HAW HAMBURG

Skalierbarkeit eines (Unity-)Projekts: Einführung eines Dependency Injection Frameworks, Assembly Definitions und Design Patterns

Verfasser:
Vincent Krenzke [redacted]

Erstgutachter:
Prof. Dr. Eike Langbehn
Zweitgutachterin:
Prof. Anke Günther

Masterarbeit

in

Master Zeitabhängige Medien - Games
Fakultät Design Medien und Informationen
Department Medientechnik

2. November 2023

Abstrakt

In einem Spielprojekt muss auf ständig wechselnde Anforderungen flexibel reagiert werden können. Gerade die Codebasis durchlebt dabei sehr viele Anpassungen und Wandlungen. Doch wie kann der Aufwand bei diesen Änderungen möglichst minimiert werden? Anhand eines Praxisbeispiels sollen Strukturen und Methoden aufgezeigt werden, welche die Codebasis robuster gestalten und den Weg für Skalierung bereiten. Konkret werden in einem Unity-Projekt Assembly Definitionen eingeführt, das Dependency Injection Framework Extenject integriert und Design Patterns sowie Prinzipien der Softwareentwicklung angewandt. Das Ergebnis spiegelt eine modulare sowie erweiterbare Codebasis wider und zeigt die ersten Erfahrungen mit einem Dependency Injection Framework auf.

Inhaltsverzeichnis

1 Einleitung

Projekte in der Spieleentwicklung können unterschiedliche Ausmaße annehmen. Von Gamejam-Projekten, welche innerhalb weniger Tage umgesetzt werden, über kleinere Hobbyprojekte, bis hin zu vollwertigen Spieletiteln mit mehreren Stunden Inhalt. In allen Anwendungsfällen kann es sinnvoll sein, eine bereits etablierte Spiel-Engine zu wählen, um somit den Fokus der Entwicklung auf das Spiel zu legen und zusätzlich eine vertraute Entwicklungsumgebung für ein Team zu schaffen. Doch in der Art und Weise wie die Entwicklung des Spiels konkret umgesetzt wird, gilt es zwischen verschiedenen Herangehensweise zu unterscheiden. Beispielsweise kann es bei der Erstellung eines Prototypen oder der Umsetzung eines Gamejam-Spiels sinnvoll sein Projektstrukturen und auch die Struktur des Codes hintenan zu stellen, um somit den Fokus auf die Umsetzung zu legen und schneller Ergebnisse zu erzielen.

Kommt es nun doch zu dem Fall, dass ein Projekt größere Ausmaße annimmt, egal ob von vornherein geplant oder sich während der Entwicklung dazu entschieden wurde, gibt es unterschiedlichste Möglichkeiten die Strukturen des Projekts und der Codebasis aufzubauen.

1.1 Zielsetzung

Das Ziel dieser Arbeit soll es sein, ein bereits bestehendes Projekt um Strukturen zu erweitern und anzupassen, so dass dieses für eine zukünftige Weiterführung vorbereitet ist. Konkret sollen dabei *Assembly Definitionen* eingeführt, ein *Dependency Injection Framework* verwendet und verschiedene Design Patterns angewandt werden. Das Ergebnis spiegelt sich durch eine verringerte Kompilierungszeit, die Modularität und Verständlichkeit der Codebasis sowie der Robustheit der Projektstruktur wider.

1.2 Eingrenzung

Die Arbeit beschäftigt sich ausschließlich mit dem Unity-Projekt des Spiels "Light of Atlantis". Dabei dient dieses als Basis für ein Spielprojekt, welches als Prototyp angedacht war, nun aber als vollwertiges Spiel weiterentwickelt werden soll.

Für die Skalierung eines Spielprojekts können unterschiedliche Bereiche betrachtet werden. Der Fokus soll hierbei auf der Codebasis und der Unity Engine als Entwicklungsumgebung liegen. Weitere Themen wie beispielsweise das Projektmanagement werden hierbei nicht berücksichtigt.

Der Ausgangspunkt des untersuchten Unity-Projekts bietet bereits eine gewisse Projektgröße, wodurch die hier angewandten Methoden, Frameworks und Prinzipien nur beispielhaft Anwendung finden. Eine vollständige Umstrukturierung soll als Ziel für das Unity-Projekt gesetzt werden, nicht aber für diese Arbeit.

1.3 Realisierung

Vor der Umsetzung sollen zunächst die Aspekte einer skalierbaren Codebasis untersucht werden. Diese dienen dann als Grundlage für die angewandten Techniken und bieten die Möglichkeit einer Einsortierung der vorgenommenen Anpassungen und deren Ergebnisse.

Als nächstes gilt es die bereits im Vorfeld identifizierten Methoden, Frameworks und Prinzipien anzuwenden. Hierzu wird zunächst das Unity-Projekt aktualisiert und somit auf einen modernen Ausgangszustand gebracht. Daraufhin folgen die Einführungen von Assembly Definitionen und einem Dependency Injection Framework. Bei Letzterem handelt es sich dabei um das Package *Extenject*, welches sich mit den Abhängigkeiten im Code beschäftigt und die bereits existierenden Events durch Signale ersetzen soll.

Mit Hilfe der eingesetzten Methoden und Frameworks gilt es dann, die Codebasis weiter zu Modularisieren und die Verständlichkeit des Codes zu erhöhen. Hierzu sollen Design Patterns wie das *Object Pooling* und die *Factory* Anwendung finden sowie das Prinzip *SOLID*.

Zur zusätzlichen Reflexion der Ergebnisse werden Experteninterviews durchgeführt. Hier sollen in zwei ausführlichen Gesprächen weitere Erfahrungen zu der Thematik eingeholt werden.

2 Grundlagen

Im folgenden Kapitel werden zunächst die Grundlagen einiger Methoden, Frameworks und Prinzipien näher beschrieben. Hierbei geht es um ein allgemeines Verständnis in der jeweilige Thematik sowie mögliche Anwendungsfälle aufzuzeigen.

2.1 Skalierbare Codebasis

Eine Codebasis weist vielerlei Eigenschaften auf. Um diese skalierbar zu gestalten, müssen viele Aspekte berücksichtigt werden. Im Kern existiert eine Codebasis, um die Funktion einer Anwendung zu definieren. Änderungen werden dabei regelmäßig durchgeführt und folgen einem bestimmten Ablauf. Abbildung 2.1 veranschaulicht diese Durchführungen. Zunächst geht es darum das konkrete Problem zu verstehen und sich mit den Anforderungen vertraut zu machen. Im Anschluss müssen die betroffenen Systeme und Schnittstellen verstanden und die entsprechenden Inhalte erarbeitet werden. Durch die Implementierung der Lösung folgt meist eine Umstrukturierung der Systeme, welche im Anschluss wieder an die Codebasis angepasst werden sollten. (Nystrom, 2015, S. 27-30)

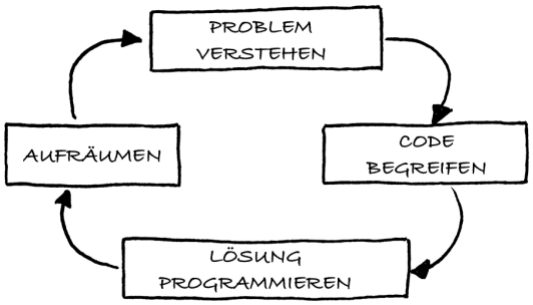

Abbildung 2.1. Die einzelnen Schritte einer Änderung an der Codebasis.
(Nystrom, 2015, S. 29)

Die einzelnen Schritte des beschriebenen Ablaufs lassen sich stark von den Eigenschaften der Codebasis beeinflussen. Beispielsweise profitiert die Verständlichkeit des Codes von einer guten Lesbarkeit und Struktur. Des Weiteren lassen sich Lösungen leichter implementieren, wenn bereits klar definierte Schnittstellen vorhanden sind und ggfs. wiederverwendet werden können. Dabei geht es häufig weniger darum den Code zu schreiben und mehr darum diesen zu organisieren. (Nystrom, 2015, S. 27-30)

Folgende Merkmale zeichnen dabei eine skalierbare Codebasis aus:

- Es lassen sich Änderungen vornehmen, ohne die vorhandene Struktur zu verändern. (Nystrom, 2015, S. 27-30)

- Veränderungen führen nicht zu Fehlern in bereits bestehenden Systemen. (Nystrom, 2015, S. 27-30)
- Änderungen lassen sich durch die Verständlichkeit des Codes und den Strukturen, mit einer Einfachheit durchführen. (Nystrom, 2015, S. 27-30)

Um solche Strukturen erreichen zu können, sind besonders wichtig Eigenschaften, wie die Modularisierung, Lesbarkeit, Erweiterbarkeit, Testbarkeit und Wiederverwendbarkeit. All das sind Aspekte einer Codebasis, die mit bereits etablierten Methoden, Frameworks und Prinzipien unterstützt werden können.

2.2 Assembly Definitionen

Als Assembly Definition versteht man in Unity ein Asset, welches einzelne Skripte in einer Assembly organisiert. Eine Assembly selbst beinhaltet dabei die kompilierten Klassen und Strukturen und bildet somit eine C# code library. Unity ordnet bereits neu angelegte Skripte in vordefinierte Assemblies ein. So landet der Code aus Editor-Ordnern innerhalb der Assembly-CSharp-Editor und der restliche Code innerhalb der Assembly-CSharp. (Unity Technologies, 2023a)

Diese Anordnung funktioniert gerade in kleineren Projekten sehr gut. Bei einer größeren Codebasis können dabei folgende Probleme auftreten:

- Jede Änderung in einem Skript hat zu Folge, dass alle anderen Skripte ebenfalls neu kompiliert werden müssen, wodurch sich die allgemeine Kompilierungszeit erhöht. (Unity Technologies, 2023a)
- Jedes Skript hat direkten Zugriff auf Typen aus allen anderen Skripten, wodurch sich die Refaktorisierung erschwert und die Modularisierung der Codebasis abnimmt. (Unity Technologies, 2023a)
- Alle Skripte werden für jede Plattform Kompiliert. (Unity Technologies, 2023a)

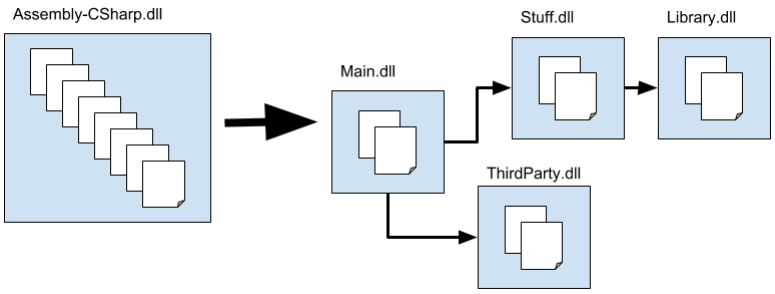

Abbildung 2.2. Durch die Aufteilung des Codes in mehrere Assemblies, kann Modularität und Wiederverwendung gefördert werden.
(*Quelle: Unity Technologies (2023), https://docs.unity3d.com/Manual/ScriptCompilationAssemblyDefinitionFiles.html*)

Durch das Definieren von eigenen Assemblies kann den gelisteten Problemen entgegengewirkt werden. Diesen Assemblies können dann die eigenen Skripte zugeordnet werden, wodurch diese nicht länger in den vordefinierten Assemblies landen. Abbildung 2.2 stellt einmal beispielhaft eine Aufteilung der Codebasis in unterschiedliche Assemblies dar. Wird nun eine Änderung in *Main.dll* vorgenommen, so muss lediglich diese neu kompiliert werden, da *Stuff.dll* oder *ThirdParty.dll* keine Referenzen auf *Main.dll* besitzen und somit nicht von Änderungen betroffen sind. Ebenso könnte die *Library.dll* Anwendung in einem anderen Projekt finden, da auch hier die Abhängigkeiten des Codes nur in eine Richtung bestehen und somit eine Modularität hergestellt wird. (Unity Technologies, 2023a)

2.3 Dependency Injection

Dependency Injection ist ein mächtiges Werkzeug, um Konstruktion und Nutzung von Objekten im Code zu trennen. Es dient als Anwendung für das Prinzip *Inversion of Control*, welches zweitrangige Verantwortungen eines Objekts an die zuständigen Objekte abgibt, so dass eine saubere Trennung der Verantwortungen stattfindet. (Martin, 2008, S. 157)

Ein einfaches Beispiel der Dependency Injection ist ein Konstruktor einer simplen C#-Klasse. Hier werden die benötigten Abhängigkeiten als Übergabeparameter definiert und somit bei der Erstellung des Objekts direkt übergeben. Dadurch muss die Abhängigkeit nicht erst bereitgestellt oder gar erst erstellt werden, wenn diese benötigt wird. In Unity stehen den Klassen, welche von *MonoBehaviour* erben und somit als Komponente an *GameObjects* zur Verfügung stehen, kein direkter Konstruktor bereit. Hier werden Abhängigkeiten oftmals durch den Unity-Inspektor zugewiesen. In den Klassen können beispielsweise durch public-Felder oder auch *Serialized Fields* eine Schnittstelle zum Unity-Inspektor geschaffen werden, welche dann auch für Abhängigkeiten verwendet werden kann. Es wird also eine Dependency Injection vorgenommen.

Hier kann nun die Art der Dependency Injection durch ein Framework angepasst werden. In dieser Arbeit findet das Dependency Injection Framework *Extenject* Anwendung. Dieses bietet die Injection für Konstruktoren, Felder, Eigenschaften und Funktionen an. Dabei wird das vorher definierte Prinzip der Inversion of Control auf die Spitze getrieben. Die Abhängigkeiten wandern in der Hierarchie der Anwendung immer weiter nach oben, bis diese im sogenannten *composition root* ankommen. Dieser stellt die höchste Ebene des *Object graph* der Anwendung dar und wird in dem Framework zum Großteil durch sogenannte *Installers* realisiert. Hier werden also alle Abhängigkeiten beim Start der Anwendung bereitgestellt und dann in die jeweiligen Objekte durch Attribute an den Feldern, Eigenschaften und Funktionen injected. (Modest Tree Media, 2023)

Folgende Vorteile können dadurch für eine Codebasis entstehen:

- Der Code kann weniger stark miteinander verstrickt sein und ist somit lose gekoppelt. (Modest Tree Media, 2023)
- Durch die lose Kopplung ist die Codebasis weniger anfällig bei Anpassungen. (Modest Tree Media, 2023)

- Die Struktur des Frameworks forciert das Nachdenken über die Schnittstellen der einzelnen Klassen, wodurch modularer Code angeregt wird. (Modest Tree Media, 2023)
- Ein einfacheres Testen des Codes ist durch die Bereitstellung der Abhängigkeiten gegeben. (Modest Tree Media, 2023)
- Die Klassen können sich auf ihre speziellen Pflichten fokussieren, wodurch das *Single Responsibility Principle* unterstützt wird. (Modest Tree Media, 2023)

2.4 Prinzipien

Prinzipien in der Softwareentwicklung können dazu beitragen, *Clean Code* zu produzieren. Dieser zeichnet sich durch eine lesbare, änderbare, erweiterbare und wartbare Codebasis aus. Im Folgenden sollen einmal auf grundlegende Prinzipien eingegangen werden, welche Anwendung in dem Projekt finden.

2.4.1 SOLID

Das Akronym SOLID steht für fünf Prinzipien in der Softwareentwicklung. Es wurde von Robert C. Martin geprägt und hilft dabei, den Code verständlicher, erweiterbar und modularer zu gestalten. (Martin, 2017, Design Principles)

Single Responsibility Principle (SRP)

In diesem Prinzip geht es darum, dass ein Modul nur für genau einen Akteur verantwortlich sein sollte. Mit Akteuren sind dabei Benutzer:innen oder weitere Interessengruppen gemeint, welche die Anforderungen an das System vorgeben und dadurch einen Grund für Änderungen darstellen. (Martin, 2017, SRP: The Single Responsibility Principle)

Häufig wird das Prinzip auch damit in Verbindung gebracht, dass eine Klasse oder Funktion nur eine einzige Verantwortung übernehmen und somit nur solche Logik beinhalten sollte. Hierdurch lässt sich die Codebasis in kleinere Abschnitte unterteilen und somit die Lesbarkeit, Erweiterbarkeit und Wiederverwendbarkeit fördern. (Wilmer Lin, 2022, S. 8, 11)

Open-Closed Principle (OCP)

Das Open-Closed-Prinzip besagt, dass eine Klasse offen für Erweiterungen, aber geschlossen für Modifikationen sein sollte. Klassen können somit um neue Funktionalitäten erweitert werden. Im Idealfall wird dabei aber der bestehende Code nicht weiter modifiziert. (Martin, 2017, OCP: The Open-Closed Principle)

Hierdurch lassen sich aufwändige und fehleranfällige Erweiterungen der Logik vermeiden. Die Klassen können in Zukunft leichter ausgebaut und gedebuggt werden. (Wilmer Lin, 2022, S. 12, 14)

Liskov Substitution Principle (LSP)

Dieses Prinzip beschäftigt sich mit abgeleiteten Klassen und wie diese robuster und flexibler gestaltet werden können. Dabei muss eine abgeleitete Klasse immer anstelle ihrer Basisklasse einsetzbar sein. Subtypen müssen sich also wie ihr Basistyp verhalten, bieten aber weiterhin die Möglichkeit der Erweiterung. Dadurch lässt sich die Flexibilität und Erweiterbarkeit der Codebasis erhöhen sowie Strukturen mit Interfaces und Vererbung aufzeigen. (Wilmer Lin, 2022, S. 15, 18, 21)

Interface Segregation Principle (ISP)

Bei dem Prinzip geht es darum, dass ein Client nicht von den Funktionen abhängig sein darf, die er gar nicht benötigt. Hierdurch sollen größere Interfaces vermieden und der Inhalt in kleinere und dadurch auch spezifischere Interfaces ausgelagert werden. Im Zuge dessen können nun einzelne Klassen sich die spezifischen Interfaces aussuchen und implementieren, wodurch unnötige Abhängigkeiten von Funktionalität vermieden wird. Dies hilft auch dabei die Codebasis weiter zu entkoppeln und in Zukunft einfacher zu verändern oder ein Deployment durchzuführen. (Wilmer Lin, 2022, S. 21, 24)

Dependency Inversion Principle (DIP)

Das Prinzip besagt, dass Klassen auf einem höheren Abstraktionslevel nicht von Klassen auf einem niedrigen Abstraktionslevel abhängig sein sollten. Beide sollten diese Abhängigkeit über Abstraktion definieren. Hierdurch lassen sich Abhängigkeiten reduzieren, die Codebasis weiter entkoppeln und die einzelnen Klassen zusammenhängender gestalten. (Wilmer Lin, 2022, S. 24, 25)

2.4.2 Don't Repeat Yourself (DRY)

David Thomas and Andrew Hunt beschreiben dieses Prinzip folgendermaßen: „Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."(David Thomas, 2019, S. 31). Dabei sammeln, organisieren, warten und nutzen die Programmierer:innen das angesprochene Wissen. Dieses kann sich mit der Zeit verändern. Beispielsweise können neue Anforderungen an ein System gestellt werden oder durchgeführte Tests zeigen, dass eine andere Lösung für ein Problem sinnvoller ist. So verbringen die Programmierer:innen einen großen Teil der Entwicklung damit, ein System anzupassen und zu warten und somit auch das Wissen neu anzuordnen. Kommt es nun zu Doppelungen des Wissens in einem System, so erhöht sich der Wartungsaufwand und die Fehleranfälligkeit. Hier kann durch das Prinzip entgegengewirkt werden. Dabei bezieht dieses sich nicht nur auf den Code sondern findet auch Anwendung in anderen Bereichen wie den Daten, der Dokumentation oder der API eines Systems. (David Thomas, 2019, Seiten 30-37)

3 Durchführung

Im Folgenden soll auf die konkrete Umsetzung eingegangen werden. Dabei werden die vorher definierten Methoden, Frameworks und Prinzipen Anwendung finden. Ebenfalls erfolgt eine Auswertung der Ergebnisse. Diese werden im Anschluss noch genauer reflektiert.

