

# State of the art Software Engineering

## PROJEKTARBEIT

für die Prüfung zum

Bachelor of Science

des Studienganges Medizinische Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

**Florian Koch & Jan Metzger**

Abgabedatum 30.04.2022

Bearbeitungszeitraum

Semester 5 & 6

Matrikelnummer

1697008 & 3029924

Kurs

Tinf19B1

Ausbildungsfirma

Agilent Technologies Deutschland GmbH  
Waldbronn

## Erklärung

Wir versichern hiermit, dass wir die Projektarbeit mit dem Thema: “State of the art Software Engineering” selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

---

Ort    Datum

---

Unterschrift

# Contents

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Aufgaben definition . . . . .	1
<b>2</b>	<b>Umsetzung</b>	<b>3</b>
2.1	Entwurfsmuster . . . . .	3
2.1.1	Klassendiagramm . . . . .	3
2.2	Programming Principles . . . . .	4
2.2.1	SOLID . . . . .	4
2.2.2	GRASP . . . . .	7
2.2.3	DRY . . . . .	9
2.3	Clean Architecture . . . . .	9
2.3.1	Abstraction Code . . . . .	9
2.3.2	Domain Code . . . . .	9
2.3.3	Application Code . . . . .	10
2.3.4	Adapter . . . . .	10
2.3.5	Service . . . . .	10
2.3.6	Plugin . . . . .	11
2.3.7	UI Code . . . . .	11
2.4	Domain Driven Design . . . . .	12
2.4.1	Ubiquitous Language . . . . .	12
2.4.2	Repositories . . . . .	12
2.4.3	Aggregates . . . . .	13
2.4.4	Entities . . . . .	13
2.4.5	Value Objects . . . . .	13
2.5	Unit Tests . . . . .	13
2.5.1	ATRIP-Regeln . . . . .	14
2.5.2	Code Coverage . . . . .	15

2.5.3	Einsatz von Mocks . . . . .	16
2.5.4	AAA-Normalform . . . . .	18
2.5.5	Welches Framework und warum . . . . .	18
2.5.6	Mutation Tests . . . . .	18
2.6	Refactoring . . . . .	19
2.6.1	Beispiele für Refactoring . . . . .	19
2.6.2	Code Smells . . . . .	20
<b>3</b>	<b>Extras</b>	<b>22</b>
3.1	Build pipeline . . . . .	22

# List of Figures

1.1	Anforderungen für die Projektarbeit . . . . .	2
2.1	Klassendiagramm des Application Codes . . . . .	4
2.2	Schichtenarchitektur der Barbell Tracker Anwendung . . . . .	10
2.3	Screenshot der Code Coverage unserer Anwendung (Barbell Tracker) . .	16
2.4	Screenshot der Unit Test Summary unserer Anwendung (Barbell Tracker)	17
2.5	Mutationtests der Service Schicht . . . . .	19

# Abkürzungsverzeichnis

<b>UI</b> User Interface . . . . .	8
<b>MVVM</b> Model View ViewModel . . . . .	6
<b>WPF</b> Windows Presentation Foundation . . . . .	6
<b>SVG</b> Scalable Vector Graphics . . . . .	3
<b>SOLID</b> Single Responsible -, Open/Close -, Liskov Substitution -, Interface Segregation - and Dependency Inversion Principle . . . . .	4
<b>SRP</b> Single Responsibility Principle . . . . .	5
<b>OCP</b> Open/Close Principle . . . . .	5
<b>LSP</b> Liskov Substitution Principle . . . . .	6
<b>ISP</b> Interface Segregation Principle . . . . .	6
<b>DIP</b> Dependency Inversion Principle . . . . .	7

