version: 2015. – [Online; Stand 06. Oktober 2015]

[Unity Technologies 2015a] UNITY TECHNOLOGIES: *Unity Documentation - MonoBehaviour.StartCoroutine*. http://docs.unity3d.com/ScriptReference/MonoBehaviour.StartCoroutine.html. Version: 2015. – [Online; Stand 06. Oktober 2015]

[Unity Technologies 2015b] UNITY TECHNOLOGIES: *Unity Documentation - Object.FindObjectOfType*. http://docs.unity3d.com/ScriptReference/Object.FindObjectOfType.html. Version: 2015. – [Online; Stand 06. Oktober 2015]

[Unity Technologies 2015c] UNITY TECHNOLOGIES: *Unity Manual - Learning the Interface*. http://docs.unity3d.com/Manual/LearningtheInterface.html. Version: 2015. – [Online; Stand 06. Oktober 2015]

[Unity Technologies 2015d] UNITY TECHNOLOGIES: *Unity Manual - Messaging System*. http://docs.unity3d.com/Manual/MessagingSystem.html. Version: 2015. – [Online; Stand 06. Oktober 2015]

[Unity Technologies 2015e] UNITY TECHNOLOGIES: *Unity Manual - Script Execution Order Settings*. http://docs.unity3d.com/Manual/class-ScriptExecution.html. Version: 2015. – [Online; Stand 06. Oktober 2015]

[Unity Technologies 2015f] UNITY TECHNOLOGIES: *Unity Public Relations - Company Facts*. http://unity3d.com/public-relations. Version: 2015. – [Online; Stand 06. Oktober 2015]