3.1 Aktualisierung der Tools

Da in das Projekt ein größeres Framework integriert wird und Strukturen in der Codebasis angepasst werden, ist dies ein optimaler Zeitpunkt, um die bereits eingesetzten Tools auf den neusten Stand zu bringen. Hierzu wird ein Update für Unity und die integrierten Pakete durchgeführt. Solche Updates können in größeren Projekten eine Menge Aufwand verursachen. Gerade Major Updates, welche Neuerungen und Verbesserungen mit sich bringen, können dabei vorher definierte Schnittstellen neu strukturieren und somit zu Konflikten führen (Preston-Werner, o. D.).

Dabei beinhaltet das Update folgende Pakete:

- Unity Inputsystem - Verarbeitung der Eingaben
- Game 2D WaterKit - Rendering des Wassers
- Pro Camera 2D - Kamerasystem
- Node Canvas - Behavior Tree für Gegner
- Astar Pathfinding Project - Pathfinding für Gegners

Obwohl viele Bereiche des Spiels von den Aktualisierungen betroffen sind, kommt es nicht zu Konflikten und das Spiel muss in keiner Hinsicht angepasst werden. Das hängt damit zusammen, dass es sich bei den Aktualisierungen um Minor oder Patch Versionen handelt, welche die Schnittstellen unberührt lassen. Ob die mitgebrachten Neuerungen Anwendung in dem Projekt finden können, bleibt abzuwarten. Die Behebung von Bugs, Performance-Verbesserungen oder die Erweiterung um Funktionalität, ist dabei ein Mehrgewinn für das Projekt.

Bei dem Update von Unity wird die verwendete Version 2020.3.33f1 auf die neuste LTS (Long Term Support) Version 2022.3.5f1 gebracht. Hierbei handelt es sich um ein Major Update. Unity stellt dabei Richtlinien zur Verfügung, welche wesentlichen Änderungen auflisten und Hilfestellungen bei den Anpassungen der Schnittstellen geben. Dabei weist Unity darauf hin, das ein Update auf eine neuere LTS Version immer Schrittweise geschehen sollte. So muss sich zunächst an den Richtlinien für

die 2021 LTS Version orientiert und gegebenenfalls Änderungen vorgenommen werden. Wenn bei diesem Update dann keine weiteren Konflikte im Weg stehen, kann der nächste Schritt auf die 2022 LTS Version durchgeführt werden. (Unity Technologies, 2023d)

Das Update verlief dabei ohne größere Schwierigkeiten, es traten lediglich kleinere Problem auf. Zum Beispiel wurden die Einstellungen bei den verwendeten 2D-Lichtern zurückgesetzt, da das zugehörige experimentelle Paket nun offiziell veröffentlicht ist.

Durch das Update wurden viele Verbesserungen im Editor geschaffen. Neben schnelleren Iterationszeiten oder verbesserten Tools für das Debuggen, gab es auch konkrete Verbesserungen der 2D-Tools. So lassen sich die bereits verwendeten Sprite Atlases nun auch im Editor nutzen und ein 2D Profiler für die Physik kann das Optimieren erleichtern. (Unity Technologies, 2023c)

Des Weiteren ist es nun möglich Features aus der C# 9.0 Version zu nutzen. Allerdings steht hier nicht der volle Umfang zur Verfügung. Dafür lassen sich nun die fehlenden Features der C# 8.0 Version im vollen Umfang einsetzen. So können nun Funktionen in Interfaces eine Standardimplementierung erhalten sowie Ranges angelegt werden, um beispielsweise Zugriff auf mehrere Elemente in einem Array zu erhalten. (Unity Technologies, 2023b)

3.2 Assembly Definitionen

Das Projekt kam bisher ohne eigens definierte Assembly Definitionen aus. Einige der genutzten Pakete haben ebenfalls keine Assembly Definitionen definiert, wodurch sich die vordefinierten Assemblies von Unity weiter vergrößern. Hierdurch kommt es bei Änderungen am eigenen Code zu längeren Kompilierungszeiten, da auch der Code der Pakete berücksichtigt wird.

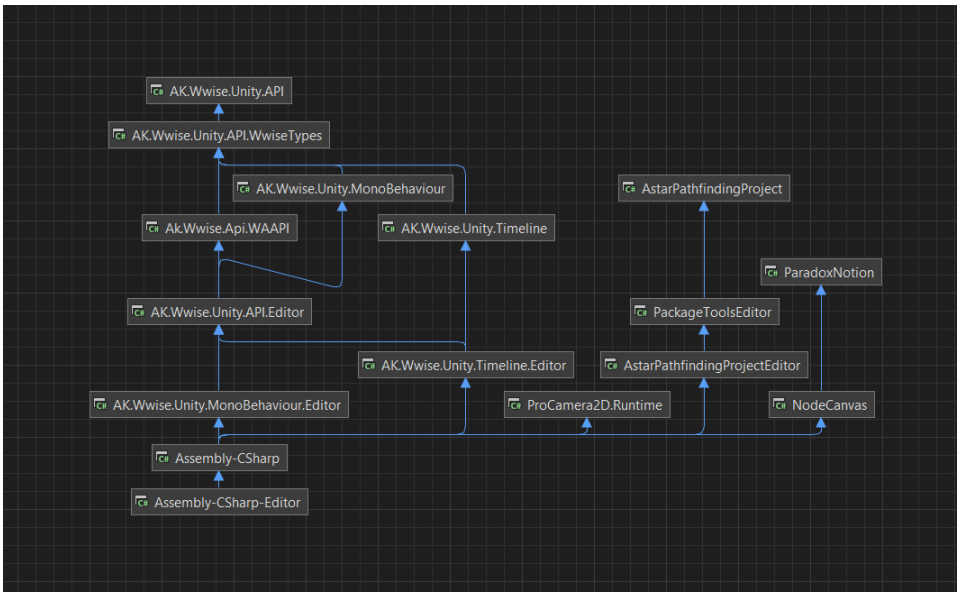

Abbildung 3.1. Das Projekt beinhaltet bereits eine Menge an Assemblies, jedoch sind einige Pakete nicht ausgelagert.
(*Quelle: Eigene Darstellung in JetBrains Rider 2022*)

Abbildung 3.1 zeigt dabei einmal den Aufbau des Projekts anhand der definierten Assemblies. Hierbei tauchen viele der genutzten Pakete im Projekt auf. Die übrigen landen dann gemeinsam mit dem eigenen Code in der Assembly-CSharp oder Assembly-CSharp-Editor. Nun sollen mit weiteren Assembly Definitionen die externen Pakete ausgelagert sowie die eigene Codebasis unterteilt werden.

Als erstes wird eine Assembly Definition explizit für die externen Pakete eingeführt. Hierzu liegen die Pakete innerhalb des Projekts in einem zugehörigen Ordner. Dieser wird nun mit einer Assembly Definition versehen, welche dann den nicht durch eine Assembly Definition abgedeckten Code der externen Pakete beinhaltet. Hierdurch werden auch alle zukünftigen Pakete miteinbezogen, wenn die Notwendigkeit besteht.

Der nächste Schritt beinhaltet die Unterteilung der eigenen Codebasis. Dabei werden Teile betrachtet, welche als eigenständiges Modul funktionieren und ausschließlich vom Rest der Codebasis verwendet werden, nicht aber selbst auch von der Codebasis abhängig sind. Dies ist eine wichtige Unterscheidung. Hierbei würde es zu einer zyklischen Anordnung der Assembly-Referenzen kommen, wodurch diese nicht aufgelöst werden können und es somit zu einem Fehler kommt (Unity Technologies, 2023a). Dies ist auch immer ein erster Hinweis darauf, dass der Code und dessen Abhängigkeiten sehr miteinander verstrickt sind und im Idealfall mehr modularisiert werden sollte.

Konkret wird jeweils eine Assembly Definition für den Ordner *Interfaces* und *DataPersistence* eingeführt. Bei ersterem handelt es sich dabei um eine Anordnung aller im Projekt befindlichen Interfaces. Diese bilden die Schnittstellen zu verschiedenen Komponenten des Systems. Dabei werden diese in den meisten Fällen vom Rest der Codebasis verwendet. Lediglich in zwei Fällen gibt es auch eine Referenz

auf die Codebasis selbst. Diese müssen umstrukturiert werden, wozu zunächst der Sinn eines solchen Ordners für Schnittstellen hinterfragt wird. Der Überblick aller Schnittstellen an einem Ort ist sicherlich ein Vorteil, jedoch müssen bereits jetzt einzelne Interfaces ausgelagert werden und auch in Zukunft können neue Interfaces Abhängigkeiten zur Codebasis aufweisen. Auffällig ist dabei auch, dass die betroffenen Schnittstellen Abhängigkeiten auf Teile der Codebasis aufweisen, welche sich für eigenständige Module anbieten und somit auch Assembly Definitionen erhalten können. So werden nun die einzelnen Interfaces in ihre zugehörigen Teile der Codebasis verschoben. Die übrigen erhalten dann eine gemeinsame Assembly Definition, welche in Zukunft möglicherweise hinfällig ist.

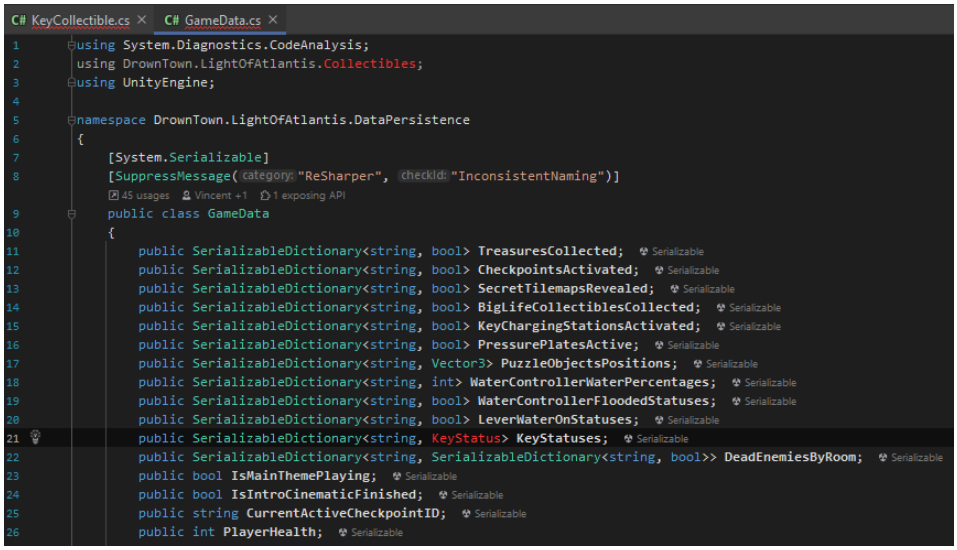

Abbildung 3.2. Eine Abhängigkeit der Collectibles verweilt noch in dem Modul der DataPersitence.
(*Quelle: Eigene Darstellung in JetBrains Rider 2022*)

Bei dem Modul der *DataPersistence* handelt es sich um das System zum Speichern und Laden des Spiels. Das gesamte Modul weist lediglich zwei Referenzen auf den Rest der Codebasis auf. Abbildung 3.2 zeigt dabei eine dieser Abhängigkeiten. Hierbei beinhaltet die Klasse *GameData* die Daten zum Speichern und Laden des Spiels. Unter anderem gibt es Schlüssel im Spiel, welche eingesammelt und benutzt werden können. Diese werden in den Daten durch einen *Enum* abgebildet und weisen dabei eine Abhängigkeit auf die *Collectibles* auf. Da die Werte eines *Enum* auch als *int* dargestellt werden können, lässt sich durch ein einfaches Refactoring diese Abhängigkeit vermeiden. So wird in der GameData-Klasse der Typ *KeyStatus* durch einen *int* ersetzt und die Daten somit nur noch durch primitive Datentypen bzw. Unity-Typen dargestellt.

Die übrige Abhängigkeit kommt durch den *EventManager* zustande. Dieser ermöglicht die Kommunikation zwischen Komponenten durch C#-Events. Hierfür können sich beliebige Komponenten registrieren oder ein Event auslösen. Momentan löst das Modul *DataPersistence* ein Event beim Speichern aus, welches von mehreren Komponenten der Codebasis genutzt wird. Eine Lösung wäre es, das Auslösen des Events aus dem Modul auszulagern und somit die Abhängigkeit zum *EventManager* zu vermeiden. Möglich wäre es auch eine Schnittstelle innerhalb des Moduls zu

schaffen. So kann der *EventManager* unterteilt werden und das Modul *DataPersistence* einen eigenständigen *EventManager* erhalten, über welchen dann die Kommunikation zwischen den Modulen stattfindet. Solch eine Schnittstelle lässt sich auch durch Signale in *Extenject* abbilden. Dies wird näher in einem späteren Abschnitt beleuchtet.

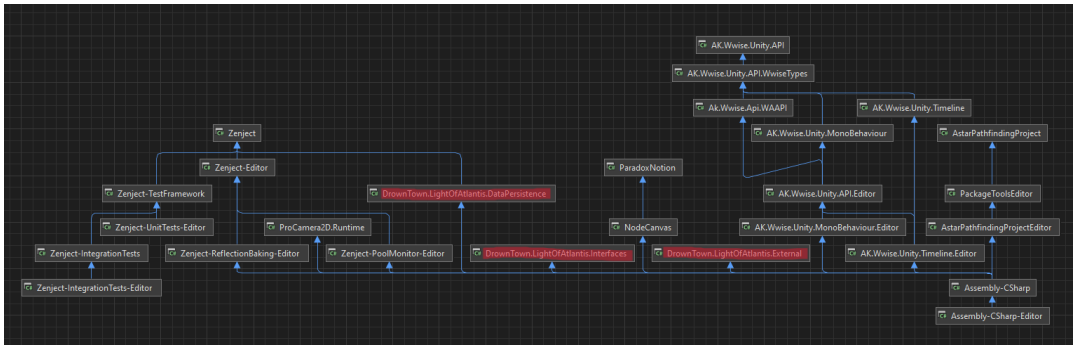

Abbildung 3.3. Übersicht der Assemblies mit Auslagerungen und dem Dependency Injection Framework *Extenject*.
(*Quelle: Eigene Darstellung in JetBrains Rider 2022*)

In Abbildung 3.3 ist nun erneut eine Übersicht der definierten Assemblies im Projekt gegeben. Dabei ist auch bereits das Dependency Injection Framework *Extenject* ins Projekt integriert worden. Zu sehen sind auch die eigens definierten Assemblies, welche Teile der Codebasis beinhalten und somit die vordefinierten Assemblies entlasten. Kommt es nun zu Änderungen in einer dieser vordefinierten Assemblies, so muss keine der anderen hier zu sehenden Assemblies neu kompiliert werden. Durch dieses Vorgehen lassen sich die Abhängigkeiten weiter verringern, wodurch das System auch weniger Fehleranfällig wird. Ebenso kann dadurch der Entwicklungsprozess beschleunigt werden, da sich die Kompilierungszeiten reduzieren.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code	Lines of Executable code
AstarPathfindingProjectEditor (Debug)	59	738	6	179	0	1.751
PackageToolsEditor (Debug)	70	114	4	60	0	248
ProCamera2D.Runtime (Debug)	100	0	0	0	0	0
AK.Wwise.Unity.Timeline.Editor (Debug)	62	43	4	35	294	112
AK.Wwise.Unity.Timeline (Debug)	73	252	5	81	1.467	475
AK.Wwise.Unity.API.WwiseTypes (Debug)	89	244	5	61	1.559	373
Ak.Wwise.Api.WAAPI (Debug)	92	296	3	98	2.578	505
AK.Wwise.Unity.MonoBehaviour.Editor (Debug)	73	474	6	165	3.218	1.133
Assembly-CSharp-Editor (Debug)	68	3.157	5	292	5.153	7.127
AK.Wwise.Unity.MonoBehaviour (Debug)	80	968	8	188	5.644	1.602
AK.Wwise.Unity.API.Editor (Debug)	80	1.104	4	241	6.458	2.161
AstarPathfindingProject (Debug)	79	5.084	8	353	6.950	9.744
NodeCanvas (Debug)	77	3.370	7	359	15.225	6.588
AK.Wwise.Unity.API (Debug)	86	4.002	7	273	16.620	6.772
ParadoxNotion (Debug)	83	6.605	6	490	26.916	11.673
Assembly-CSharp (Debug)	80	6.259	7	488	35.138	11.997

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code	Lines of Executable code
Ak.Wwise.Api.WAAPI (Debug)	92	296	3	98	2.578	505
AK.Wwise.Unity.API (Debug)	86	4.002	7	273	16.620	6.772
AK.Wwise.Unity.API.Editor (Debug)	80	1.104	4	241	6.458	2.161
AK.Wwise.Unity.API.WwiseTypes (Debug)	89	244	5	61	1.559	373
AK.Wwise.Unity.MonoBehaviour (Debug)	80	968	8	188	5.644	1.602
AK.Wwise.Unity.MonoBehaviour.Editor (Debug)	73	474	6	165	3.218	1.133
AK.Wwise.Unity.Timeline (Debug)	73	252	5	81	1.467	475
AK.Wwise.Unity.Timeline.Editor (Debug)	62	43	4	35	294	112
Assembly-CSharp (Debug)	79	3.054	8	304	16.618	5.811
Assembly-CSharp-Editor (Debug)	64	15	1	24	108	50
AstarPathfindingProject (Debug)	79	5.084	8	353	6.950	9.744
AstarPathfindingProjectEditor (Debug)	59	738	6	179	0	1.751
DrownTown.LightOfAtlantis.DataPersistence (Debug)	80	64	5	62	426	133
DrownTown.LightOfAtlantis.External (Debug)	76	6.283	7	415	40.159	13.130
DrownTown.LightOfAtlantis.Interfaces (Debug)	100	15	0	2	45	0
NodeCanvas (Debug)	77	3.370	7	359	15.225	6.588
PackageToolsEditor (Debug)	70	114	4	60	0	248
ParadoxNotion (Debug)	83	6.605	6	490	26.916	11.673
ProCamera2D.Runtime (Debug)	100	0	0	0	0	0
Zenject (Debug)	84	3.543	16	536	1.651	7.007
Zenject-Editor (Debug)	79	169	7	95	0	454
Zenject-IntegrationTests (Debug)	96	150	8	80	1.122	92
Zenject-IntegrationTests-Editor (Debug)	88	421	5	230	37	1.811
Zenject-PoolMonitor-Editor (Debug)	75	108	6	49	771	241
Zenject-ReflectionBaking-Editor (Debug)	70	221	4	117	0	595
Zenject-TestFramework (Debug)	84	39	3	36	0	95
Zenject-UnitTests-Editor (Debug)	92	1.152	3	522	10.528	2.996

Abbildung 3.4. Auflistung aller Assemblies im Projekt und dessen Metriken.
(*Quelle: Eigene Darstellung in Visual Studio 2019*)

Abbildung 3.3 schlüsselt einmal einige Code-Metriken auf. Interessant ist hierbei die jeweils ganz rechte Spalte, welche sich mit den Zeilen an ausführbaren Code beschäftigt. Darunter wird die Anzahl an Operationen im ausführbaren Code aufsummiert (Microsoft, 2022).
Im oberen Bereich der Grafik sind alle Assemblies zu beginn des Projekts aufgelistet. Die Assembly-CSharp beinhaltet dabei nahezu 12.000 Zeilen an ausführbaren Code. Der untere Bereich zeigt das Projekt nach Definition weiterer Assemblies und reduziert die ausführbaren Zeilen an Code auf rund 6.000 innerhalb der Assembly-CSharp. Der Umfang konnte also um ca. die Hälfte verringert werden. Ebenso konnte in der Assembly-CSharp-Editor das Ganze von knapp 7.000 Zeilen ausführbaren Code auf 50 herabgesetzt werden. Auch die anderen Metriken haben sich teilweise erheblich angepasst. Hier können aber keine direkten Schlüsse gezogen werden, ob diese sich verbessert haben oder einfach nur in andere Assemblies ausgelagert wurden. Ein möglicher Schritt wäre also die Aufschlüsselung der einzelnen Assemblies und dessen Inhalte.