<i>LIST OF FIGURES</i>	vi
<b>KISS</b> Keep it simple, stupid . . . . .	5
<b>GRASP</b> General Responsibility Assignment Software Patterns/Principles . . . .	7
<b>DRY</b> Don't Repeat Yourself . . . . .	9
<b>UI</b> User Interface . . . . .	8
<b>XAML</b> Extensible Application Markup Language . . . . .	8
<b>UWP</b> Universal Windows Platform . . . . .	11
<b>ATRIP</b> Automatic, Thorough, Repeatable, Independent and Professional . . . .	14
<b>FIFO</b> First in first out . . . . .	3
<b>AAA</b> Arrange, Act und Assert . . . . .	18

# Chapter 1

## Einleitung

### 1.1 Aufgaben definition

Diese Hausarbeit wird unsere Fähigkeiten in der professionellen Softwareentwicklung an Hand einem selbst gewähltem Praxisprojekt unter Beweis stellen. Dazu wurde die Anwendung Barbeltracker (Barbel englischen Wort für Langhantel) entwickelt. Themen die in dieser Projektarbeit behandelt wurden sind Entwurfsmuster, Clean Architecture, Domain Driven Design, Programming Principles, Unit Tests und zuletzt das Refactoring. In dieser Hausarbeit wird zuerst die Theorie kurz wiederholt. Anschließend wird auf unseren Codesubstrat verwiesen und näher da gelegt, warum dieses Thema auf diese Art und Weise umgesetzt oder nicht umgesetzt wurde. In der Abbildung 1.1 finden Sie mehr Details zu den Aufgabenfeldern.

Den Codesubstrat der diese Projektarbeit begleitet, finden Sie auf unserem Github Repo (<https://github.com/Hawk1401/BarbellTracker>).