A MVC(S) Architekturübersicht

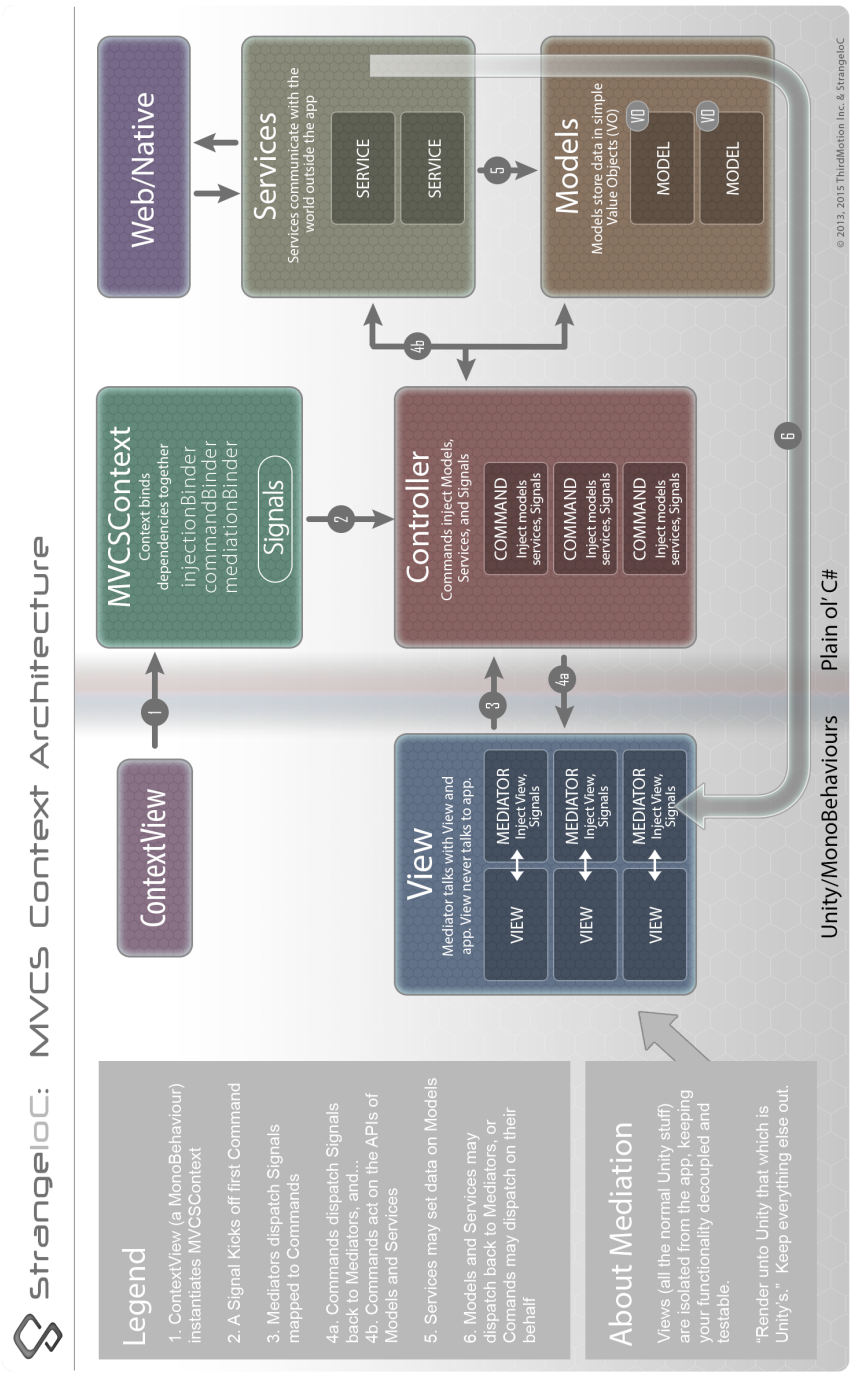

Abb. A.1: Schaubild MVC(S) Context Architektur. Quelle: Third Motion, Inc., 2015a

B Mediation in StrangeIoC

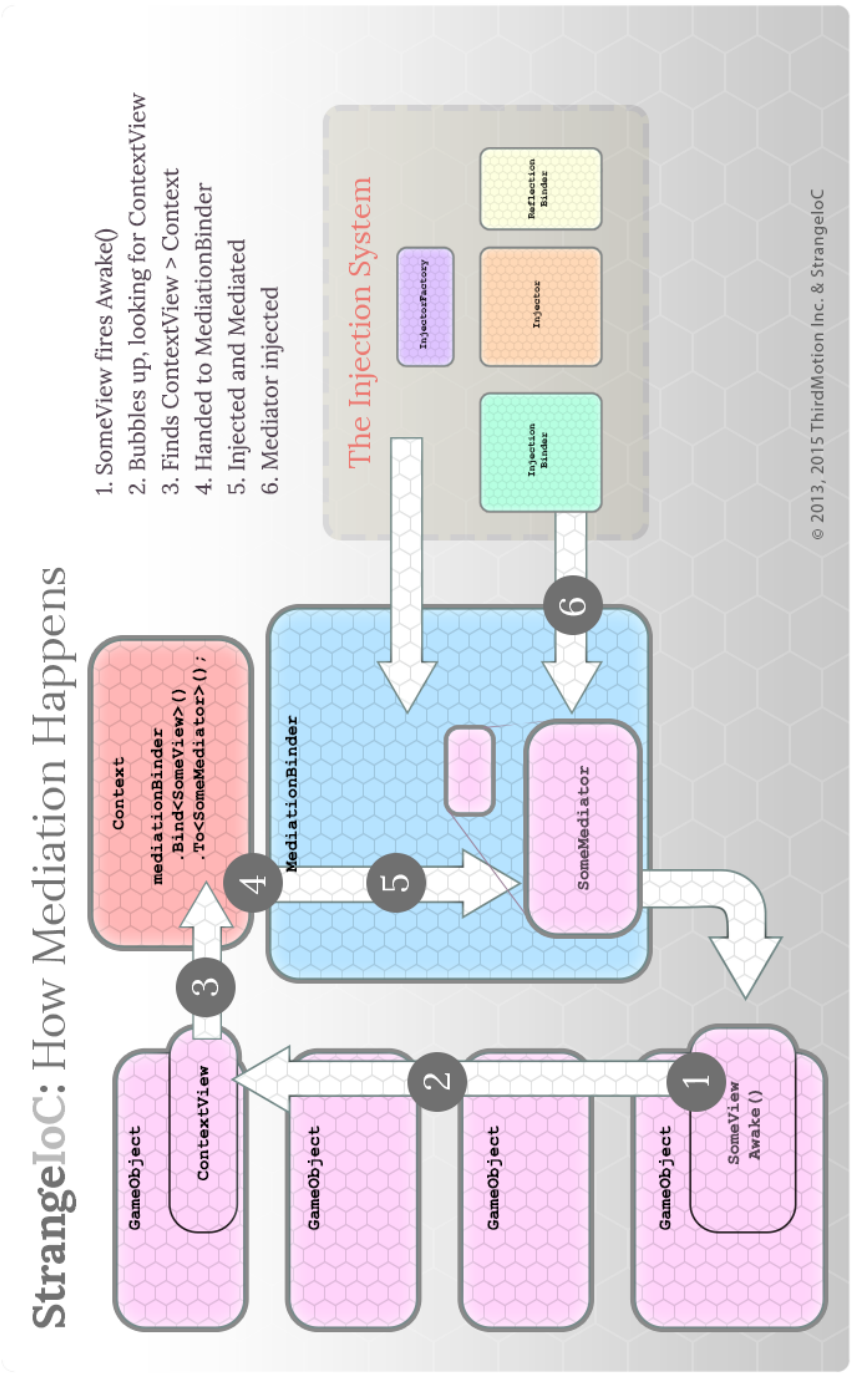

Abb. B.1: Schaubild zur Verdeutlichung des Mediation-Prozesses. Quelle: Third Motion, Inc., 2015c