Die durchgeführten Änderungen sollen nun durch Daten überprüft werden. Hierzu wird das Tool *Compilation Visualizer* zur Messung des Prozesses der Assembly-Kompilierung in Unity herangezogen (Marcel Wiessler, 2023). Mit Hilfe dessen werden Messungen des jetzigen Stands und des Ausgangszustands durchgeführt. Dabei wird jeweils zehnmal eine Änderung an einem Editor-Skript und an einem Nicht-Editor-Skript vorgenommen. Durch dieses Vorgehen wird gezielt eine Kompilierung der Assembly-CSharp sowie Assembly-CSharp-Editor forciert. Im Anschluss werden die Messwerte in eine Tabelle übertragen (siehe Anhang A). Diese umfassen dabei die Kompilierungszeit und die Reload-Zeit, wobei ersteres in die Kompilierung der jeweiligen Assembly sowie weitere Schritte des Kompilierungsprozesses unterteilt ist.

Aus den Messungen geht hervor, dass durch die Einführung weiterer Assemblies die durchschnittliche Zeit für Assembly-Kompilierungen verringert wurde. So benötigt die Assembly-CSharp-Editor nun knapp 0.6 Sekunden weniger Zeit für die Kompilierung. Die Assembly-CSharp spart im Durchschnitt 1.3 Sekunden bei der Kompilierung. Aufgrund des Einbindens eines zusätzlichen Frameworks und der Erstellung eigener Assemblies, nimmt die Reload-Zeit zu und mindert damit das Gesamtergebnis. Nichtsdestotrotz sind die erzielten Einsparungen ein deutlicher Mehrwert für das Projekt. So wird nun bei einer Änderung an der Codebasis und dem anschließenden Testen, jedes Mal annähernd 1.0 Sekunden gespart, was langfristig in der Summe einen großen Unterschied machen kann.

Anhand der Ergebnisse würde es Sinn ergeben, die Codebasis nun noch in weitere Assemblies zu unterteilen. Allerdings würde hierbei höchst wahrscheinlich nicht nochmal die gleiche Zeitersparnis erreicht werden. Ebenso gestaltet sich das weitere Vorgehen nun deutlich schwieriger, da hier die Struktur des Projekts eine große Rolle spielt. Der Code ist momentan sehr vernetzt und das Einfügen von sinnvollen Assembly Definitions führt zu vielen fehlenden Referenzen, wodurch nun zunächst der Code umstrukturiert werden muss.

3.3 Dependency Injection Framework

Bisher wurden die Abhängigkeiten im Projekt durch den Unity-Inspektor und innerhalb der Codebasis durch Konstruktoren bzw. Getter- und Setter-Methoden gelöst. Nun gilt es das Framework Extenject einzubinden und die Codebasis entsprechend umzustrukturieren. Hierzu muss zunächst ein Context-Objekt der Szene hinzugefügt und anschließend einzelne Installers erstellt und registriert werden. Die Context-Objekte sind dabei mit einem unterschiedlichen Scope erstellbar. So kann zwischen Project-, Decorator-, Scene- und GameObject-Context gewählt werden. Diese stellen die einzelnen Einstiegspunkte des Frameworks dar und lösen die Abhängigkeiten der Anwendung auf. Dabei bestimmt die Wahl des Context-Objekts die Verfügbarkeit der Abhängigkeiten. So stehen diese im Project-Context global zur Verfügung und im GameObject-Context nur dem jeweiligen GameObject.

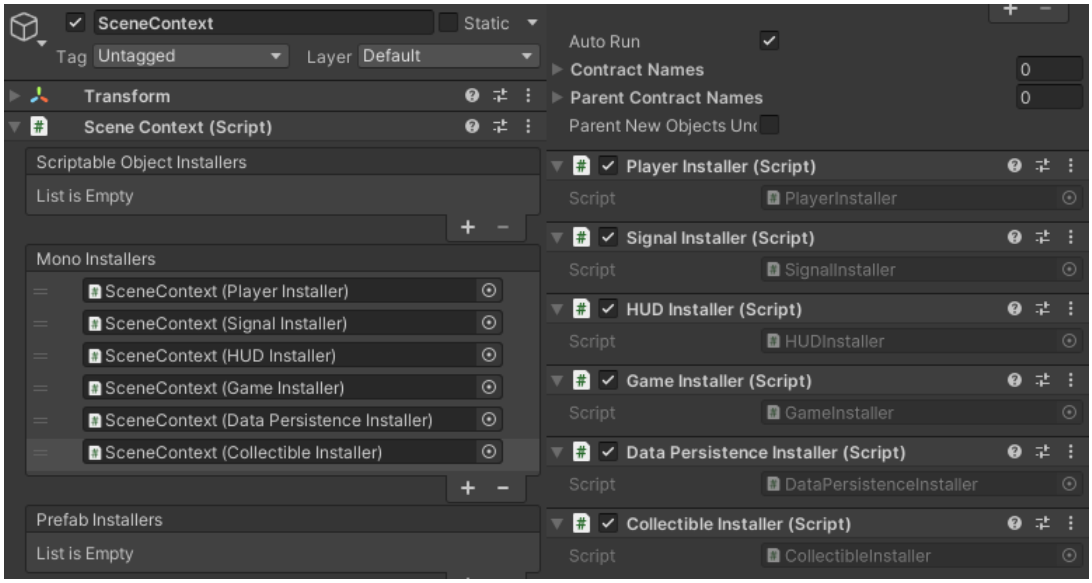

Abbildung 3.5. Mehrere Installers wurden dem Scene-Context-Objekt hinzugefügt und bilden somit den Aufbau der Abhängigkeiten.
(*Quelle: Eigene Darstellung in Unity 2022.3.5f1*)

Zunächst wird ein Scene-Context-Objekt erstellt und der Szene hinzugefügt. Daraufhin registrieren sich einzelne Installers auf dem Scene-Context, indem diese als Komponente hinzugefügt und in einer entsprechenden Liste aufgenommen werden. Abbildung 3.5 zeigt das entsprechende GameObject im Unity-Inspektor. Hier sind die angesprochenen Listen zu sehen, welche zwischen unterschiedlichen Arten von Installers unterscheidet. Ein Installer für Prefabs oder ScriptableObjects bringt dabei bereits bekannte Vorteile mit sich. So können Prefabs in weiteren Szenen wiederverwendet werden und auch spezifische Inhalte pro Prefab abbilden. ScriptableObjects eignen sich besonders gut für Einstellungen. Diese können zur Laufzeit angepasst und persistiert werden. Ebenso lassen sich die Einstellungen schnell und einfach austauschen.

```
public class PlayerInstaller : MonoInstaller
{
    Frequently called  0+7 usages  Vincent Krenzke *
    public override void InstallBindings()
    {
        Container.Bind<PlayerInputBinder>().FromComponentInHierarchy().AsSingle();
        Container.Bind<PlayerInputReader>().FromComponentInHierarchy().AsSingle();
        Container.Bind<RunningMovement>().FromComponentInHierarchy().AsSingle();
        Container.Bind<SwimmingMovement>().FromComponentInHierarchy().AsSingle();
        Container.Bind<WaterSurfaceMovement>().FromComponentInHierarchy().AsSingle();
        Container.Bind<JumpingMovement>().FromComponentInHierarchy().AsSingle();
        Container.Bind<WaterDash>().FromComponentInHierarchy().AsSingle();
        Container.Bind<Shockwave>().FromComponentInHierarchy().AsSingle();
        Container.Bind<SimpleMeleeAttack>().FromComponentInHierarchy().AsSingle();
        Container.Bind<Hookshot>().FromComponentInHierarchy().AsSingle();
        Container.Bind<SwimmingRotator>().FromComponentInHierarchy().AsSingle();

        Container.DeclareSignalWithInterfaces<SignalPlayerDied>();
        Container.DeclareSignalWithInterfaces<SignalPlayerDiveChanged>();
        Container.DeclareSignalWithInterfaces<SignalPlayerTeleported>();
        Container.DeclareSignalWithInterfaces<SignalPlayerControlsLocked>();
        Container.DeclareSignalWithInterfaces<SignalPlayerControlsUnlocked>();
        Container.DeclareSignalWithInterfaces<SignalPlayerStartedWaterDash>();
        Container.DeclareSignalWithInterfaces<SignalPlayerStoppedWaterDash>();
        Container.DeclareSignalWithInterfaces<SignalPlayerStartedShockwave>();
        Container.DeclareSignalWithInterfaces<SignalPlayerStoppedShockwave>();
        Container.DeclareSignalWithInterfaces<SignalInputDeviceChanged>();
        Container.DeclareSignalWithInterfaces<SignalPlayerHealthChanged>();
    }
}
```

Abbildung 3.6. Der Player Installer zeigt den typischen Aufbau und die Funktion eines Installers.
(*Quelle: Eigene Darstellung in JetBrains Rider 2022*)

Ein konkreter Installer ist in Abbildung 3.6 dargestellt. Dieser ist für die Abhängigkeiten der Player-Klassen zuständig und stellt diese in einem Container bereit. Hierbei definiert der Installer sogenannte Bindings, welche dem Container hinzugefügt werden. Dadurch kann in Zukunft der Container die Abhängigkeiten auflösen und angeforderte Objekte bereitstellen. In diesem Fall werden mehrere Komponenten dem Container hinzugefügt. Dabei sind diese bereits zur Editor-Laufzeit vorhanden und werden beim Start der Anwendung über eine Suche in der gesamten Szene herangezogen. Um diese durchaus aufwändige Suche nicht wiederholen zu müssen, lässt sich das Binding als *AsSingle* definieren und somit wird immer die gleiche Instanz genutzt sowie diese auch zwischengespeichert.

```
[Inject]
new *
private void Construct(PlayerInputReader playerInputReader, RunningMovement runningMovement,
    SwimmingMovement swimmingMovement, WaterSurfaceMovement waterSurfaceMovement,
    JumpingMovement jumpingMovement, WaterDash waterDash, Shockwave shockwave,
    SimpleMeleeAttack simpleMeleeAttack, Hookshot hookshot, SwimmingRotator swimmingRotator,
    PauseMenuController pauseMenuController, MapController mapController)
{
    _inputReader = playerInputReader;
    _runningMovement = runningMovement;
    _swimmingMovement = swimmingMovement;
    _waterSurfaceMovement = waterSurfaceMovement;
    _jumpingMovement = jumpingMovement;
    _waterDash = waterDash;
    _shockwave = shockwave;
    _simpleMeleeAttack = simpleMeleeAttack;
    _hookshot = hookshot;
    _characterSwimmingRotator = swimmingRotator;
    _pauseMenuController = pauseMenuController;
    _mapController = mapController;
}
```

Abbildung 3.7. Durch die Injection-Methode werden die einzelnen Abhängigkeiten der Klasse bereitgestellt.
(*Quelle: Eigene Darstellung in JetBrains Rider 2022*)

Als konkrete Umsetzung soll einmal genauer die *PlayerInputBinder*-Klasse betrachtet werden. Diese beschäftigt sich mit den Eingaben an die Anwendung und verknüpft diese mit einzelnen Aktionen. Somit weist diese Klasse Abhängigkeiten auf viele der definierten Komponenten im Installer auf. Abbildung 3.7 zeigt das Vorgehen anhand einer Injection-Methode. Da es sich bei der Klasse um ein MonoBehaviour handelt, steht dieser kein Konstruktor zur Verfügung. Es muss also eine eigens definierte Methode mit dem Attribut *Inject* versehen werden. Wird nun eine Instanz dieser Klasse angefordert oder steht diese bereits zur Editor-Laufzeit zur Verfügung, werden nach und nach die benötigten Abhängigkeiten anhand der Methode aufgelöst. Es werden also die einzelnen Komponenten nach dem gleichen Prinzip der Reihe nach abgearbeitet und somit die Abhängigkeiten bereitgestellt. Durch die vorherige Definitionen der einzelnen Komponenten im Installer, ist der Vorgang zur Beschaffung der Abhängigkeiten bereits festgelegt und der Container muss nur noch die Schritte ausführen.

```
1 usage  Vincent Krenzke
public class CustomBlobFactory : IFactory<UnityEngine.Object, Vector3, Quaternion, Transform, Blob>
{
    private readonly DiContainer _diContainer;

    Vincent Krenzke
    public CustomBlobFactory(DiContainer container)
    {
        _diContainer = container;
    }

    Frequently called  0+1 usages  Vincent Krenzke
    public Blob Create(Object prefab, Vector3 pos, Quaternion rot, Transform parent)
    {
        return _diContainer.InstantiatePrefabForComponent<Blob>(prefab, pos, rot, parent);
    }
}
```

Abbildung 3.8. Die Custom Factory ermöglicht die Erstellung von Objekten mit Lebensenergie zur Laufzeit.
(*Quelle: Eigene Darstellung in JetBrains Rider 2022*)

Bei der Bereitstellung der einzelnen Abhängigkeiten, gibt es unterschiedliche Möglichkeiten. So können durch die Bindings im Installer auch Prefabs instanziiert, direkte Zuweisungen durch den Unity-Inspektor anhand von entsprechenden Feldern getätigt oder neue Objekte von nicht-MonoBehaviour-Klassen erstellt werden. Lediglich die Erstellung von Objekten zur Laufzeit, also nachdem bereits alle Abhängigkeiten aufgelöst wurden, hängt mit einem gewissen Mehraufwand zusammen. Dies erfordert die Einbindung von Factories, welche für die Erstellung der einzelnen Objekte zuständig sind. Abbildung 3.8 zeigt solch eine Factory für die Instanziierung von Lebensenergie-Prefabs. Dabei wird auf dem Container direkt eine neue Instanz erzeugt und die benötigten Abhängigkeiten können dementsprechenden aufgelöst werden. Anstelle der Factory, könnte der Container auch direkt in den einzelnen Klassen genutzt werden. Dies würde jedoch gegen die empfohlenen Vorgehensweisen verstoßen, welche besagen, dass der Container nur im composition root layer Verwendung finden darf. Dazu gehören Factories und auch die Installers.

3.3.1 Signale

Bisher wurde die Kommunikation von Komponenten teilweise durch C#-Events realisiert. Hierdurch lassen sich Abhängigkeiten zwischen Komponenten vermeiden und stattdessen kommunizieren diese mit einem vermittelnden Objekt. Extenject bringt hier eine eigene Lösung mit sich. Durch sogenannte *Signals* lässt sich ein sehr ähnliches System aufbauen. Hierzu werden einzelne Signals durch einfache C#-Klassen abgebildet. Die Übergabeparameter werden durch Felder bzw. Properties definiert. In Installers müssen die entsprechenden Signals-Klassen dann deklariert werden, wodurch Objekte diese nun zur Kommunikation nutzen können.

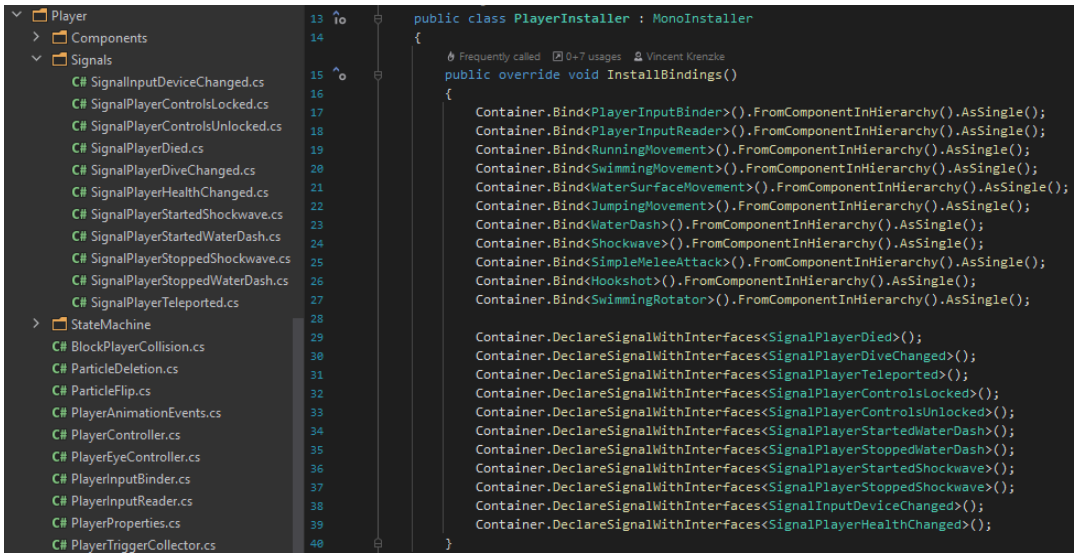

Abbildung 3.9. Innerhalb des Player Installer lässt sich ein Überblick der definierten Signals verschaffen.
(*Quelle: Eigene Darstellung in JetBrains Rider 2022*)

Nun gilt es den bisherigen EventManager durch Signals zu ersetzen. Dieser beinhaltet dabei die jeweiligen Definitionen der Events, dessen Übergabeparameter und

einen statischen Zugriff der einzelnen Methoden, um die Events auszulösen. Zunächst werden also die definierten Events in eigenständige C#-Klassen umgewandelt. Da diese dann auch wieder in den einzelnen Installers deklariert werden müssen, ist eine Gruppierung sinnvoll. So lassen sich die existierenden Signals innerhalb der zugehörigen Installers wiederfinden und sich somit ein Überblick verschaffen. Abbildung 3.9 beinhaltet einmal den Player Installer. Dieser fasst nun alle ursprünglichen Events der Player-Komponenten zusammen und deklariert diese als Signals.