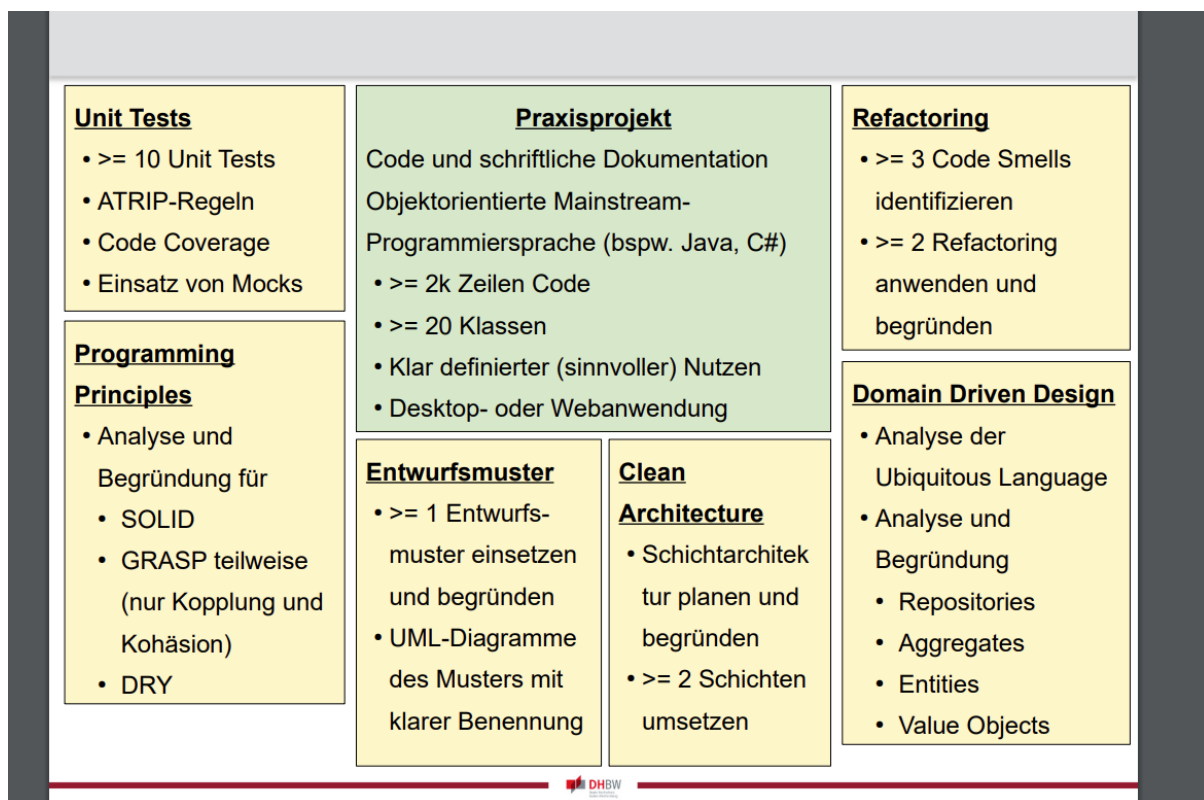


Figure 1.1: Anforderungen für die Projektarbeit

# Chapter 2

## Umsetzung

### 2.1 Entwurfsmuster

Als Beispiel für die Entwurfsmuster nutzen wir unseren Eventbus. Im Wesentlichen stellt der Eventbus eine zentrale Ereignis-Quelle zur Verfügung, an denen sich Observer für Ereignisse anmelden können und bei entsprechendem Auftreten informiert werden. Gleichzeitig dient es unseren Komponenten als zentrale Stelle, über die Ereignisse publik gemacht werden. Typischerweise existiert in einer Applikation nur genau ein Eventbus, der von allen Komponenten verwendet wird. Bei uns ist es die Klasse *EventSystem* im *ApplikationCode*. Mit den Topics im Unterordner *Event*.

Ein weiterer wichtiger Vorteil ist die dadurch entstehende Entkopplung der Komponenten. Da der Eventbus nun als zentrale Einheit in die Mitte rückt, muss weder der Observer die tatsächliche Quelle des Ereignisses kennen, noch muss die Ereignisquelle wissen, wer tatsächlich auf das Ereignis reagiert.

#### 2.1.1 Klassendiagramm

Unter folgendem Link (<https://hawk1401.github.io/BarbellTracker/>) finden Sie das gesamte Klassendiagramm als Scalable Vector Graphics (SVG) Datei. Die Grafik 2.1 zeigt das Klassendiagramm des *Application Codes*, welches unseren *EventBus* beinhaltet. Zu dem Eventbus gehören die Klassen *EventSystem*, *EventQueue*, *EventQueueItem*, die Event-Models sowie unsere Topics der Events in den Packages *EventModel* und *Event*. Das *EventSystem* benötigt die *EventQueue*, um die geschossenen *Events* ordnungsgemäß abzuarbeiten. Dazu werden die *Events* in eine First in first out (FIFO) Warteschlange eingefügt.

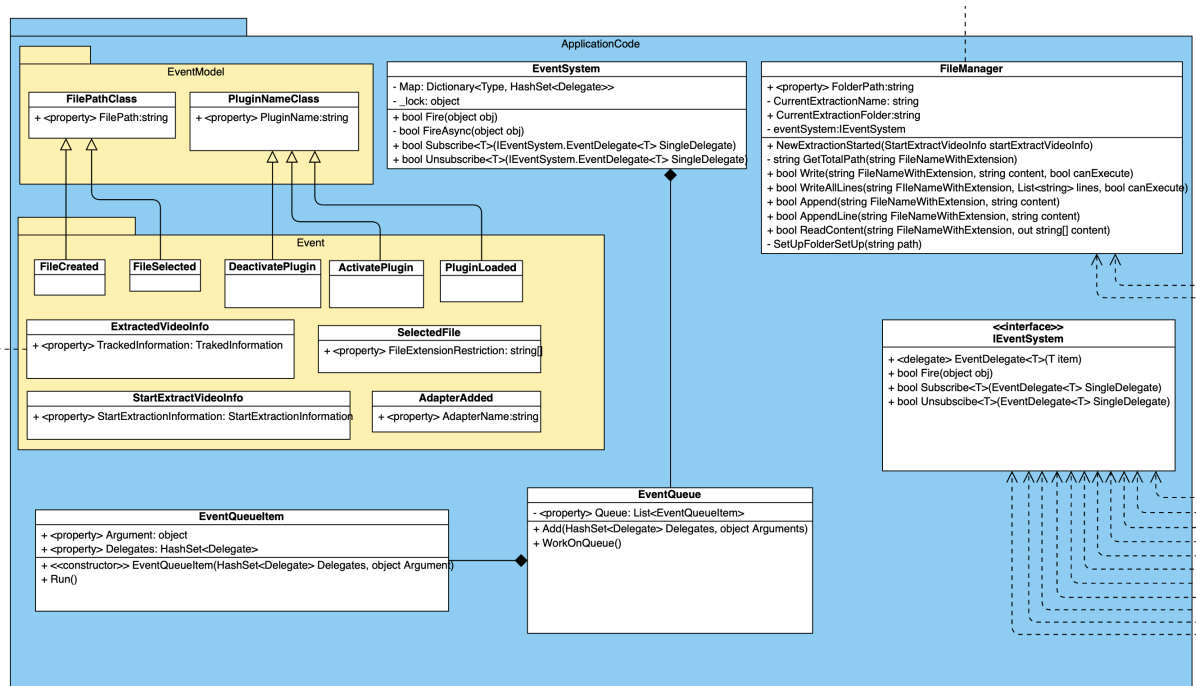


Figure 2.1: Klassendiagramm des Application Codes

## 2.2 Programming Principles

Programmierprinzipien dienen als Grundlage für Entscheidungen. Es sind allgemein anerkannte Regeln für Begründungen und Argumentationen. Diese ist in der Ursprungsform eine kontextlose Idealvorstellung doch bei der Anwendung spielt der Kontext meist eine wichtige Rolle. Die Programmierprinzipien könne auch als Leitlinie für zielgerichtetes Handeln definiert werden. Im folgenden Abschnitt werden drei Programmierprinzipien näher erklärt und aufgezeigt, wie wir diese in unserem Programmierbeispiel umgesetzt haben.

### 2.2.1 SOLID

Das Single Responsible -, Open/Close -, Liskov Substitution -, Interface Segregation - and Dependency Inversion Principle (SOLID) verfolgt das Ziel wartbare Software, erweiterbare Systeme und langlebige Codebasis zu entwickeln.

### Single Responsibility Principle

In Single Responsibility Principle (SRP) geht es um das Prinzip der eindeutigen Verantwortlichkeit. Eine Klasse sollte immer nur eine einzige Verantwortlichkeit haben. Dabei muss eine klar definierte Aufgabe für jedes Objekt existieren. Durch Zusammenspiel der Objekte wird ein übergeordnetes Verhalten erreicht. Das führt zu niedriger Kopplung und Komplexität. Diese Eigenschaft führt zu Separation of Concerns. Die Separation of Concerns ist aber nicht nur auf Klassen anzuwenden sondern auch auf Methoden/Funktionen und Variablen.

Als Negativ-Beispiel für das SRP in unserem Codesubstrat wird die Klasse *FileManager* genutzt, da diese Klasse viele Funktionen auf einmal erfüllt. Das bedeutet, die Klasse beinhaltet viele verschiedene Methoden. Sodass die Klasse Dateien Lesen, Schreiben, Verändern, Erweitern und neue Verzeichnisse anlegen kann. Es wäre besser, wenn wir für viele der verschiedenen Funktionen eigene Klassen implementieren würden. Diese würde die eindeutige Verantwortlichkeit verbessern aber auch die Übersichtlichkeit der Funktionen und es Codes erhöhen.

Als Positiv-Beispiel für das SRP in unserem Codesubstrat wird die Klasse *EventQueueItem* genutzt, da diese Klasse nur eine einzige Aufgabe hat. Die Klasse *EventQueueItem* beinhaltet das Aufrufen von mehreren Delegates mit ein und dem selben Argument. Diese Funktionalität wurde aus der Klasse *EventQueue* ausgelagert um eine niedrigere Komplexität zu schaffen.