Durch Fortführen dieser Methodik lassen sich alle weiteren Events auch als Signals auslagern. Hierdurch löst sich auch das Problem der auftretenden zyklischen Abhängigkeiten zwischen der Assembly zum Speichern und Laden und dem ursprünglichen EventManager. Die Assembly kann nun ein eigenes Signal definieren und dieses auslösen. Die restliche Codebasis registriert sich nun auf dieses Signal und kommuniziert auf diesem Wege mit dem System. Das System hat also keinerlei Referenzen auf außenstehende Komponenten.

```
[Inject]
Vincent Krenzke +1
public void Construct(SignalBus signalBus)
{
    _signalBus = signalBus;
}

Event function  Vincent Krenzke +1
private void OnEnable()
{
    _signalBus.Subscribe<SignalRoomChanged>(OnRoomChanged);
    _signalBus.Subscribe<SignalGameStartingCinematicStarted>(OnStartingCinematicStarted);
    _signalBus.Subscribe<ISignalVignetteEnabler>(OnVignetteEnabled);
    _signalBus.Subscribe<ISignalVignetteDisabler>(OnVignetteDisabled);
}

Event function  Vincent Krenzke +1
private void OnDisable()
{
    _signalBus.TryUnsubscribe<SignalRoomChanged>(OnRoomChanged);
    _signalBus.TryUnsubscribe<SignalGameStartingCinematicStarted>(OnStartingCinematicStarted);
    _signalBus.TryUnsubscribe<ISignalVignetteEnabler>(OnVignetteEnabled);
    _signalBus.TryUnsubscribe<ISignalVignetteDisabler>(OnVignetteDisabled);
}
```

Abbildung 3.10. Durch Abstract Signals lässt sich der Code weiter entkoppeln.
(*Quelle: Eigene Darstellung in JetBrains Rider 2022*)

Die konkrete Kommunikation läuft dabei über den *SignalBus* ab. Dieser wird den einzelnen Komponenten über den Konstruktor oder eine Inject-Methoden übergeben. Mithilfe des SignalBus können sich die Komponenten dann auf einzelne Signale registrieren oder diese auslösen. Um dabei Parameter übergeben zu können, wird eine neue Instanz des jeweiligen Signals erstellt und durch den Konstruktor die zugehörigen Felder bzw. Properties gesetzt.

Eine Besonderheit bei den Signals ist das Erstellen von Abstract Signals. Hierdurch lässt sich der Code weiter entkoppeln. Abbildung 3.10 zeigt einmal die Registrierung einzelner Signale. Diese koppeln sich dabei an die jeweiligen Klassen, was durch die Abstract Signals mithilfe von Interfaces wiederum entkoppelt werden kann. Durch

die Schaffung einer Schnittstelle läuft die Kommunikation ausschließlich über diese und die einzelnen Komponenten müssen nicht voneinander wissen. Zusätzlich ist es dadurch möglich, redundante Registrierungen zu vermeiden. Soll sich beispielsweise die Vignette bei mehreren Signals aktivieren, müsste die Klasse sich auf jedes Signal registrieren, obwohl jedes Mal die gleiche Methode ausgeführt wird. Durch Abstract Signals kann sich die Klasse stattdessen auf ein konkretes Interface registrieren und alle Signals, welche die Vignette aktivieren wollen, dieses Interface implementieren. Hierdurch bieten Signals weitere Abstraktionen und Modularisierungen an, wodurch spezifische Signals ausgelöst werden können und individuelle Komponenten auf diese reagieren.

3.4 Object Pooling

Das Object Pooling kann in der Verbesserung von Performance und der Reduzierung von Speichernutzung Anwendung finden (Nystrom, 2015, S. 361). Häufig wird es dabei bei der Instanziierung und Zerstörung von Objekten eingesetzt (Nystrom, 2015, S. 363). Solch ein Szenario ist auch in dem aktuellen Spieleprojekt gegeben. Hier wird bei dem Besiegen von Gegnern oder dem Zerstören bestimmter Container in der Umgebung, Lebensenergie hinterlassen. Diese besteht aus einzelnen Objekten, welche innerhalb eines Containers bzw. Gegners auf eine Anzahl von zwanzig anwachsen können. Es werden also während eines Frames bis zu zwanzig Objekte instanziiert und nach dem Einsammeln auch wieder zerstört. Bei größeren Gegnern oder weiteren Containern wäre auch durchaus eine größere Anzahl denkbar. Dies könnte gerade bei leistungsschwächeren Systemen zu spürbaren Performance-Einbrüchen führen.

Um diesen Einbrüchen entgegenzuwirken, wird ein Object Pooling implementiert. Hierdurch werden bereits zu Beginn des Spiels eine bestimmte Menge an Objekten für die Lebensenergie erzeugt. Diese werden wiederverwendet, indem die Objekte nach der Nutzung nicht zerstört, sonder innerhalb eines Pools angesammelt werden und somit wieder zur Verfügung stehen. Damit lassen sich die Performance-Probleme umgehen, indem die Belastung auf den Start des Spiels verlagert wird, dafür aber während des Spielens keinerlei Einbrüche auftreten.

```
private void Awake()
{
    PoolSmallBlobs = new ObjectPool<Blob>(OnBlobSmallCreate, actionOnGet: OnBlobTake,
        OnBlobRelease, OnBlobDestroy, collectionCheck: true, defaultCapacity, maxSize);
    PoolBigBlobs = new ObjectPool<Blob>(OnBlobBigCreate, actionOnGet: OnBlobTake,
        OnBlobRelease, OnBlobDestroy, collectionCheck: true, defaultCapacity, maxSize);
}

1 usage  Vincent Krenzke *
private Blob OnBlobSmallCreate()
{
    var blob = _blobFactory.Create(smallBlobPrefab, new Vector3(x: -1000, y: -1000, z: -1000),
        Quaternion.identity, transform);
    blob.gameObject.SetActive(false);
    blob.BlobPool = this;
    blob.GetComponent<LifeCollectible>().BlobPool = this;

    return blob;
}

2 usages  new *
private void OnBlobRelease(Blob blob)
{
    blob.transform.position = new Vector3(x: -1000, y: -1000, z: -1000);
    blob.transform.SetParent(transform);
    blob.gameObject.SetActive(false);
}
```

Abbildung 3.11. Mit der Unity internen Lösung wird ein Object Pool für die Lebensenergie im Spiel erstellt.
(*Quelle: Eigene Darstellung in JetBrains Rider 2022*)

Für die Umsetzung wird eine Unity interne Klasse genutzt. Diese bildet bereits die Funktionalitäten des Object Poolings ab. Lediglich die genauen Details müssen definiert werden. Abbildung 3.11 zeigt die Implementation anhand der Lebensenergie im Spiel. Da es zwei unterschiedliche Größen der Objekte gibt, wird jeweils ein eigenständiger Object Pool erstellt. Hierbei kann die maximale Größe sowie die anfängliche Kapazität des Object Pools bestimmt werden. Ebenso muss definiert werden, was mit den Objekten bei der Entnahme und Freigabe geschehen soll. So werden hier die Objekte jeweils aktiviert bzw. deaktiviert und dessen Position in der Welt sowie in der Hierarchie der Szene angepasst. Des Weiteren können dabei auch zusätzliche Objekte erstellt werden, wenn der Object Pool erschöpft ist und die Kapazität noch nicht das Maximum erreicht hat. Somit kann flexibel auf die Nachfrage an Objekten reagiert werden und gleichzeitig ist durch das Maximum sichergestellt, dass der Object Pool nicht auf ein undefiniertes Ausmaß anwächst.

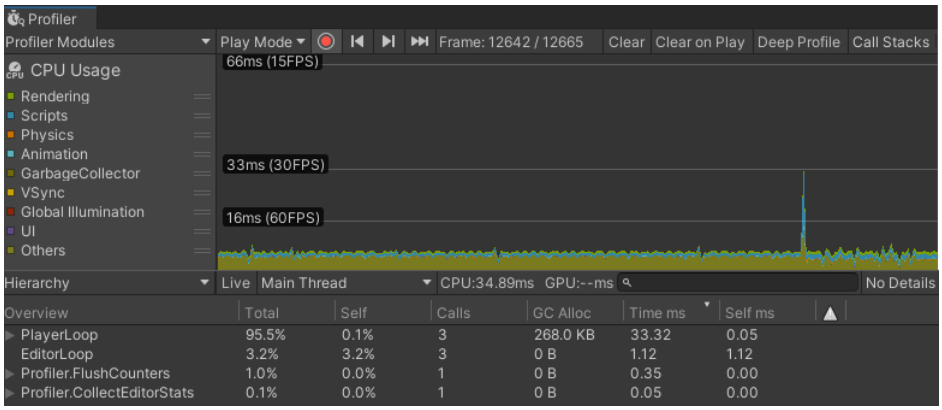

Abbildung 3.12. Durch das Erstellen der Lebensenergie Objekte werden sowohl die CPU als auch der Speicher belastet.
(*Quelle: Eigene Darstellung in Unity 2022.3.5f1*)

Im Folgenden soll einmal das Problem und die Lösung der Einbrüche anhand von Daten veranschaulicht werden. Abbildung 3.12 zeigt dabei die Performance während eines kurzen Spielabschnitts. Wenn im Spiel nicht viel passiert, also der Charakter sich lediglich im statischen Raum bewegt, erreicht das hier getestete System rund 300 FPS und eine Speicherzuweisung an den Garbage Collector von 400 Bytes. Wird nun ein bestimmter Container in der Umgebung mit einem Angriff zerstört, führt dies zu einer Instanziierung von 20 Objekten innerhalb eines Frames. Der Spike auf Abbildung 3.12 hält genau diesen Moment fest. Hier senken sich die FPS auf etwa 30 herab und es wird eine Speicherzuweisung von 268 Kilobytes erreicht. Dies gleicht sich dann über mehrere Frames wieder den ursprünglichen Wert an.

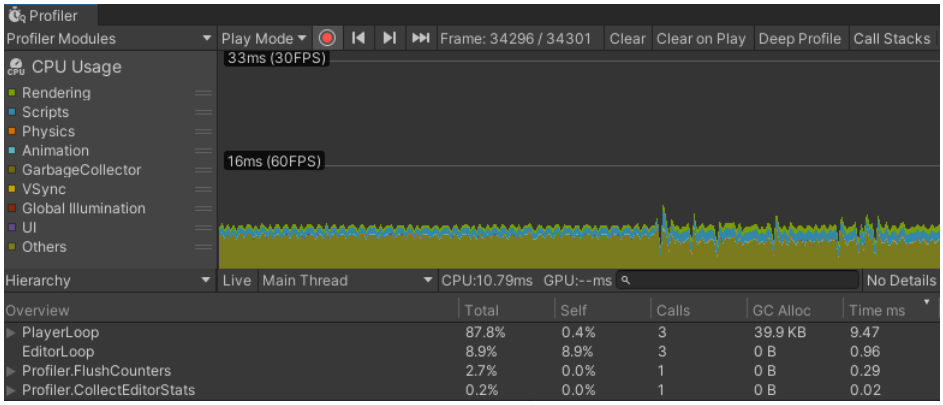

Abbildung 3.13. Durch das Object Pooling verringert sich die Auslastung der CPU und des Speichers deutlich.
(*Quelle: Eigene Darstellung in Unity 2022.3.5f1*)

Durch das implementierte Object Pooling kann diesen Einbrüchen entgegengewirkt werden. Abbildung 3.13 hält nun ebenfalls den vorher definierten Spielabschnitt fest. Der Spike erscheint nun deutlich kleiner und auch die Daten unterstützen diese Annahme. Zum Zeitpunkt der 20 angeforderten Objekte sinken die FPS nun auf 105 herab und die Speicherzuweisung erreicht einen Wert von knapp 40 Kilobytes. Komplett lässt sich die zusätzliche Belastung an das System jedoch nicht beseitigen. Hier spielt zum einen die gleichzeitige Aktivierung von 20 Objekten eine Rolle. Dies

führt innerhalb der Unity Engine zu mehreren internen Aufrufen von Funktionen. Zum Anderen ist auch die derzeitige Implementierung des Angriffs nicht optimiert, wodurch weitere Speicherzuweisungen durch Koroutinen anfallen.

3.5 Prinzipien

Im Folgenden sollen einige der bereits angesprochenen Prinzipien Anwendung in dem Projekt finden. Dabei sind diese bereits teilweise in dem Projekt vorhanden, können jedoch genauer betrachtet und ggfs. ausgebaut werden. Das Ganze soll beispielhaft an unterschiedlichen Komponenten des Projekts dargestellt werden. Das Ergebnis spiegelt eine einheitlichere und strukturierte Codebasis wider.

Single Responsibility Principle (SRP)

Die ursprüngliche Spielidee basierte auf einem einzigen spielbaren Charakter. Dies war ein Roboter, welcher sich mithilfe unterschiedlicher Fähigkeiten durch die Räume bewegt. Dieses Konzept hat sich nun weiterentwickelt und der spielbare Charakter ist eine Licht-Entität, welche verschiedene Roboter beherbergen und somit auch steuern kann. Hierdurch muss unter anderem die jetzige Player-Klasse umstrukturiert werden. Abbildung 3.14 zeigt die Aufteilung der einzelnen Player-Komponenten. Diese werden nun den einzelnen Robotern zugeordnet und finden Anwendung in den speziellen Fähigkeiten, da jeder Roboter sich bereits auch in der Fortbewegung unterscheidet. Die Komponenten erfüllen dabei jeweils eine spezielle Aufgabe und bleiben durch die Unterteilungen dabei überschaubar. Die ursprüngliche Player-Klasse wird aufgebrochen und dient nun als Abstraktionsebene für einen Player Controller, welcher für allgemeine Funktionalitäten der unterschiedlichen Roboter zuständig ist.

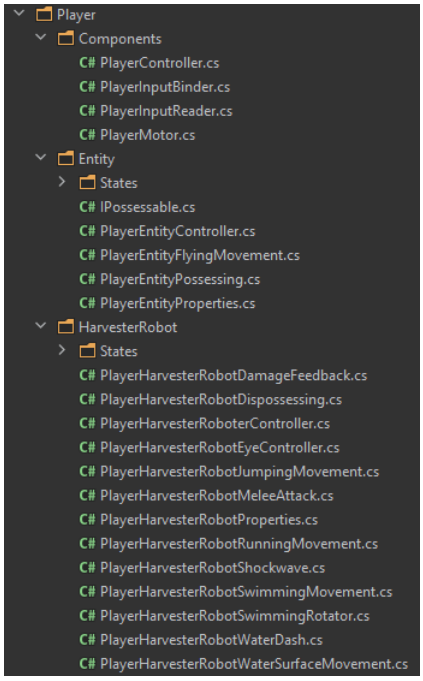

Abbildung 3.14. Durch die Anpassung des Konzepts und dem Verfolgen des SRP, ergibt sich eine neue Struktur der Player-Komponenten.
(*Quelle: Eigene Darstellung in JetBrains Rider 2022*)

Interface Segregation Principle (ISP)

Durch die unterschiedlichen spielbaren Roboter, erscheint eine einheitliche Schnittstelle zur Definition der einzelnen Roboter durchaus sinnvoll. Hierfür wird ein Interface bereitgestellt, welches die Schnittstellen eines steuerbaren Roboters enthält. Zunächst werden also alle Gemeinsamkeiten der Roboter innerhalb eines Interfaces gesammelt. Hierunter fallen Dinge, wie das Ausführen von Aktionen bei bestimmten Eingaben, einzelne Komponenten auf dem GameObject oder der aktuelle Raum, in dem sich der Roboter befindet. Durch die Implementation des Interfaces kann ein Roboter dann an den Rest des Systems angebunden und einzelne Roboter auch zur Laufzeit ausgetauscht werden.

```
public interface IPlayerMotorMovable
{
    Frequently called  2 usages  2 implementations  Vincent Krenzke
    public IDamageable PlayerDamageFeedback { get; }
    2 usages  2 implementations  Vincent Krenzke
    public IPlayerLandMovement PlayerLandMovement { get; }
    2 usages  2 implementations  Vincent Krenzke
    public IPlayerWaterMovement PlayerWaterMovement { get; }
    1 usage  2 implementations  Vincent Krenzke
    public IPlayerSwimmingRotation PlayerSwimmingRotation { get; }
    2 usages  2 implementations  Vincent Krenzke
    public IPlayerWaterSurfaceMovement PlayerWaterSurfaceMovement { get; }
    1 usage  2 implementations  Vincent Krenzke
    public IPlayerAction PlayerPossessAction { get; }
    2 usages  2 implementations  Vincent Krenzke
    public IPlayerAction PlayerFirstAction { get; }
    2 usages  2 implementations  Vincent Krenzke
    public IPlayerAction PlayerSecondAction { get; }
    2 usages  2 implementations  Vincent Krenzke
    public IPlayerAction PlayerThirdAction { get; }
    3 usages  2 implementations  Vincent Krenzke
    public PlayerMotor PlayerMotor { get; }
    Frequently called  4 usages  2 implementations  Vincent Krenzke
    public PlayerProperties PlayerProperties { get; }
    1 usage  2 implementations  Vincent Krenzke
    public InputActionAsset PlayerInputActionAsset { get; set; }
    Frequently called  4 usages  2 implementations  Vincent Krenzke
    public PlayerActionStateMachine PlayerActionStateMachine { get; }
    Frequently called  34 usages  2 implementations  Vincent Krenzke
    public PlayerController PlayerController { get; }
    Frequently called  21 usages  2 implementations  Vincent Krenzke
    public Rigidbody2D Rigidbody2D { get; }
    Frequently called  3 usages  2 implementations  Vincent Krenzke
    public Collider2D Collider2D { get; }
    Frequently called  13 usages  2 implementations  Vincent Krenzke
    public Animator Animator { get; }
    2 implementations  Vincent Krenzke
    public SpriteRenderer SpriteRenderer { get; }
    Frequently called  3 usages  2 implementations  Vincent Krenzke
    public Transform FlippableTransform { get; }
    Frequently called  4 usages  2 implementations  Vincent Krenzke
    public ParticleSystem ParticleSystemLanded { get; }
    Frequently called  7 usages  2 implementations  Vincent Krenzke
    public ParticleSystem ParticleSystemRunning { get; }
    Frequently called  7 usages  2 implementations  Vincent Krenzke
    public ParticleSystem ParticleSystemSwimming { get; }
    Frequently called  1 usage  2 implementations  Vincent Krenzke
    public ParticleSystem ParticleSystemWaterSurface { get; }
    1 usage  2 implementations  Vincent Krenzke
    public BlockPlayerCollision BlockPlayerCollision { get; }
    Frequently called  13 usages  2 implementations  Vincent Krenzke
    public Room CurrentBelongingRoom { get; set; }
```

Abbildung 3.15. Ohne das ISP ist das Interface für die Licht-Entität nicht kleinteilig genug.
(*Quelle: Eigene Darstellung in JetBrains Rider 2022*)

Die angesprochene Licht-Entität lässt sich ebenfalls steuern und nutzt dabei die gleichen Funktionen. Allerdings werden nicht alle definierten Eigenschaften des Interfaces benötigt, wodurch gegen das Interface-Segregation-Prinzip verstoßen wird. Abbildung 3.15 zeigt den Ausgangszustand des genutzten Interfaces. Hier ist es auf jeden Fall sinnvoll, das Interface in kleinere Interfaces zu unterteilen und diese damit auch spezifischer zu Gestalten. Dadurch lassen sich einzelne Teile auch für die Licht-Entität wiederverwenden und in Zukunft flexiblere Objekte definieren.

Dependency Inversion Principle (DIP)

In dem Spiel gibt es Tore, welche über Druckplatten angesteuert und somit geöffnet oder geschlossen werden können. Momentan kennen sich beide Komponenten gegenseitig und besitzen direkte Referenzen aufeinander, wodurch eine starke Koppelung entsteht. Um diese Komponenten zu entkoppeln und in Zukunft flexibler und robuster zu gestalten, lässt sich das Prinzip der Dependency Inversion anwenden. Hier wird nun ein Interface für die Kommunikation definiert. Dieses wird von den Toren implementiert und beinhaltet dabei eine Methode zum Aktivieren und Deaktivieren sowie den aktuellen Status. Die Druckplatten steuern nun nur noch dieses Interface an und weisen somit keine direkten Referenzen auf die Tore auf.

Hierdurch lassen sich die Abhängigkeiten reduzieren, wodurch zukünftige Änderungen weniger Auswirkungen auf die einzelnen Komponenten haben. Ebenso lässt sich das System einfacher erweitern. Nicht nur Tore können von den Druckplatten angesteuert werden. Alle Komponenten, die das Interface implementieren, sind direkt nutzbar. Das unterstützt auch das Open-Closed-Prinzip, welches die Erweiterbarkeit, nicht aber die Veränderung, von einzelnen Komponenten vorsieht. Ohne das Interface müsste jede neue Komponente, die von einer Druckplatte angesteuert werden soll, in der Druckplatte wiederum aufgenommen werden. Hierdurch würde ständig der Code verändert und weitere Abhängigkeiten geschaffen werden.

Don't Repeat Yourself (DRY)

Die Steuerung des Spiels ist momentan mit Tastatur und Maus oder dem Controller möglich. Dabei kann man während des Spielens bequem hin und her wechseln. Damit dieser Wechsel auch in den einzelnen Menüs funktioniert, muss das Event abgefangen und entsprechende Methoden ausgeführt werden. Davon sind die Credits, das Hauptmenü und das Pausenmenü betroffen. In allen Anwendungsfällen wird in einem jeweils zugehörigen Controller-Skript das Event abgefangen und neben spezifischen Methoden auch häufig gleicher Code ausgeführt. Dieser Code beschäftigt sich dabei beispielsweise mit der Navigation der einzelnen Menüs. So muss bei einem Wechsel auf die Controller-Steuerung ein Element ausgewählt werden, welches navigierbar ist und von dort aus mit dem Controller der Rest des Menüs erreichbar ist.

Um hier die unnötige Wiederholung des Codes zu vermeiden, wurde dieser in eine eigenständige Klasse ausgelagert. Bei Änderung an dem Verhalten muss nun nur an einer einzigen Stelle gearbeitet werden. Der spezifische Code kann weiterhin in den zugehörigen Klassen bleiben. Dieser wird dann durch das auslösen eines Events angesteuert und hat somit keinerlei explizite Verknüpfungen zum zuständigen System.

4 Evaluation

Im abschließenden Kapitel sollen die Ergebnisse reflektiert und eingeordnet werden. Dabei wird auf die Zielsetzung Bezug genommen und diese näher betrachtet. Eine Auswertung der Experteninterviews soll die erlangten Erkenntnisse stützen bzw. weitere Ansätze aufzeigen. Zum Schluss wird ein Fazit gezogen und die konkreten Erfahrungen definiert.

4.1 Assembly Definitionen

Die integrierten Assembly Definitionen haben zum erwünschten Ergebnis geführt. Es wurde eine Verkürzung der Kompilierungszeiten erreicht. Ebenso wurden die Strukturen der Codebasis genauer analysiert. Hierdurch ließen sich Systeme identifizieren, welche als eigenständiges Modul agieren können und somit keinerlei Referenzen auf den Reste der Codebasis aufweisen. Die Modularität wurde dadurch weiter ausgebaut.

Die Kompilierungszeiten haben gezeigt, dass sich schon bereits wenige oberflächliche Assembly Definitionen lohnen können. Viele der genutzten externen Packages verfügen über keine eigenständigen Assembly Definitionen. In den geführten Interviews gaben beide Personen an, dass sie eigene Assembly Definitionen für externe Packages definieren (Person A, Zeile 9-10)(Person B, Zeile 8-10). Dabei werden die externen Packages beim Importieren darauf überprüft, ob diese bereits Assembly Definitionen beinhalten. Bei Bedarf werden dann weitere hinzugefügt. In der Durchführung dieser Arbeit wurde eine einzige Assembly Definition für alle externen Packages definiert. Dadurch konnte die größte Zeitersparnis erlangt werden. Um die Abhängigkeiten der Codebasis auf die externen Packages weiter zu unterteilen, könnte der Einsatz einzelner Assembly Definitionen durchaus Sinn ergeben. Hierdurch können dann die externen Packages an den jeweiligen Stellen im Code eingebunden werden, ohne dabei Abhängigkeiten auf weitere Packages als Nebeneffekt aufzubauen.

Für beide Personen ist die Verringerung der Kompilierungszeiten ein schöner Nebeneffekt. Hauptsächlich werden die Assembly Definitionen jedoch zur Strukturierung der Codebasis verwendet (Person A, Zeile 30-34)(Person B, Zeile 14-16). Hierdurch lässt sich der Code weiter separieren und die Abhängigkeiten sind klar voneinander getrennt. Des Weiteren erkennt man dadurch auch früh eine Kopplung verschiedener Module, wodurch die einzelnen Referenzierungen hinterfragt und alternative Lösungen für das Problem gesucht werden können. Auch innerhalb der eigenen Codebasis können weitere Assembly Definitionen Anwendung finden. So müssen zunächst weitere Systeme identifiziert werden, welche als eigenständige Module auftreten können bzw. keine Abhängigkeiten zum Hauptteil der Codebasis aufweisen. Hierfür konnten die eingesetzten Desing Patterns und Prinzipien bereits

die Abhängigkeiten reduzieren. Dies muss nun fortgesetzt und in den restlichen Teilen der Codebasis umgesetzt werden.

4.2 Dependency Injection Framework

Durch die Einbindung des Dependency Injection Frameworks Extenject, konnte das Konzept einer alternativen Dependency Injection anhand eines Anwendungsbeispiels näher betrachtet werden. Nach einer anfänglichen Einarbeitungszeit sowie dem Einlesen in die Dokumentation, konnten erste Ergebnisse erzielt werden. Somit wurden die C#-Events durch Signals ersetzt und erste Dependency Injection durch das Framework vorgenommen. Hierbei zeigten sich auch die deutlichen Unterschiede in der Arbeitsweise. Diese erschien zunächst sehr ungewohnt und viele vermeintlich einfache Umsetzungen scheiterten bereits früh an der mangelnden Erfahrung. Hinzu kam, dass bereits eine große Codebasis existierte und das Framework nun in diese integriert werden musste. Als sich dann aber die ersten Konzepte etablierten, konnten schnell die Eigenheiten und der Mehrwert identifiziert werden.

So ließen sich die Abhängigkeiten, wenn diese in dem Framework definiert waren, unproblematisch verwenden. Dabei tauchte häufig das Problem auf, dass wenn eine Komponente in das Framework integriert wurde, diese meist auf weiteren Komponenten und somit auch auf Abhängigkeiten basierte, was zu zusätzlichen Anpassungen führte, obwohl zunächst nur ein kleiner Teil angepasst werden sollte. Auch hier spielt der Faktor einer bereits existierenden Codebasis eine große Rolle. Ein weiteres Problem stellten die Abhängigkeiten in Verbindung mit Objekten, welche zur Laufzeit erstellt werden, dar. Das Framework definiert zunächst alle Abhängigkeiten zum Start der Anwendung. Kommen nun zur Laufzeit weitere hinzu, muss über umständliche Factories das Objekt bereitgestellt werden. Hierdurch kommt es zu viel unnötigen Boilerplate-Code. Ebenso ist die Syntax nicht sehr intuitiv gewesen, was zusätzlichen Zeitaufwand bedeutete und zu Frustrationen geführt hat.

Bei der Umstellung der C#-Events auf die Framework internen Signals, konnten einige Problem gelöst werden. So konnte das System zum Speichern und Laden des Spiels in eine eigenständige Assembly ausgelagert werden. Hierbei erhielt das System eigene Signals, über welche die Kommunikation von externen Modulen abläuft. Des Weiteren konnte mit Hilfe von abstrakten Signals das Registrieren einiger Komponenten auf unzählige Events bzw. Signals verringert werden. Durch die zusätzliche Abstraktionsebene werden die einzelnen Komponenten entkoppelt und die Codebasis dadurch weiter modularisiert. Gerade bei Komponenten, welche die gleiche Funktion bei unterschiedlichsten Signals ausführen, findet eine Vereinfachung statt und somit wird die Wartbarkeit deutlich erhöht sowie die Fehleranfälligkeit reduziert.

Die Integrierung eines solchen Frameworks stellt sich als große Herausforderung dar. Person A aus den Interviews spricht sich für eine bereits frühe Integration aus, da hierdurch die Gesamtarchitektur des Systems betroffen ist (Person A, Zeile 109-111). Des Weiteren muss dieses Framework und die damit in Verbindung stehende Architektur, auch von allen im Projekt verstanden und genutzt werden (Person A, Zeile 93-98).

Neben den anfänglichen Einarbeitungszeiten, spielt die Performance auch eine wichtige Rolle. Durch solche größeren Frameworks kommt es zu einer erhöhten Auslastung des Systems. Gerade bei der Serialisierung der Daten treten langsamere Prozesse auf, da hier JSON oder XML als Zwischenformate verwendet werden. Auch rechenaufwändige Reflections kommen zum Einsatz, wodurch Probleme bei einer Portierung auf die Nintendo Switch auftreten können. Ebenso präsentieren sich erhöhte Kompilierungszeiten, durch die Einbindung einer großen Codebasis. (Person B, Zeile 91-106)

Durch die Experteninterviews wurden die erlangten Erfahrungen und Ergebnisse weiter unterstützt. Die Einbindung eines Dependency Injection Frameworks sollte bereits früh in der Projektplanung abgewogen werden. Der zusätzliche Aufwand, eine bestehende Codebasis an ein Framework dieser Größe anzupassen und dadurch wertvolle Entwicklungszeit zu verlieren, steht für kleinere Teams und Projekte in keinem Verhältnis. Ebenso müssen die Teammitglieder mit den Eigenheiten vertraut gemacht werden sowie ggfs. die eigenen Strukturen und Abläufe an das Framework angepasst werden. Hierdurch können neue Erfahrungen gesammelt werden, jedoch auf Kosten von Zeit und Qualität. Somit begründet sich auch die Entscheidung das Framework wieder aus dem Projekt zu entfernen. Dadurch lassen sich die gewohnten Strukturen wieder nutzen und der Fokus liegt klar auf der Entwicklung des Spiels.

Die Einbindung des Frameworks hat alternative Perspektiven auf den Umgang mit Abhängigkeiten aufgezeigt. So ist es unheimlich mächtig, Abhängigkeiten nur einmal definieren zu müssen und dann nahezu mühelos nach belieben einsetzen zu können. Hierdurch werden jedoch die einzelnen Abhängigkeiten weniger hinterfragt und die Codebasis nimmt an Kopplungen zu. Daher ist es umso wichtiger die Abhängigkeiten zwischen einzelnen Assemblies, aber auch innerhalb dieser, im Auge zu behalten und auf ein Minimum zu reduzieren. Die eingesetzten Signals unterstützten diesen Prozess. Hier soll nach ähnliche Strukturen Ausschau gehalten werden, welche gerade die Flexibilität der abstrakten Signals widerspiegeln.

Des Weiteren wäre die Untersuchung alternativer Dependency Injection Frameworks interessant. Extenject stand an oberster Stelle, da es sich nahtlos mit Unity integrieren lässt und aktiv von der Community genutzt und gewartet wird. Andere Lösungen können eine ähnliche Struktur anbieten und sind möglicherweise weniger Komplex und gleichzeitig einsteigerfreundlicher aufgestellt. Solch eine Untersuchung würde aber nicht im Kontext eines bestehenden Projekts geschehen. Hier würde zunächst innerhalb eines kleineren Scopes experimentiert werden und ggfs. dann eine Integration in ein neues Projekt erfolgen.

4.3 Strukturen und Prinzipien

Die Einsetzung des Factory-Patterns wurde durch das Dependency Injection Framework vorgegeben. Ohne diese Vorgabe, hätte das Design Pattern keine Anwendung gefunden. Die erstellten Objekte unterscheiden sich nur in der Größe voneinander. Ebenso ist eine Trennung zwischen der Erstellung und Nutzung der Objekte innerhalb des Codes nicht gegeben. Der zusätzliche Boilerplate-Code gestaltet den Code nur unübersichtlicher.

Durch das Object Pooling wurde eine Verbesserung der Performance sowie eine Reduzierung der Speichernutzung erzielt. Dabei gibt es noch weiteren Spielraum für Verbesserungen. Gerade im Kontext einer erhöhten Nutzung der Objekte im Pool, kann es zu späteren Zeitpunkten im Projekt erneut zu Problemen kommen. Das Aktivieren und Deaktivieren der zugehörigen GameObjects führt zu größeren Performance-Auslastungen. Hier könnten beispielsweise die Objekte lediglich die einzelnen Skripte deaktivieren und das Objekt in der Szene an einen gesonderten Ort ausgelagert werden. Somit ließen sich unnötige Aktivierung bzw. Deaktivierungen vermeiden. Des Weiteren könnte die Hitbox des Angriffs nicht mit den einzelnen Objekten interagieren, um unnötige Kollisionsabfragen zu vermeiden. Bevor dort jedoch weitere Verbesserungen vorgenommen werden, müssen die Probleme zunächst wieder präsenter werden. Gerade in Kombination mit einem Angriff, wird momentan das Spiel für wenige Frames verlangsamt, um einen Hit-Stop-Effekt zu erzeugen. Hierdurch fallen die Performance-Probleme vermutlich weniger stark auf.

Die angewandten Prinzipien fanden an unterschiedlichsten Stellen der Codebasis ihren Einsatz. Dabei konnten sie zur Steigerung der Modularität, Lesbarkeit, Wartbarkeit und Erweiterbarkeit beitragen. Die ursprüngliche Codebasis bediente sich dabei bereits einiger dieser Prinzipien. Diese wurden nun weiter ausgeführt bzw. durch weitere ergänzt. Nun gilt es diese Prinzipien beizubehalten und idealerweise auf den Großteil der Codebasis anzuwenden. Hierdurch lässt sich eine Einheitlichkeit des Codes schaffen, wodurch die Verständlichkeit und Lesbarkeit weiter erhöht wird.

Der Einsatz der richtigen Prinzipien und auch Design Patterns geht oftmals auf Erfahrungswerte zurück. Für Person B lassen sich bereits angewandte Lösungen wiederverwenden und Probleme auch durch Kombinationen dieser Lösungen bewältigen (Person B, Zeile 167-169, 113-115). Bei Person A ergeben sich häufig aus der konkreten Herangehensweise die möglichen Design Patterns, dabei können diese auch durchaus erst während der Umsetzung genauer definiert werden (Person A, Zeile 204-209). Beide Personen sehen die meisten Prinzipien nicht als dogmatisch an (Person A, Zeile 127-129)(Person B, Zeile 129-131). Diese dienen in erste Linie als Guidelines und finden auch in vielen Anwendungsfällen ihren Einsatz. Jedoch sollten diese Prinzipien nicht forciert werden. Es geht in erster Linie darum, eine eigene Struktur aufzubauen und eine Art der Kommunikation mit der Codebasis. Dabei ist das Wichtigste, Konsistent zu bleiben. Solange man sich in der Codebasis zurechtfindet und auch für Außenstehende die Einstiegshürde angemessen bleibt, können die eigenen Strukturen und Prinzipien frei gewählt werden.

4.4 Einordnung

Das Ziel einer robusteren Projektstruktur wurde erreicht. Diese zeichnet sich nun durch klarer definierte Abhängigkeiten zwischen den einzelnen Systemen aus. Ebenso wurde durch den Einsatz der Prinzipien und Design Patterns die Struktur der Codebasis gestärkt. Hier lassen sich durch eine erhöhte Lesbarkeit und Erweiterbarkeit in Zukunft Veränderungen einfacher durchführen. Die verringerte Kompilierungszeit bildet sich durch definierte Assembly Definitionen ab, welche die Codebasis zusätzlich modularisieren.

Der Gegenstand einer skalierbaren Codebasis wurde anhand des Praxisbeispiels näher beleuchtet. Dabei spielen die Abhängigkeiten eine sehr große Rolle. Diese koppeln die einzelnen Komponenten und Systeme miteinander, wodurch Aspekte wie die Lesbarkeit, Erweiterbarkeit oder auch Wartbarkeit betroffen sind. Das Entkoppeln wird von Robert Nystrom wie folgt definiert: „Meiner Auffassung nach bedeutet die Entkopplung zweier Codeabschnitte, dass man den einen verstehen kann, ohne den anderen verstehen zu müssen."(Nystrom, 2015, S. 30). Das Verständnis des Codes spielt hier also eine sehr wichtige Rolle. Dieses Verständnis kann beispielsweise mithilfe von Prinzipien oder Design Patterns verbessert werden, denn diese unterstützen die Aspekte der Lesbarkeit oder Wartbarkeit des Codes. Ebenso werden Abhängigkeiten reduziert, wodurch sich das Nötige Wissen über den Code verkleinert, um Veränderungen vorzunehmen. Robert Nystrom definiert dabei auch das Ziel einer Softwarearchitektur: „Minimierung des Wissens, das man im Kopf haben muss, um Fortschritte machen zu können."(Nystrom, 2015, S. 30). Auch hier steht das Verständnis über den Code im Vordergrund. Die Architektur und im kleineren Maßstab die Design Patterns, können das nötige Wissen weiter reduzieren und somit das Verständnis der Codebasis erhöhen.

Um eine Codebasis skalierbar zu gestalten, sollten die einzelnen Aspekte, wie die Lesbarkeit oder Erweiterbarkeit des Codes, berücksichtigt werden. Hierbei ist der Faktor der Konsistenz ausschlaggebend. Diese unterstützt die angesprochenen Aspekte in vielerlei Hinsicht. Auch die Personen in den durchgeführten Interviews waren sich einig. Für eine skalierbare Codebasis ist die Konsistenz entscheidend (Person A, Zeile 113-120)(Person B, 211-222). Es können noch so viele Design Patterns oder Prinzipien Anwendung finden, wenn die Disziplin fehlt und die Codebasis nicht konsequent geführt wird, ist ein Projekt deutlich schwerer zu skalieren.

Neben der Codebasis, sind auch andere Teile eines Spielprojekts skalierbar. Beispielsweise gibt es bei dem Aufbau der Szenen-Hierarchie in Unity unterschiedliche Vorgehensweisen. Die Testabdeckung oder auch das Aufstellen von Guidelines und Conventions, sind Aspekte die berücksichtigt werden müssen. In dieser Arbeit wurden die genannten Bereiche nicht näher beleuchtet. Vieles davon ist aber eine weitere Art der Kommunikation, wo auch hier wieder die Konsistenz ein wichtige Rolle spielt.

4.5 Fazit

Für mich war diese Arbeit ein Erfolg. Ich konnte Erkenntnisse in einem Bereich sammeln, welcher schon immer mein Interesse geweckt hat. Gerade das Einbinden eines Dependency Injection Framework war besonders spannend. Bisher konnte ich mich nur in der Theorie mit dem Konzept auseinandersetzen. Durch die praktische Umsetzung konnte das Konzept genauer beleuchtet und erste Erfahrungen gesammelt werden. Auch wenn das erste Ergebnis nüchtern ausgefallen ist und das Framework wieder aus dem Projekt genommen wurde, kann ich mich in Zukunft weiter mit der Thematik befassen und ggfs. Alternativen finden.

Die Verknüpfung mit einem eigenen Spielprojekt hat einen großen Unterschied gemacht. Das Spiel Light of Atlantis war für mich ein großer Bestandteil der letzten Jahre. Nun soll das Projekt auch ernsthaft weitergeführt werden. Umso wichtiger war mir die Verbindung zu dieser Arbeit, da die gewonnenen Erfahrung direkt in das Projekt einfließen und gleichzeitig die Arbeiten sehr praxisnah erprobt werden konnten.