### Open/Closed Principle

Das Open/Close Principle (OCP) beschäftigt sich mit der Erweiterbarkeit von bestehender Software. Dabei sind Software Entitäten wie Module, Klassen und Methoden offen für Erweiterungen aber geschlossen bezüglich Veränderung. Ein Beispiel für Erweiterung ist die Vererbung. Dabei verändert nur die Unterklasse ihr Verhalten. Wichtig ist zu vermeiden, dass bestehender Code verändert wird. Jedoch sollte vermieden werden spekulativ Komplexität mit einzubauen. Denn es gilt ebenfalls Keep it simple, stupid (KISS).

Für die Umsetzung von OCP wurde möglichst an vielen Stellen Interfaces genutzt. Interfaces ermöglichen uns eine klar definierte Struktur aufzubauen bei denen sich alle Objekte darauf verlassen können, dass diese Struktur auch so umgesetzt wurde. Die Implementierte Version der Interfaces können dennoch problemlos erweitert werden ohne die Grundstruktur zu verändern. Fünf Beispiele in unserem Codesubstrat für OCP Interfaces sind:

1. BarbellTracker.Adapter.Interface.IPlugin
2. BarbellTracker.Adapter.Interface.IProcessingPlugin
3. BarbellTracker.Adapter.Interface.ITrackerPlugin
4. BarbellTracker.Adapter.Interface.IUIAdapter
5. BarbellTracker.Services.Interface.ICalculator<T>

### Liskov Substitution Principle

Im Liskov Substitution Principle (LSP) sollten Objekte in einem Programm durch Instanzen ihrer Subtypen ersetzbar sein, ohne die Korrektheit des Programms zu ändern. Daraus resultieren strikte Regeln für Vererbungshierarchie. Subtypen sollen sich so verhalten wie ihr Basistyp. Daraus resultiert, dass ein Subtyp die Funktionalität lediglich erweitern, nicht aber einschränken darf.

Im Codesubstrat wird das LSP im *WPF\_DesktopClient* angewendet. Hier haben wir das Windows Presentation Foundation (WPF) mit dem Pattern Model View View-Model (MVVM) umgesetzt, dafür werden *ViewModels* entwickelt, die die Informationen und Funktionalitäten für die *View* bereitstellen. Da es Funktionalitäten gibt, die alle *ViewModels* benötigen, erben die *ViewModels* von der *ViewModelBase*. Die *ViewModelBase* erbt wiederum von *PropertyChangedNotifier*. Damit die *View* Änderungen im *ViewModel* registriert, wird der *PropertyChangedNotifier* benötigt. Das gleiche gilt für entgegengesetzte Richtung von *View* zu *ViewModel*. Durch das Vererben der Grundfunktionen für die *ViewModels* ersparen wir uns den *PropertyChangedNotifier* in jeder *ViewModel*-Klasse neu zu implementieren und reduzieren dadurch Fehler bei mehrmaligem definieren.

### Interface Segregation Principle

Im Interface Segregation Principle (ISP) soll ein Klient nicht von Details eines Service abhängig sein, die er gar nicht benötigt. Das bedeutet, es ist besser mehrere spezifische Interfaces als ein Allrounder-Interface zu implementieren, was zu hoher Kohäsion führt. Klassen oder Typen mit hoher Kohäsion repräsentieren eine gut definierte Einheit.

Wir haben bei den drei Interfaces *IPlugin*, *IProcessingPlugin* und *ITrackerPlugin* ISP angewandt. Hierbei wurde die benötigte Funktionalität auf drei Plugins aufgeteilt. Das *IPlugin* verfügt hierbei über die Funktion, die jedes Plugin benötigt. Das ist der Namen so wie eine Beschreibung. Die *IProcessingPlugin* Schnittstelle stellt hierbei die Funktionen

bereit, die benötigt werden Daten zu verarbeiten. Das *ITrackerPlugin* Interface stellt die Methoden bereit, welche benötigt werden um Daten aus einer beliebigen Datei aus zu lesen und über das Eventsystem zu schicken.

### Dependency Inversion Principle

Im Dependency Inversion Principle (DIP) geht es um die Entkopplung. Das heißt, Abstraktionen sollten nicht von Details abhängen, sondern Details von Abstraktionen. Auch Klassen hoher Ebenen sollten nicht von Klassen niedriger Ebenen abhängen. Hier sollten ebenfalls beide von Interfaces abhängen. Die Abhängigkeit auf konkrete Klassen ist eine starke Kopplung, die vermieden werden sollte. Zur Auflösung starker Kopplung kann Dependency Injection genutzt werden. Durch die Umsetzung der DIP kann eine bessere Wiederverwendbarkeit des Codes erlangt werden. Ebenfalls werden dadurch klare Schnittstellen definiert und eine flexiblerer Zusammenarbeit ermöglicht.

Wir haben uns dazu entschlossen DIP an einigen Stellen in unseren Code anzuwenden, zwei Beispiele sind hierfür das *IService* sowie das *IEventSystem* Interface. Hiermit lösen wir die Kopplung so gut wie es geht auf. Beim Eventsystem gibt es zum aktuellen Zeitpunkt auch nur eine Implementierung, deshalb könnte theoretisch das Interface auch weggelassen werden, ohne dass es kurzfristige Folgen hätte. Wir machen es jedoch nicht, da es für zukünftige Erweiterungen vom Vorteil ist mehrere Implementierungen zu haben.

### 2.2.2 GRASP

General Responsibility Assignment Software Patterns/Principles (GRASP) bietet Standardlösungen für typische Fragestellungen in der Softwareentwicklung. GRASP besteht aus neun Prinzipien:

- Low Coupling
- High Cohesion
- Indirection
- Polymorphism
- Pure Fabrication
- Protected Variations
- Controller

- Information Expert
- Creator

Für den Programmentwurf wird auf das Grundkonzept von Low Coupling und High Cohesion eingegangen.

### Low Coupling

Low Coupling steht für geringe oder lose Kopplung. Dabei steht Kopplung für ein Maß für die Abhängigkeit einer Klasse von ihrer Umgebung/ andere Klassen. Durch die geringe Kopplung erreicht man leichte Anpassbarkeit, gute Testbarkeit, Verständlichkeit durch weniger Kontext und erhöht die Wiederverwendbarkeit.

Schwache Kopplung wurde im Codesubstrat durch das Eventsystem angewendet, aber auch im User Interface (UI) WPF kontext. Wir haben uns für WPF mit dem MVVM Pattern entschieden, da wir durch das Binding zwischen der *View* und dem *ViewModel* eine schwache Kopplung des UIs definierenden Codes (*View* mit Extensible Application Markup Language (XAML)) und dem *ViewModel*, das die Funktionalität der UI mit sich bringt, erreichen. Durch die schwache Kopplung in unserem Code können wir Modifikationen leichter durchführen, ohne das die Veränderungen globale Auswirkungen haben und globale Anpassungen nötig sind.

### High Cohesion

Kohäsion ist ein Maß für den inneren Zusammenhalt einer Klasse. Hier geht es um die Frage wie eng arbeiten die Methoden und Attribute einer Klasse zusammen. Der Ideale Code besitzt eine hohe Kohäsion und eine geringe Kopplung.

Die *ServiceCache<T>* Klasse ist ein Beispiel für hohe Kohäsion. Die Klasse verfügt über zwei öffentliche Methoden, welche ihre gesamte Logik auf insgesamt fünf private Methoden auslagert. Durch die hohe Kohäsion erreichen wir eine Prozessverteilung auf die einzelnen Methoden der Klasse und vereinen dabei zusammenhängende Aufgabenfelder in einer Klasse.