Generell konnte ich mir Prinzipien und Methoden aneignen, welche in Zukunft das weitere Vorgehen beeinflussen werden. So schreckte ich vor Assembly Definitionen immer zurück. Jetzt kann ich mir ein Projekt ohne diese nicht mehr vorstellen. Der Mehrwert für die Struktur der Codebasis ist unermesslich. Ebenso verstehe ich nun immer mehr wie wichtig lesbarer Code und die Reduzierung von Abhängigkeiten ist. Bei jeder Änderung an der Codebasis oder bei jeder Umsetzung eines Features müssen zunächst die betroffenen Systeme verstanden werden. Es macht einen riesen Unterschied, wenn man im Team arbeitet oder auch nach längerer Zeit seinen eigenen Code zu Gesicht bekommt und man erkennt sofort ein strukturiertes Vorgehen wieder. Der Code liest sich wie ein gutes Buch und man kann diesen bereits nach kurzer Zeit verstehen.

Die größte Erkenntnis ist für mich das disziplinierte und konsequente Vorgehen. Das bezieht sich natürlich auf die Codebasis. Diese sollte an jeder Stelle die gleiche Handschrift tragen. Das bedeutet auch, dass die hier teilweise angewandten Methoden und Prinzipien jetzt weitergeführt werden sollen und auf den Reste der Codebasis angewandt werden. Aber das konsequente und disziplinierte Vorgehen findet auch Anwendung in vielen anderen Bereichen der Spieleentwicklung. So sollten beispielsweise die Strukturen innerhalb der Spiel-Engine oder dem Projektmanagement einen roten Faden haben und vor Allem Konsistenz mit sich bringen.

Abbildungsverzeichnis

Literatur

David Thomas, Andrew Hunt (2019). *The Pragmatic Programmer: Your journey to mastery, 20th Anniversary Edition*. Addison-Wesley Professional

Marcel Wiessler, hybridherbst (2023). *Compilation Visualizer for Unity*. URL: https://github.com/needle-tools/compilation-visualizer. Abrufdatum: 01.11.2023

Martin, Robert C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education

Martin, Robert C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson Education

Microsoft (2022). *Code metrics values*. URL: https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2019. Abrufdatum: 01.11.2023

Modest Tree Media (2023). *Extenject*. URL: https://github.com/Mathijs-Bakker/Extenject. Abrufdatum: 01.11.2023

Nystrom, Robert (2015). *Design Patterns für die Spieleprogrammierung*. mitp Verlag

Preston-Werner, Tom (o. D.). *Semantic Versioning*. URL: https://semver.org/lang/de/. Abrufdatum: 01.11.2023

Unity Technologies (2023a). *Assembly definitions*. URL: https://docs.unity3d.com/Manual/ScriptCompilationAssemblyDefinitionFiles.html. Abrufdatum: 01.11.2023

Unity Technologies (2023b). *C-Sharp Compiler*. URL: https://docs.unity3d.com/2022.3/Documentation/Manual/CSharpCompiler.html. Abrufdatum: 01.11.2023

Unity Technologies (2023c). *Unity 2022 LTS for programmers*. URL: https://unity.com/releases/lts/programmers#harness-enhanced-2d-development. Abrufdatum: 01.11.2023

Unity Technologies (2023d). *Upgrading Unity*. URL: https://docs.unity3d.com/Manual/UpgradeGuides.html. Abrufdatum: 01.11.2023

Wilmer Lin Thomas Krogh-Jacobsen, Peter Andreasen Scott Bilas (2022). *Level up your code with game programming patterns*. Unity Technologies

Anhang A

Ohne weitere Assembly Definitions

			Editor-Script					Nicht-Editor-Script			
	Total	Compilation	Csc	Reload		Total	Compilation	Csc (Assembly-CSharp)	Csc (Assembly-	Csc - Total	Reload
	4,849	1,13	0,243	3,719		5,852	2,02	0,355	0,242	0,597	3,832
	4,918	1,15	0,242	3,768		5,849	2	0,35	0,245	0,595	3,849
	4,878	1,15	0,24	3,728		5,879	2,03	0,351	0,236	0,587	3,849
	5,069	1,36	0,246	3,709		5,649	1,94	0,367	0,24	0,607	3,709
	4,985	1,26	0,263	3,725		5,735	2,01	0,362	0,242	0,604	3,725
	4,84	1,06	0,24	3,78		5,62	1,84	0,353	0,241	0,594	3,78
	4,788	1,06	0,24	3,728		5,728	2	0,351	0,241	0,592	3,728
	4,836	1,06	0,251	3,776		5,796	2,02	0,36	0,233	0,593	3,776
	4,871	1,06	0,231	3,811		5,861	2,05	0,358	0,306	0,664	3,811
	4,917	1,16	0,242	3,757		5,767	2,01	0,349	0,237	0,586	3,757
Durchschnitt:	**4,8951s**	**1,145s**	**0,2438s**	**3,7501s**	**Durchschnitt:**	**5,7736s**	**1,992s**	**0,3556s**	**0,2463s**	**0,6019s**	**3,7816s**
Median:	**4,8745s**	**1,14s**	**0,242s**	**3,7425s**	**Median:**	**5,7815s**	**2,01s**	**0,354s**	**0,241s**	**0,5945s**	**3,778s**

Mit definierten Assembly Definitions

			Editor-Script					Nicht-Editor-Script			
	Total	Compilation	Csc	Reload		Total	Compilation	Csc (Assembly-CSharp)	Csc (Assembly-	Csc - Total	Reload
	4,71	0,59	0,15	4,112		4,92	0,73	0,253	0,115	0,39	4,189
	4,65	0,59	0,15	4,059		5,09	0,83	0,25	0,122	0,39	4,256
	4,61	0,5	0,13	4,104		4,84	0,63	0,26	0,117	0,4	4,213
	4,68	0,59	0,13	4,091		5,21	0,95	0,251	0,116	0,38	4,267
	4,62	0,44	0,14	4,175		4,95	0,73	0,253	0,116	0,38	4,215
	4,68	0,6	0,12	4,086		4,84	0,64	0,252	0,117	0,39	4,204
	4,7	0,59	0,13	4,097		4,94	0,74	0,246	0,118	0,39	4,21
	4,68	0,6	0,12	4,069		4,85	0,64	0,245	0,113	0,38	4,21
	4,7	0,59	0,12	4,114		4,83	0,63	0,248	0,113	0,38	4,194
	4,8	0,6	0,14	4,206		5,08	0,84	0,252	0,115	0,39	4,239
Durchschnitt:	**4,683s**	**0,569s**	**0,133s**	**4,1113s**	**Durchschnitt:**	**4,955s**	**0,736s**	**0,251s**	**0,1162s**	**0,387s**	**4,2197s**
Median:	**4,68s**	**0,59s**	**0,13s**	**4,1005s**	**Median:**	**4,93s**	**0,73s**	**0,2515s**	**0,116s**	**0,39s**	**4,2115s**

Person A - Transkript

Interviewer

Dann fangen wir auch direkt mit der ersten Kategorie an, nämlich Assembly Definitionen. Kannst du mir vielleicht sagen, inwiefern diese bei dir in den Projekten Anwendung finden.

Person A

Ja, bei mir ist der gesamte Code in Assembly Definitionen untergebracht. Ich habe in meinem Fall eigentlich eine Assembly für den gesamten Runtime Game Code, eine für Editor Scripts und Unity Tests. Ich habe eine hauseigene Utility Library über den Package Manager, eine im Projekt selber, aber die ist tatsächlich in der Runtime Assembly mit drin, habe ich noch nicht extrahiert, könnte man mal machen. Eine für Playmode Tests, eine für manual Tests, also interaktive rumspiel-Tests und natürlich all die Libraries, die ich importiere. Ich mach das auch so, wenn ich irgendwelche Packages importiere mit Skripten, die ich benutzen will, kommen die halt auch in eine Assembly.

Interviewer

Genau, weil tatsächlich echt häufig, wenn du externe Packages importierst, kommen die häufig nicht mit Assembly Definitionen, was ein bisschen schade ist.

Person A

Ja, das ist sehr schade und sehr nervig, denn ich will eigentlich Packages nicht anfassen, weil das einfach das Updaten und Warten der Dinger so unangenehm macht. Viele sind natürlich älter. Unity hat die Assembly Definitionen ja auch erst irgendwann später eingeführt. Ich weiß nicht mehr, wann. Ich glaube auf jeden Fall, bevor ich mein letztes großes Projekt aufgesetzt habe. Muss also länger als 5 Jahre her sein, aber noch keine 10, irgendwo dazwischen.

Interviewer

Ich weiß es auch nicht genau, aber ja genau, also ältere Packages, denen sei es verziehen. Aber gerade bei den Neuen sollte das der Standard sein, dass wenn ich sie integriere, sie auch ihre eigenen Assembly Definitionen mitbringen. Kannst du mir vielleicht sagen, was sie konkret für dich in den Projekten so an Mehrwert erbringen?

Person A

Oh ja, also du hast ja schon die Kompilierungszeiten erwähnt, die werden ein bisschen kürzer. Allerdings sind die jetzt auch nicht unendlich lang und dagegen steht natürlich der Aufwand, dass man die Assembly Definitionen selbst verwalten muss, was ja auch Zeitaufwand bedeutet. Gut sei es drum. Ja, die Kompilierungszeiten sind gut, aber für mich ist eigentlich viel charmanter daran, dass der Code wirklich sauber separiert ist. Also Namespaces vernünftig getrennt sind und man auch einfach immer genau weiß, was man braucht und referenziert und was nur benutzt wird. Dadurch vermengen sich Dinge einfach weniger stark und die einzelnen Aufgaben sind sauberer getrennt.

Interviewer

Jetzt sagst du, dass du die Grenze bei der Unterteilung von Runtime und Editor ziehst. Könntest du dir vorstellen deine eigene Codebasis noch weiter zu unterteilen?

Person A

Ja, also ich habe das Gefühl es könnte sich lohnen, wenn das alles Game Code ist, der projektspezifisch ist. Ja, das könnte man machen. Das fühlt sich für mich nach zu viel Overhead an, für relativ wenig Mehrwert, weil der Projekt-Code meistens ja einfach dadurch, dass es das gleiche Spiel ist, doch eigentlich immer sehr eng verzahnt ist. Da weiß ich nicht wirklich, gibt es da Teilbereiche, die völlig isoliert leben, von was anderem und trotzdem projektspezifisch sind? Wenn es eine Utility ist, Pathfinding zum Beispiel, das kann man ja

extrahieren und sagen, das ist ein eigenes Ding. Aber wenn man zwei Teile in seiner Codebasis hat, die beide sehr spezifisch für das Spiel sind, aber trotzdem so separat, dass sie sich vielleicht nicht gegenseitig referenzieren müssen. Weiß ich nicht. In den Projekten von mir gibt es glaube ich sowas nicht unbedingt.

Interviewer

Gehen wir weiter von Assembly Definitionen auf das Dependency Injection Framework. Da würde mich auf jeden Fall interessieren, auf welche Arten du momentan Abhängigkeiten in deinen Projekten bereitstellst.

Person A

Ja, das ist ein bisschen ein schmerzhafter Punkt, weil es da mit Unity immer schwierig ist. Unity macht es einen teilweise sehr schwierig und dann wiederum sehr einfach in einigen Bereichen, dadurch, dass es zu viele Möglichkeiten gibt, von denen einige in bestimmten Fällen einfach gar nicht oder sehr schlecht funktionieren. Und es endet damit, dass man mehrere Verfahren in seinem Projekt benutzt. Also ich habe mehrere. Aber die Folgen einem gewissen Konzept, sag ich mal. Die ganzen Referenzen auf andere Komponenten innerhalb eines Prefabs oder zwischen Objekten in einer Szene, sind bei mir eigentlich immer Drag and Drop als Unity-Referenzen in SerializedFields. Das ist deshalb gemacht, weil die Alternative ja irgendeine Runtime-Zuweisung ist. Das hat natürlich gegebenenfalls Vorteile für das Testing . Aber das heißt auch, dass es immer eine Initialisierung geben muss, bevor die Komponenten funktionieren. Also typischerweise lauter GetComponent-Calls in Awake. Und das macht die Testbarkeit auch wieder schwieriger, weil die Skripte an einem GameObject jetzt im Edit-Mode oft nicht benutzt werden können. Also manchmal will ich eine Preview von irgendeiner Visualisierung-Komponente erzeugen in Edit-Mode, geht aber nicht, wenn sie die Referenzen nicht hat. Außerdem sind es auch Szenen-Ladezeiten, die damit einhergehen oder unter Umständen kleine Glitches, die kommen, wenn du Objekte instanziierst und diese erstmal 100 GetComponent-Aufrufe machen. Und ich habe momentan ein VR-Projekt und da sind die Frame-Zeiten halt sehr kurz. Man braucht diese stabilen 72 Frames pro Second, die eigentlich nie unterboten werden dürfen. Und dann ist das halt auch kritisch, wenn sich das auf die Performance auswirkt. Deswegen habe ich geschaut was mache ich, dass wirklich am wenigsten Runtime-Performance-Kosten hat und daher diese Realisierung über den Unity-Inspektor. Problematisch wird es bei verschachtelten Objekten. Dort die Zuweisungen von Hand zu machen, kostet einfach sehr viel Zeit. Es gibt noch ein paar weitere Referenzierungsmethoden, die ich so habe. Manchmal nutze ich ScriptableObject-Assets, die auch GameObjects referenzieren können, um beispielsweise Cross-Scene-References zu ermöglichen. Das hat auch Vor- und Nachteile. Man muss sicherstellen, dass die Referenzen up-to-date sind, aber dafür kann man viel damit machen. Singletons benutze ich auch. Dann benutze ich viele Properties, wenn ich die Zuweisung im Unity-Inspektor mache, wodurch ich viel Boilerplate-Code spare. Dann benutze ich einen custom Dependency Injection Service Provider. Den benutze ich für eine Handvoll Sachen. Den habe ich eingebaut, ursprünglich um mein Projekt testbar zu machen, weil ich die Prämisse hatte, dass ich versuchen will, eine saubere Testabdeckung zu haben für die wichtigsten Sachen. Ist unterwegs ein bisschen eingeschlafen, muss ich ganz ehrlich sagen, aber Service Provider ist trotzdem recht nützlich. Das benutze ich aber nur für reine C-Sharp-Klassen, die keine GameObjects sind.

Interviewer

Also gerade der letzte Punkt klingt ja schon so ein bisschen auch nach Dependency Injection Framework, auch wenn es ein Service Provider ist. Ab welcher Größe eines Unity-Projekts würdest du sagen, könnte sich ein größeres Framework für die Dependency Injection eignen? Hast du da schon Erfahrungen gemacht, wo du sagst, ab einer gewissen Größe könnte sich das schon lohnen?

Person A

Also ich habe nie ein externes Tool wie zum Beispiel Zenject (Extenject) für Unity ausgiebig angeguckt, aber so groß ist das ja auch wieder nicht. Also mein Service Provider ist super schlank, wirklich nur eine kleine Klasse, die ein Paar Default-Implementationen beinhaltet. Ich habe das Gefühl, dass lohnt sich ja eigentlich von Anfang an. Weil es ein paar Sachen in Unity gibt, die ja statisch sind. Sowas wie Time.DeltaTime und das kann einem beim Entwickeln schon echt oft in die Beine grätschen, dass man darüber gar keine Kontrolle hat, weil man das jetzt testen will oder simulieren will. Weil man einfach konsequent überall statt der Time-Klasse

seinen Time-Service benutzt, wodurch man das halt kontrollieren kann. Und das macht einen das Leben oft erheblich viel einfacher. Und so kompliziert müssen die ja nicht zu benutzen sein. Aber das Problem, dass man natürlich hat, es ist eine weitere maintainance-bedürftige Geschichte, die auch jeder in dem Projekt verstehen muss. Ich habe mal ein größeres Softwareprojekt gehabt mit über 100 Leuten und da hatten wir natürlich Dependency Injection. Und die damit zusammenhängenden Bugs sind oft sehr schwierig zu finden. Zum Beispiel, wenn wir jetzt eine falsche Implementation für irgendein Interface benutzen, kann das sehr nervig sein, herauszufinden was schief gegangen ist. Es ist also eine weitere Fehlerquelle.

Interviewer

Genau das wäre auch gleich die nächste Frage, welche Schwierigkeiten dabei auftreten können, wenn man sowas nutzt und auch gerade in größeren Projekten. Gibt es vielleicht andere Schwierigkeiten, die du damit identifizieren kannst?

Person A

Also ich habe es jetzt nicht in dem Kontext, also so weitreichend auf GameObject-Ebene genutzt. Also stelle ich mir ein bisschen mühselig vor, gerade wenn es Referenzen innerhalb eines Prefabs sind. Was genau ist im Unity-Inspektor exposed, also kann ich zur Laufzeit dann noch feststellen, welche Werte die einzelnen Referenzen haben? Wie sieht es mit der Reihenfolge der Referenzen aus? Muss ich mir da Gedanken machen? Aber die Frameworks sind ja auch irgendwo im produktiven Einsatz und durchgetestet, also wird sicherlich schon seinen Sinn haben. Ich glaube aber, wenn man das macht, dann sollte man das wahrscheinlich von Anfang an versuchen, in seinem Projekt zu haben, weil das weitreichende Konsequenzen hat für die Gesamtarchitektur. Am Anfang kann es dann wie Overkill wirken und einen vielleicht ein bisschen in der Entwicklung verlangsamen. Aber ich glaube, nach hinten raus kann sich das dann lohnen, wenn man das konsequent durchzieht. Was immer wichtig ist in einem skalierbaren Unity-Projekt, ist Konsistenz. Es gibt für alle Probleme in der Spielentwicklung 10000 Lösungen und das Schlimmste, was man haben kann, ist gar nicht die falsche Lösung, sondern zu viele Lösungen. Wenn ich ein neues Feature implementieren muss und ich mich in Teile von dem Projekt wieder reinfuchsen muss, weil ich das vor Ewigkeiten gemacht habe oder jemand anders gemacht hat, aber ich jedes Mal wieder neu verstehen muss, wie alles zusammenhängt und sich referenziert und funktioniert, weil es anders ist als im Rest des Projekts, dann hält das einfach auf und es erzeugt andauernd Fehler. Also das ist glaube ich das Allerwichtigste, dass wir einfach versuchen, konsequent zu sein.

Interviewer

Dann springen wir kurz zu den Prinzipien. Jetzt ging es ja gerade auch um die Skalierbarkeit eines Unity-Projekts. Und Konsistenz ist dort ein gutes Stichwort. Gibt es vielleicht noch andere Sachen, wo du sagst, das muss auf jeden Fall in einem größeren Unity-Projekt drin sein, damit es skalierbar bleibt?