Die Klasse *VelocityToAdapterTable* ist ein Beispiel für geringe Kohäsion, die Klasse hat hierbei vier öffentliche Methoden, die alle ihre Logik nicht weiter auslagern. Somit verfügt die gesamte Klasse über keine privaten Methoden. Bei diesem Beispiel war es nicht möglich die einzelnen Aufgaben zu verbinden, da diese zu unterschiedlich sind, was zu einer geringen Kohäsion führte.

### 2.2.3 DRY

Bei Don't Repeat Yourself (DRY) geht es darum, Funktionen nur einmal zu programmieren. Dadurch wird universell anwendbarer Code, Bauskripte, Testpläne und Dokumentationen entwickelt.

In unserem Codesubstrat wurde DRY in der UI Schicht angewandt. Wir haben hier die *PropertyChangedNotifier* Funktionalität an alle *ViewModels* vererbt. Dadurch muss der *PropertyChangedNotifier* nicht in jedem *ViewModel* erneut implementiert werden. Dies verhindert Schreib- bzw. Implementierungsfehler. Es vereinfacht auch Anpassungen am Code zu vollziehen, ohne diese an mehreren Stellen ebenfalls durch führen zu müssen.

## 2.3 Clean Architecture

Wir haben uns entschieden unsere Applikation in mehrere Schichten zu unterteilen, siehe Abb. 2.2. Technologien wie die UI Komponenten sind ganz außen angesiedelt, da diese stark Zeitabhängig sind und nichts mit der Domain zu tun haben. Wir nutzen für das UI WPF oder Command Line. Eine Schicht weiter innen haben wir den Plugin- und Service-Code. Die Service Schicht wurde eingeführt um Funktionen aus der Plugin Schicht aus zu lagern, da mehrere Plugins den selben Service nutzen. Unsere Schichtenarchitektur besteht aus folgenden Schichten: Abstraction Code, Domain Code, Application Code, Adapter, Service, Plugin und UI Code (WPF Desktop).

### 2.3.1 Abstraction Code

In der *Abstraction Code* Schicht haben wir unsere "Naturgesetze" festgelegt. Für den *Barbell Tracker* ist es die Definition und Mathematik für 2D Vektoren. Da sich die Mathematik hinter den Vektoren nicht ändert, haben wir uns dazu entschieden den 2D Vector in die *Abstraction Code* Schicht zu packen.

### 2.3.2 Domain Code

Hier geht es um langwierige Definitionen und Eigenschaften wie den *TracketInformation* und der *StartExtractionInformation*. Da es hier aber im Gegensatz zum 2D Vector um reine Domain Spezifische Dinge geht, wurden diese in der separaten *DomainCode* Schicht implementiert.