Person A

Die professionelle Software-Entwicklung habe ich eigentlich erst im Berufsalltag kennengelernt. Und da habe ich dann auch ein paar Prinzipien kennengelernt. Die sind für mich aber nicht so dogmatische Prinzipien, die ich irgendwann mal gelernt habe und mit denen ich dann aktiv in die Entwicklung reingehe, sondern im Prinzip sind das die Prinzipien, die ich in meiner Unity-Entwicklung eher über die Jahre mir angeeignet habe. Zum Beispiel sowas wie Don’t Repeat Yourself. Aber dabei darf man es auch nicht übertreiben. Also ich weiß noch immer bei all diesen Fragen um Prinzipien. Allein das Wort Prinzipien sorgt schon dafür, dass die Leute sich streiten, was sind denn jetzt gute, falsche, richtige oder verkehrte Prinzipien? Was muss man unbedingt machen, was darf man auf keinen Fall machen? Und da gibt es eigentlich nie eine richtige Antwort drauf glaube ich. Ich kann nur sagen, was für mich irgendwie gut funktioniert hat oder was mir viel geholfen hat. Ich versuche Abstraktionsebenen zu vermeiden, wenn sie nicht notwendig sind. Typischerweise mache ich das mehr als ich sollte, glaube ich, um eben mich nicht zu viel zu wiederholen. Das hier funktioniert gleich, es muss auf jeden Fall zusammengeführt und abstrahiert werden. Aber ich glaube das ist oft eigentlich nicht unbedingt eine gute Idee, weil das von der Wartung her schwierig ist und Unity nicht besonders gut mit Abstraktionsebenen umgeht. So mit abstrakten Klassen und Interfaces ist Unity ja manchmal ein bisschen

schwierig. Man kann keine Interface-Referenzen serialisieren zum Beispiel. Und wie gesagt, Konsistenz ist super wichtig. Man kommt immer wieder irgendwo hin und man könnte etwas anders und besser machen, aber dann wäre es anders als woanders im Projekt und allein das ist ein Nachteil und das muss man sich bewusst machen, dass man nicht überall versucht zu optimieren, bis es für diesen Teil richtig ist, sondern das ganze Projekt irgendwie im Kopf hat dabei. Ich versuche immer meine Skripts superklein zu machen. Die machen eine Sache. Das kann man dann sehr stark separieren. Ich versuche alles auseinanderzuziehen so gut es geht in den einzelnen Skripten, auch wenn ich dann oft gar nicht so viel wiederbenutze, wie man es denken würde. Aber es ist trotzdem gut, die Dinger zu trennen, weil es einfach einfacher ist mit der Codebasis zu arbeiten, wenn man kleiner Skripte hat, und man sieht auf einen Blick, was diese tun. Ich schreib so gut wie gar keine Kommentare. In meiner Erfahrung, wenn du Kommentare an deinen Code schreibst, dann hast du vorher was falsch gemacht. Dokumentation ist gut, also wenn du schreibst, was die Methode macht. In deinen Algorithmus einen Kommentar reinschreiben, dann muss daran irgendwas falsch sein und dann schreibe ich das neu und benenn das richtig und extrahiere Methoden vernünftig und benennen die Methoden richtig, damit der Code sich selbst erklärt. Nichts ist schlimmer als irgendwelche kryptischen Variablen, wo man dann den ganzen Algorithmus verstehen muss oder irgendwelche Kommentare lesen muss, damit man weiß, was die überhaupt bedeuten sollen. Also ich benenne meine Variablen dann auch oft recht ausführlich. Man kann seine Variablen und seine Methoden vernünftig benennen, dann braucht man auch keine Kommentare. Additive Szenen sind super, also alles, was isoliert existieren kann, wo alle Referenzen intern abgehandelt sind, kommt in eine separate Szene. Mein Projekt hat so etwa 10 Szenen in der Regel parallel geladen. ScriptableObjects sind super. Damit kann man prima auch so Daten von Code separieren. So Datenpakete, also zusammengehörige Daten, die irgendwelche Objekteigenschaften beinhalten, kann man in einem ScriptableObject sammeln und allein dadurch klarstellen, diese Dinger gehören zusammen. Ich glaube, es ist wichtig, dass man sein Projekt so aufsetzt, dass man die Sachen, die man macht, superschnell testen kann. Ohne, dass man jetzt das ganze Spiel starten muss. Also die Prefabs sollten möglichst alleine funktionieren. Also ich glaube es immer gut, wenn man es schafft, dass man ein Prefab in eine leere Szene ziehen kann und nicht die Konsole mit Error-Messages gefüllt wird. Überhaupt sorge ich dafür, dass wenn man das Projekt aufmacht, keine Errors oder Warnings auftreten. Die haben ja einen Grund. Nichts ist schlimmer als irgendwelche Assets, die in ihrem Code irgendwelche Warnings rausschmeißen im Normalbetrieb. Generell sollte man auch dafür sorgen, dass die Unity Engine selbst, also der Editor immer schnell läuft. Das raubt einem Nerven, Motivation, Produktivität und alles, wenn der Unity-Editor langsam ist. Der Editor muss immer schnell laufen und wenn man irgendwann das Gefühl hat, der Editor läuft zu langsam, dann sollte man was tun. Das merkt man immer dann, wenn man mal ein neues kleines Projekt wieder aufmacht und wenn man was nachgucken will. Das läuft normalerweise in unter einer Sekunde, aber wenn ich das bei mir in meinem Projekt mache, dann dauert es noch 5 Sekunden länger und dann vermeidet man das zu machen. Stattdessen sollte man dann vielleicht mal den Profiler anschmeißen und versuchen rauszufinden, was eigentlich so lange dauert. Und vielleicht wird man das los. Dann, wenn man so ein Projekt hat, das immer größer wird, stellt man irgendwann fest, dass man die gleichen Sachen immer wieder tut und dann sollte man sich dafür Tooling bauen. Das vergisst man oft. Man muss seine Routinen, die man so hat, beobachten und sich überlegen, muss ich das eigentlich wirklich immer manuell machen, kann ich das Automatisieren oder kann ich da zumindest ein Tool draus machen? Und superwichtig: Versionskontrolle und saubere, vernünftige Commits. Und immer, wenn man was committet, einmal reingucken, was für Änderungen man da gerade eigentlich committet und ob man das auch alles so meint. Das hilft auch, wenn man da nochmal so durchgeht. Was habe ich eigentlich geändert? Dann findet man. Ah ja, verdammt, hier hatte ich einen Log eingebaut, das wollte ich ja wieder rausnehmen. Oder, ah ja, hier hatte ich kurz etwas deaktiviert, das sollte ja aber gar nicht so bleiben. Sowas hält das Projekt dann sauber über einen langen Zeitraum. Das sind so meine wichtigsten Sachen, die ich durchziehe und mal in anderen Teams gesehen habe.

Interviewer

Wie sieht es bei dir aus mit Code Style Guides? Also ich mein du arbeitest ja wahrscheinlich jetzt gerade an deinem Projekt noch größtenteils alleine, was das Programmieren angeht, aber existiert so etwas als Dokument oder machst du das größtenteils nach Gefühl?

Person A

Ich benutze einen Code Formatter, der einfach ein Regelwerk festhält und das wird konsequent durchgezogen. Mein Code ist dadurch immer gleich. Ich möchte auf einen Blick möglichst viel sehen, was passiert. Deswegen mag ich das nicht so gerne, wenn es irgendwie 3 nested Codeblöcke gibt, und man den Überblick verliert. Sowas wäre, wenn man mehrere Leute im Team hat, miteinander abzusprechen. Dann sollte man sich auch gegenseitig immer den Code zeigen, vielleicht so Reviews machen, das Mal besprechen, wenn man da Sachen fundamental verändert und dann vielleicht sogar Guidelines festhalten. Nicht alles kann man bei einem Code Formatter abbilden.

Interviewer

Dann kommen wir noch einmal zu einem letzten Punkt, und zwar Design Patterns. Das hatten wir eben auch schon mal so ein bisschen angerissen, so Designpatterns. Das sind die Lösungen für vielerlei Softwareprobleme, aber da würde ich dich einfach gern mal fragen, zu welchem Zeitpunkt entscheidest du dich so ein Design Pattern dann einzusetzen?

Person A

Reine Gefühlssache. Ich fange oft an, meinen Code einfach erstmal stumpf am Stück wegzuschreiben. Und dann irgendwann merkt man ja, dass man sich permanent wiederholt oder dass man Strukturen erzeugt hat, die man gar nicht wollte. Ab und zu muss man halt einen Schritt zurückgehen und sich überlegen, was für Strukturen habe ich hier eigentlich erzeugt mit meinem Code. Und dann hilft es ja oft, einfach so ein Design Pattern dann anzuwenden. Das macht auch den Code einfach lesbarer, weil es dem ganzen halt eine Struktur gibt, die sowieso da ist, aber halt nicht sichtbar ist. Mit meinem vorherigen Statement, das Konsistenz so wichtig ist, da muss man immer dagegen abwägen, dass man die Sachen ja auch nicht übertreiben muss. Es muss ja nicht alles immer gleich volles Rohr, alles auspacken, einfach nur, damit es Konsistent ist. Also wenn ich irgendein Design Pattern wähle, dann breche ich das vielleicht trotzdem auch manchmal noch mal wieder auf. Ich weiß nicht so genau, das ist eine sehr abstrakte Frage. Es ergibt sich einfach irgendwie immer so ein bisschen, man muss sich fragen, was will ich mit meinem Spiel am Ende denn eigentlich erreichen.

Interviewer

Genau, ich glaube darauf zielt die Frage halt auch so ein bisschen ab. Design Patterns finden halt häufig Anwendung, man muss aber trotzdem vielleicht erstmal abwägen, lohnt sich das überhaupt? Und du hast ja jetzt auch schon gesagt, für dich ist halt wieder Konsistenz ein wichtiges Stichwort.

Person A

Ja, man hat auch einfach so bestimmte Patterns, die man einfach nicht ertragen kann. Weil man sich denkt, wenn das hier an 3 Stellen wiederholt wird und ich da irgendwann mal was ändern will, dann muss ich das an 3 Stellen machen, da muss ich jetzt auf jeden Fall extrahieren und eine Methode wählen. Dass geht aber nicht wirklich, weil die jeweils irgendwie da drin sind, so sehr verzahnt sind. Und dann muss ich da noch irgendwelche zusätzlichen Ebenen einführen, damit ich das überhaupt extrahieren kann. Und dann fragt man sich, ist es wirklich so schlimm, wenn ich mich jetzt einmal wiederhole? Riskiere ich, dass ich das noch 30 mal schreibe oder bleibt es bei diesen 3 Malen? Also, sich wiederholender Code hat halt auch seine Vorteile. Das einem die Flexibilität gegeben wird auch mal ein wenig zu variieren. Und sobald man es aber extrahiert, müssen auch alle Aufrufe exakt gleich funktionieren und dann findest du wieder einen, bei dem das aber wirklich nicht geht, weil der ja doch ein bisschen anders ist und der ist dann wieder eine Ausnahme und dann hast du eigentlich gar nichts von deinem Mehrwert, dadurch dass du da dieses Pattern drauf losgelassen hast. Aber alle Kosten. Also du musst es warten, es ist unübersichtlicher, weil das vielleicht irgendwelche zusätzlichen Abstraktionsebenen hat.

Interviewer

Es ist ein ständiges Hin und Her. Das wird man auch nie loswerden. Das Projekt wächst halt und dann werden immer wieder diese Stellen und diese Fälle auftreten und dann muss man halt immer wieder individuell

abwägen. Habe ich jetzt Ressourcen dafür? Habe ich die Zeit dafür? Und ist es wirklich so schlimm? Irgendwo muss man halt Abstriche machen.

Person A

Genau, es ist nie alles perfekt. So ein Projekt lebt und entwickelt sich, die Anforderungen verändern sich. Und das heißt einfach, dass man auch regelmäßig mal in irgendeiner Form aufräumen muss. Spieleprojekte werden mit der Zeit auch immer langsamer, weil sich die Sachen immer mehr miteinander verzahnen und jede Änderung immer Konsequenzen hat, die man alle berücksichtigen muss.

Interviewer

Ja, so wie du sagst, die Projekte wachsen halt. Da muss man in regelmäßigen Abständen mal schauen, wo kann ich vielleicht an den Stellschrauben ein bisschen drehen, damit ich da wieder ein bisschen mehr Performance rausholen kann, damit ich da ein bisschen weiter aufräume, ein bisschen mehr Konsistenz wieder reinbringe, so dass mir das wiederum passt und ich wieder normal weiterarbeiten kann, bis dann irgendwann mich das wieder einholt und dann dieser Ablauf wieder von vorne beginnt. Vielen, vielen Dank für das Interview. Dann stoppe ich jetzt die Aufnahmen.

Person B - Transkript

Interviewer

Und dann fangen wir auch direkt an mit der ersten Kategorie, wo es so ein bisschen um Assembly Definitionen gehen soll. Und da würde mich einfach mal interessieren, inwiefern du in deinen Projekten Assembly Definitionen anwendest.

Person B

Ja, also für mich ist es so. Es gibt keinen Code ohne Assembly Definition. Also es gibt keine Skripte, die einfach so im Projekt rumliegen, sondern die haben immer einen Namespace, die haben immer eine Definition. Und ich gehe sogar so weit, dass wenn ich Third-Party-Packages importiere oder irgendwelche anderen Packages habe, die nicht mit Assemblies arbeiten, dann setze ich von Hand Assembly Definitionen drauf. Hauptsächlich, um halt zu schauen, dass die Abhängigkeiten klar sichtbar sind und dass da nichts ineinander wächst. Also klar, man hat halt auch so ein bisschen technische Vorteile, wie jetzt, dass sich die Kompilierungszeit deutlich reduziert, weil nicht immer alles neu gebaut werden muss sondern immer nur der Abhängigkeitsbaum, der neu gebaut wird, je nachdem an welcher Stelle der Code geändert wurde. Aber der größte Vorteil ist, finde ich, dass man relativ gut Editor-Code von Debug-Code vom Testing-Code von Runtime-Code trennen kann. Was häufig der Fall ist, dass das halt eben im Laufe des Projekts sich sonst sehr schnell verwäscht. Und was ich überhaupt nicht leiden kann, sind halt Ausnahmen über Präprozessoranweisungen. Das sehe ich sehr ungerne, da habe ich lieber den Code tatsächlich dann in einer eigenen Editor Assembly und das macht zwar manchmal Sachen schwieriger, also vor allem wenn man sich dann mit dem Odin-Inspektor auseinandersetzt. Der kann das auch, dass man über den Editor Code erweitert, ohne dass es in dem Runtime-Skript drin liegt. Aber ein Vorteil ist tatsächlich, dass man nicht so häufig kaputte Builds bekommt. Des Weiteren lassen sich dadurch die Abstraktionen zwischen den Packages halt gut aufrechterhalten. Einfach um einerseits klar definiert für mich zu haben, wie ich eine Library benutze. Also um es zum Beispiel mir einfacher zu machen oder deren Verwendung zu vereinfachen oder zu bündeln, aber auch gleichzeitig, weil ich halt weiß, dass jetzt, wenn man länger dabei ist, leider Libraries auch veralten, nicht mehr gewartet werden, irrelevant werden. Dass man die leicht wieder austauschen kann, also dass man die auch leicht wieder loswird oder halt durch eine andere ersetzen kann. Genau, und da helfen diese Assembly Definitionen halt wahnsinnig gut.

Interviewer

Ok, bei den externen Packages hast du dann eine Assembly Definition, wo alle externen Packages reinkommen, die keine eigene mitbringen oder gehst du dann wirklich in das Package rein und definierst eine eigene für jedes Package?

Person B

Letzteres. Mehrere Assembly Definitionen dann auch meistens. Also wenn ich feststelle, dass ich das für ein Package machen muss, dann ist das auch schon ein Grund, das Package nicht zu benutzen. Und wenn ich feststelle, dass ich nicht einfach sagen kann du bekommst jetzt mehrere Assembly Definitionen, sondern es noch komplizierter wird, dann ist das Package nicht gut geschrieben, das hat keinen guten Code, damit will ich nicht arbeiten und dann fliegt es auch wieder raus.

Interviewer

Du sagtest, du teilst das dann natürlich auch so ein bisschen auf, zwischen Editor und Runtime für den Spielcode. Gibt es innerhalb der Runtime dann auch mehrere Assembly Definitionen?

Person B

Sehr viele. Also ich habe für quasi jeden sub-namespace eine eigene Assembly Definition. Das geht so weit, dass ich teilweise 5 bis 10 Skripte pro Assembly Definition habe. Und es geht halt nicht nur um Runtime oder Editortime. Das wird ja komplizierter übers Unit-Testing. Es gibt ja dann noch so quasi den Testing-namespace, dann gibt es den Debug-namespace, dann gibt es verschiedene Abstaffelungen an compile flavors die ich habe.

Also baue ich jetzt eine Version, die für den Release gedacht ist, baue ich eine Version, die fürs interne Testing gedacht ist, baue ich eine Version mit oder ohne Cheats. Und all diese Sachen möchte ich nicht nur deaktivieren, sondern ich möchte sie zum Beispiel über die Assembly Definitionen komplett exkludieren. Also die sollen nicht mal annähernd in meinem Code vorkommen.

Interviewer

Dann komm wir einmal zum nächsten Punkt. Dabei handelt es sich um Dependency Injection Frameworks. Da würde mich generell erst mal interessieren, wie du so im Allgemeinen die Abhängigkeiten in deinen Projekten auflöst, also wie werden diese bereitgestellt?

Person B

Ich bin eher darauf bedacht, eigentlich so viel wie möglich über den Unity-Inspektor zu machen. Also Ich bin der Meinung, dass man direkte Zuweisungen machen sollte, um auch eine klare Vorstellung davon zu haben, dass eine Abhängigkeit besteht. Und um sicherzugehen, dass man keine Abhängigkeiten verschleiert. Ein Dependency Injection Framework lädt auch dazu ein, dass man dann denkt, ja, das hängt ja gar nicht zusammen, oder der Code gehört nicht zusammen. Oder noch besser, Unity selbst checkt gar nicht, dass der Code in das Projekt gehört und wirft das direkt raus. Deswegen bin ich ein Freund davon, wenn Sachen erkennbar miteinander verknüpft sind. Und ich mach das hauptsächlich über den Unity-Inspektor. Vor allem über Serialized References, wenn ich etwas entkoppeln möchte. Weil man über Serialized References halt auch Interfaces realisieren kann und ich ein sehr großer Freund von Interfaces bin und das Entkoppeln über Interfaces zu betreiben. Was sich dann für mich so äußert, dass ich sage, ich erwarte hier halt meine Schnittstelle und welche Klasse die dann tatsächlich erfüllt, mache ich über die Zuweisung. Dazu muss man sagen, dass das nur funktioniert, wenn man auch Odin im Projekt hat, da Unity mit dem Workflow nicht so gut klarkommt. So Sachen wie Scriptable Objects benutz ich auch noch ganz gerne, aber bei Scriptable Objects muss halt klar sein, dass die Abhängigkeit, die ich im Projekt habe, quasi schon im Projekt ist. GameObject.Find ist auf Todesstrafe verboten. Das gibt es in meinen Projekten nicht, auch nicht als Ausnahme, auch nicht in Awake. Mit Namen wird nichts gesucht, auch nicht mit Text, das ist einfach Käse, das braucht es nicht. FindObjectOfType dagegen benutze ich in Ausnahmefällen ganz gerne, vor allem, wenn es nur so ein Init-Ding ist und man einen Workaround braucht, und man ist sich nicht so hundertprozentig sicher, wie man die Sachen verknüpfen würde. Im Plural FindObjectsOfType akzeptiere ich wiederrum wenig, weil ich der Meinung bin, dass man meistens über eine Objektstruktur in der Szene ein viel besseres Ergebnis erzeugt.

Interviewer

Gibt es für dich dann irgendwann den Moment oder auch eine Projektgröße, wo du dir ein externes Dependency Injection Framework vorstellen könntest, oder geht dir das zu sehr gegen dein Prinzip, bei welchen du die Verknüpfungen im Unity-Inspektor sehen möchtest?

Person B

Ich habe verschiedene Dependence Injections schon ausprobiert und ich war immer unzufrieden damit. Also ich habe da eine lange Liste an Gründen, was mich daran stört. Deswegen sehe ich keinen Grund es zu verwenden. Also ich sehe nicht den Vorteil, den mir ein klar definiertes Interface bietet. Vor allem, weil ein klar definiertes Interface mich dazu zwingt, guten Code zu schreiben. Was ein Dependency Injection Framework nicht unbedingt tut. Also, ich sehe die Vorteile nicht, die ich im Vergleich habe, wenn die Kontrolle in meiner Hand liegt. Ich habe immer das Gefühl Dependency Injection sind riesige Frameworks, die so eine Massive Menge an Code mitbringen müssen und komplett vorschreiben, wie die Arbeit in dem Projekt zu laufen hat, weil du dann nichts mehr machen kannst ohne dieses Framework.

Interviewer

Gibt es besondere Herausforderungen oder Schwierigkeiten bei der Verwendung solcher Frameworks?

Person B

Die Schwierigkeiten, die ich halt sehe, ist die Serialisierung der Daten von den Dependency Injection Frameworks. Diese ist oft mega langsam, weil sie super viele Informationen brauchen, die du nicht einfach mit der normalen Serialisierung abspeisen kannst. Das heißt dann ist meistens irgendwie nochmal JSON oder XML oder irgendwie ein anderes Format dazwischen. Meistens laufen die Daten von der Dependency Injection wahnsinnig volatil und zwischen den Unity-Versionen und zwischen den einzelnen Versionen, crashen die mal ganz gerne und dann gehen die Zuweisungen kaputt und die Typen sind nicht mehr klar auffindbar, vor allem wenn sich irgendwie der Namespace ändert oder die Assembly sich ändert. Ich fand es auch meistens doof, dass das Debugging und der Stacktrace innerhalb von Unity nicht so wahnsinnig gut mit den Dependency Injections funktioniert, weil der Stacktrace sich sehr stark dadurch verschleiert, dass man erstmal durch dieses komplette Framework durch muss, bevor man überhaupt an den eigentlichen Code kommt. Die meisten Dependency Injection Frameworks laufen nicht ohne Reflections in irgendeiner Form, was zum Beispiel auf der Switch super langsam ist. Selbst wenn es kompiliert und man hat irgendwie eine C++ nahe Version, dann ist aber trotzdem wahnsinnig viel Overhead da oben drauf, um diese ganzen Reflection libraries dann ordentlich abzubilden, nur um irgendwelche Methoden aufzurufen. Die Kompilierungszeit erhöht sich stark und die Abhängigkeiten graben sich halt so tief ins Spiel ein, dass es halt eigentlich alles nur noch über diese Injections läuft.

Interviewer

Dann kommen wir jetzt zur nächsten Kategorie, den Design Patterns. Da würde mich interessieren, wann du dich konkret entscheidest, ein Design Pattern einzusetzen. Wenn du ein Problem identifiziert hast und du ein passendes Pattern kennst, oder braucht es da noch ein bisschen mehr bis so ein Design Pattern auch wirklich bei dir im Code landet?

Person B

Patterns sind ja dazu da, um einen das Leben prinzipiell in allem einfacher zu machen. Bei mir kommen meistens schon während ich definiere, was die Aufgabe von einem Modul sein soll, mehrere Ideen, welche Design Patterns man verwenden könnte. Und ich sag bewusst plural, weil eigentlich ist es ja nie ein Pattern. Ich glaube halt, dass ist auch häufig der Fehler, der gemacht wird, dass man halt irgendwie denkt, jetzt muss man da genau ein Pattern anwenden und die Struktur, alles an Code muss sich dann diesem kompletten Pattern unterwerfen. Und dann stellt man irgendwie fest, das passt gar nicht zusammen, oder die Struktur lässt sich da gar nicht drauf anwenden. Häufig auch, weil man versucht diese Patterns zu kleinteilig auf zu kleinen Ebenen anzuwenden. Ich persönlich sehe es so, eigentlich sollte man immer mit einem Pattern arbeiten. Zumindest auf der Ebene der Kommunikation und der Struktur. Eine einzelne Klasse bindet sich ja darüber an. Also eigentlich stehen die gar nicht so für sich allein. Ich baue auch manchmal erstmal Code und stelle dann fest, eigentlich hättest du ein Command-Pattern hier einführen müssen. Oder eigentlich hättest du schon direkt dich damit auseinandersetzen müssen, dass es hier sehr zeitkritisch zur Sache geht, also sorg dafür, dass diese zeitkritischen Sachen abgefangen werden. Und dann hier sind deine 3 Möglichkeiten, wie du damit umgehen kannst. Was aber dann häufig eher daher rührt, dass sich ja bei Spielen permanent die Anforderungen ändern. Dann ist es eher eine Herausforderung ein Design Pattern zu wählen, bei dem man weiß, man hat noch ein bisschen Luft nach oben, als eins zu wählen, wo man weiß, entweder das oder das andere. Wobei meistens kann man die Sachen halt auch kombinieren. Ich sehe die auch nicht so wahnsinnig dogmatisch, ich sehe die eher als Rezepte. Worauf kommt es an, wofür wird es angewandt? Und dann versuche ich das quasi als Guideline schon von vornherein mit reinzubringen. Weil vor allem in Teams, also vor allem, wenn man ein Paar mehr Leute ist, hilft es wahnsinnig stark und wahnsinnig gut, wenn alle so ein bisschen wissen, was Sache ist und alle so die Leitplanken sehen. Und ein Design Pattern führt schnell dazu, dass man eine gemeinsame Sprache spricht und führt schnell dazu, dass die anderen auch wissen, in welche Richtung weiterentwickelt werden muss. Wenn ich kein Patterns habe, dann fange ich vielleicht an irgendwas reinzubauen. Und der nächste kommt. Macht dasselbe. Da finde ich, hilft das immer. Also das Team auf eine gemeinsame Sprache, auf eine gemeinsame Ebene führen, dass man weiß, wie man den Code erweitert, ohne dass man noch großartig sich austauschen muss und rausfinden muss, wie man das jetzt macht.

Interviewer

Also für dich ist es dann meistens auch eher der Aspekt, wie du gerade sagtest, dass eine Kommunikationslinie geschaffen wird, wo alle erkennen, was da gerade passiert und man erkennt dann diese Patterns halt auch wieder. Also weniger, ich habe jetzt ein Pattern angewendet, ich habe dadurch ein bisschen Code gespart.

Person B

Man muss halt klar beachten, dass man ja nicht nur Zeit in die Entwicklung des Codes steckt, sondern auch Zeit in die Kommunikation des Codes und Zeit, die man miteinander kommuniziert. Deswegen bin ich der vollen Überzeugung, dass alles, was dazu beiträgt, dass man klar ersichtlich und einfach verstehen kann, was Code macht, ist das Nonplusultra. Ja, es ist ein „Mehraufwand“, aber das geht ja ins Fleisch und Blut über, also dass man relativ schnell dann auch so seine Lieblingspattern hat und so ein bisschen seine eigene Sprache hat, sein eigenes Vorgehen hat. Das Team lernt das so gemeinsam kennen. Und natürlich ist dann nicht alles der Königsweg und alles irgendwie super sauber. Das wünscht man sich natürlich, aber das ist auch utopisch, dass dort in der Produktion, die irgendwie wirtschaftlich arbeiten soll, funktionieren kann. Aber es hilft so unfassbar dabei, große Strukturen zu abstrahieren, vor allem, wenn man halt weiß, ich kann diese Patterns Schachteln und dann quasi von klein ins Große oder vom Großen ins Kleine immer wieder auch gedanklich die Ebene wechseln. Es hilft ja total, all den Code außenrum zu vergessen. Ich muss mich nicht damit auseinandersetzen, wie diese anderen Systeme funktionieren und wann diese funktionieren, weil ich mich dann sehr konzentriert eben auf diese eine Anwendung oder diese eine Klasse oder diese eine Funktion fokussieren kann.

Interviewer

Das war jetzt sehr viel pro Design Patterns und sehr viel Positives. Sind die dann manchmal vielleicht auch hinderlich in der Entwicklung? Hast du das schon irgendwie Erfahrung machen können?

Person B

Ja, aber da muss ich ganz klar sagen, fehlende Erfahrung. Also das war dann eher so ein Moment, wo ich sage, ich hatte einfach die Erfahrung nicht. Jetzt habe ich sie gemacht und das habe ich jetzt abgespeichert, dass ich nicht noch mal versuche, so etwas zu verwenden. Und ich sehe es halt häufig auch bei jüngeren Programmierern oder bei Anfängern, dass man irgendwie alles immer in ein Pattern packen möchte. Was auch eine gute Herangehensweise ist, um es zu lernen, aber man fällt halt dann auch häufiger mal hin. Das gehört einfach dazu. Und das passiert dann einem aber später im Beruf nicht mehr oder nicht mehr so häufig, weil man halt schon viele Patterns ausprobiert hat. Und dann sieht man ein Problem und denkt sich, ich habe dieses Problem schon zehnmal gelöst, mindestens. Also es gibt so viele Probleme, die ich auf so viele unterschiedliche Arten schon gelöst habe. Da reizt mich dann auch nicht mehr, irgendwie was Neues auszuprobieren, sondern gehe ich mit dem, wo ich so das beste Gefühl hatte, wo ich weiß, in 5 von 8 Fällen bin ich damit gut gelaufen. Und dann gehe ich eher damit, als dann noch mal ein Experiment zu machen. Wohingegen, wenn ich etwas zum ersten Mal löse, es ja immer ein Experiment ist. Und dann, macht es zeitlich, wirtschaftlich, geistig keinen Unterschied, ob ich direkt versuche, da irgendwie eine Struktur reinzubringen, weil überarbeiten und refaktorisieren musst man es eh.

Interviewer

Du hattest eben schon angesprochen, dass Kommunikation gerade im Team sehr wichtig ist. Da würden wir dann auch direkt zu der nächsten Kategorie kommen, nämlich Prinzipien. Da geht's genau um das. Also die Wartbarkeit und auch die Verständlichkeit der Codebasis. Hast du da irgendwelche Prinzipien, auf die du schwörst?

Person B

Ehrlich gesagt, ich bin bei diesen Prinzipien immer nicht so überzeugt. Das sind häufig Buzzwords. Gefühlt aus meiner persönlichen Meinung, möchte halt irgendein Programmierberater dir jetzt irgendwie verkaufen, was jetzt so das neueste krasse Prinzip ist. Und viele davon haben gute Ideen. Also ich schau mir die dann an und ich denke so, ja das macht schon irgendwie Sinn und ich verstehe, was ihr mir sagen möchtet, und mein Bauchgefühl sagt mir in den meisten Situationen auch, dass ich ähnlich vorgehen sollte. Aber ich denke mir

jetzt nicht die ganze Zeit, ich muss aufpassen, dass ich nicht gegen dieses Prinzip verstoße oder gegen dieses Prinzip verstoße. So bin ich persönlich nie rangegangen. Also ich bin dann schon eher über ein Design Pattern rangegangen, als zu sagen, dass muss jetzt dieses krasse Prinzip sein. Und jetzt kommt die Ausnahme. Also ich finde KISS ist immer noch ganz oben für alles und sollte man auch auf seine Prinzipien anwenden. Also umso einfacher man sich es macht und umso klarer man sich dessen ist, dass egal für wie toll man sich hält, am Ende sind wir halt alle einfach nur sehr einfache Menschen, die sich sehr wenige Dinge auf einmal merken können, die sich sehr begrenzt auf Dinge konzentrieren können. Und Sachen simpel zu halten oder simpel zu machen, ist verdammt schwer. Aber ist genau das, was einen dann vorwärtsbringt. Also aus meiner Sicht muss Code sich lesen wie ein gutes Buch. Wenn man Kommentare braucht, wenn man irgendwie das Gefühl hat, man muss dort super viel erklären oder man benutzt den neuesten syntaktischen Sugar, da würde ich mich immer fragen, habe ich das selber schon so verinnerlicht, dass ich das einfach so lesen kann? Hat mein komplettes Team das so verinnerlicht, dass man das einfach so lesen kann? Oder mache ich es gerade eigentlich allen nur schwerer zu verstehen, was hier passiert? Man lernt natürlich noch dazu, also die Sprache entwickelt sich ja weiter. Ich will nicht sagen, dass man darauf komplett verzichten sollte. Ist vielleicht auch ein schlechtes Beispiel. Man sollte wirklich versuchen, es simpel zu halten, für sich selbst und für die anderen. Wenn man es nicht für die anderen machen kann oder will, sollte man sich immer bewusst sein, wie stehen Dinge später? Man geht mal irgendwie eine Woche in Urlaub, man macht mal 2 Wochen was anderes und dann freut man sich einfach, wenn man in der Vergangenheit Code produziert hat, den man auf Anhieb verstehen kann, dann ist alles richtig.

Interviewer

Das kann ich 100% unterschreiben. Ich vergesse gerne mal nach einem Monat schon, wenn ich da wieder reinschaue, wo bin ich hier gelandet, wer hat das produziert? Dann abschließend vielleicht noch die Frage, was macht für dich jetzt ein skalierbares Unity-Projekt aus, was kommt da so alles zusammen?

Person B

Also prinzipiell muss ich sagen, die Frage ist viel, viel zu offen gestellt, als dass ich das beantworten könnte. Weil die Frage ist, was möchtest du skalieren? Das Einzige, was ich explizit verallgemeinern könnte, ist, es braucht einfach wahnsinnig viel Disziplin und wahnsinnig viel Best Practice. Das ist das, was am Ende dafür sorgt, dass eine Produktion in allem, in dem es existiert, irgendwie skalierbar wird. Weil nichts davon kommt natürlich. Sei es, dass man eben viele Konventionen einführt oder sich bewusst ist, was konfigurierbar sein muss. Zu jedem bisschen was hilft, musst du ja sagen, weil Games sich wahnsinnig schlecht skalieren. Also es ist verdammt schwer Spiele zu skalieren. Und am Ende braucht man jedes bisschen, das hilft. Und das Effizienteste ist immer Disziplin. Disziplin im Sinne von, es fängt bei der Benennung an. Und nichts davon kriegt man geschenkt. Etwas skalierbar machen und skalierbar halten kostet viele Ressourcen. Dar Scope ist am Ende immer entscheidend. Also wie groß wurde das Projekt überhaupt mal angedacht? Es ist glaube ich wahnsinnig wichtig, dass der Scrope generell schon mal passt, weil diese Vorstellung, dass digital ja automatisch ins unendlich skalieren kann, stimmt ja auch nicht. Eine meiner Learnings ist auch, dass nach oben skalieren genauso schwer ist wie nach unten skalieren.

Interviewer

Du hast recht, die ist sehr offen gestellt die Frage. Aber ich finde es gut, dass du trotzdem für diese offene Frage, wo du sagst, es spricht halt sehr viele Aspekte an, einen Begriff nennen konntest, der irgendwie auf alles passt, nämlich die Disziplin. Ansonsten sage ich schon mal vielen Dank, dass du dir die Zeit für das Interview genommen hast, und dann beende ich jetzt die Aufnahme.

Interviewleitfaden

Vorbemerkung

Im Rahmen meiner Masterarbeit untersuche ich das Thema der Skalierbarkeit von (Unity-) Projekten. Der Fokus liegt dabei auf der Codebasis. Dabei geht es um Themen wie Assembly Definitionen, Dependency Injection Frameworks und Design Patterns. Dazu würde ich dir im Folgenden gerne ein paar Fragen stellen. Zu Analysezwecken würde ich das Interview gerne aufzeichnen. Die Aufzeichnung wird selbstverständlich vertraulich behandelt, ausschließlich für meine Masterarbeit verwendet und nicht an Dritte weitergegeben.

Hast du noch Fragen, bevor wir beginnen?
Dann werde ich die Aufnahme jetzt starten.

Interview

Assembly Definitionen

- Inwiefern finden Assembly Definitionen Anwendung in deinen Projekten?
    - **Gar nicht:** Gibt es in den Projekten Methoden zur Verringerung der Kompilierungszeiten und der Modularisierung des Codes?
    - **Ansonsten:** Welchen Mehrwert bieten Assembly Definitionen in den Projekten?
        - Kannst du beschreiben, welchen Umfang typischerweise eine Assembly Definition in deinen Projekten hat?

Dependency Injection Framework

- Auf welche Art(en) werden die Abhängigkeiten in deinen Projekten bereitgestellt?
    - **Mögliche Vorschläge:** "*Unity-Inspektor*", "*GameObject.Find*", …
    - Warum hast du dich genau für diese Methode(n) entschieden?
- Ab welcher Größe eines Unity-Projekts, würdest du ein externes Dependency Injection Framework in Erwägung ziehen?
    - Kannst du deine Antwort begründen?
- Welche Schwierigkeiten können bei dem Einsatz eines externen Dependency Injection Frameworks auftreten?

Design Patterns

- Zu welchem Zeitpunkt entscheidest du dich, ein Design Pattern einzusetzen?
    - **Mögliche Vorschläge:** "*Sobald sich der Aufwand rechtfertigen lässt*", "*sobald ein Design Pattern Anwendung finden kann, auch wenn es zukünftig vielleicht keinen weiteren Mehrwert bietet*", …
- Inwiefern können Design Patterns vielleicht hinderlich in der Entwicklung sein?

Prinzipien

- Welche Prinzipien in der Softwareentwicklung helfen dir dabei, die Codebasis verständlicher und wartbarer zu machen?
    - **Mögliche Vorschläge:** *SOLID, KISS, DRY*, …
- Was macht für dich ein skalierbares Unity-Projekt aus?
    - **Mögliche Vorschläge:** *Dokumentation, Ordnerstruktur, Codebasis, Scene-Management, …*

Schlussteil

Vielen Dank, dass du dir Zeit für dieses Interview genommen hast. Gibt es noch etwas, das du ergänzen möchtest?

Einverständniserklärung zum Interview

Masterarbeit: Skalierbarkeit eines (Unity-)Projekts: Einführung eines Dependency Injection Frameworks, Assembly Definitions und Design Patterns
Interviewer_in: Vincent Krenzke
Interviewdatum: 21.09.2023

Ich erkläre mich dazu bereit, im Rahmen des genannten Themas zur Masterarbeit an dem Interview teilzunehmen. Ich wurde über das Ziel des Interviews informiert. Ich kann das Interview jederzeit abbrechen, ohne dass mir dadurch irgendwelche Nachteile entstehen.

Ich bin damit einverstanden, dass das Interview mit einem Aufnahmegerät aufgezeichnet und dann von dem/der Interviewer_in in Schriftform gebracht wird. Die Audiodateien werden zum Projektende am 22.10.2023 gelöscht. Die Transkripte der Interviews werden anonymisiert, d.h. ohne Namen und Personenangaben gespeichert. Mir wird außerdem versichert, dass das Interview in wissenschaftlichen Veröffentlichungen nur in Ausschnitten zitiert wird, um sicherzustellen, dass ich auch durch die in den Interviews erzählte Reihenfolge von Ereignissen nicht für Dritte erkennbar werde.

Unter diesen Bedingungen erkläre ich mich bereit, das Interview zu geben, und bin damit einverstanden, dass es aufgezeichnet, verschriftlicht, anonymisiert und ausgewertet wird.

_____	_____
Ort, Datum, Unterschrift Interviewte_r	Ort, Datum, Unterschrift Interviewer_in

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Master-Thesis mit dem Titel:

Skalierbarkeit eines (Unity-)Projekts: Einführung eines Dependency Injection Frameworks, Assembly Definitions und Design Patterns

. . . . . .

selbständig und nur mit den angegebenen Hilfsmitteln verfasst habe.
Alle Passagen, die ich wörtlich aus der Literatur oder aus anderen Quellen
wie z. B. Internetseiten übernommen habe, habe ich deutlich als Zitat
mit Angabe der Quelle kenntlich gemacht.

. . . . . .

Datum, Unterschrift