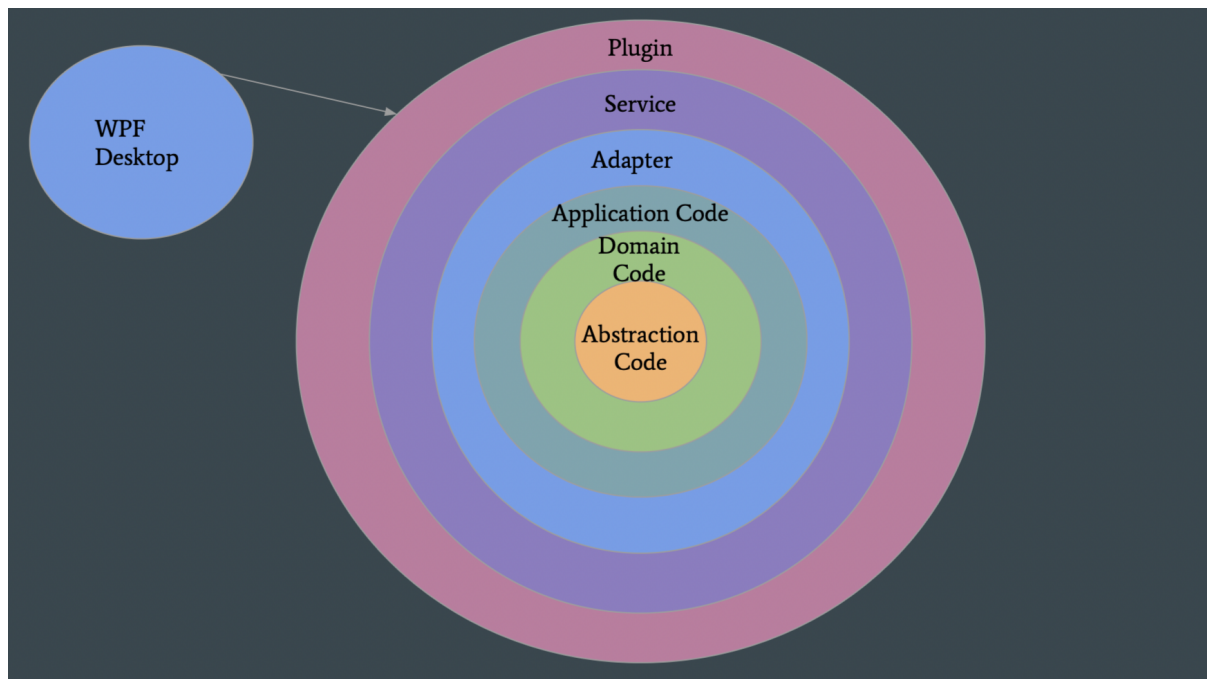


Figure 2.2: Schichtenarchitektur der Barbell Tracker Anwendung

### 2.3.3 Application Code

In der *Application Code* Schicht stellen wir grundlegende Funktionen unserer Anwendung zur Verfügung. Dazu gehört das *EventSystem*, welches die Kommunikation in unser Application bereitstellt. Aber auch der *FileManager* welcher grundlegende Funktionen rund um Dateien und Ordnern bereitstellt.

### 2.3.4 Adapter

Die *Adapter* Schicht stellt die Schnittstellen und Models für die Plugins zur Verfügung. Darüber hinaus beinhaltet es die Manager und die UI Adapter.

### 2.3.5 Service

Die von uns eingeführte Schicht *Service* spaltet die *Plugin* Schicht weiter auf. Alle Services verarbeiten Informationen, dass bedeutet diese haben immer ein Input und ein Output. Die Plugins nutzen diese Services um die nötigen Informationen in der richtigen Form bekommen. Das soll verhindern, dass zwei Plugins die selbe (Teil-) Aufgabe erfüllen. Somit wird die Kommunikation und Abhängigkeiten zwischen Plugins reduziert bzw.

verhindert.

### 2.3.6 Plugin

In der *Plugin* Schicht werden Verarbeitungsfunktionen definiert und der *UI Code* Schicht bereitgestellt.

### 2.3.7 UI Code

Für die UI in unserer Applikation nutzen wir WPF von .NET. WPF bietet das MVVM Programmier-Pattern für Clean Architecture. In WPF wird das UI mit XAML in der *View* definiert, sie beinhaltet keine Logik. Das *ViewModel* dient als Bindeglied für die *View* und dem *Model*. Das bedeutet Sie stellt alle nötigen Daten und Funktionalitäten für die *View* bereit. Im *Model* ist nur Code der nichts mit der UI zu tun hat. In unserem Fall sind das alle Schichten unterhalb der *UI Code* Schicht.

Das MVVM-Muster hilft dabei, die Geschäfts- und Präsentationslogik einer Anwendung von ihrer UI sauber zu trennen. Die Aufrechterhaltung einer sauberen Trennung zwischen Anwendungslogik und Benutzeroberfläche hilft bei der Behebung zahlreicher Entwicklungsprobleme und kann das Testen, Warten und Entwickeln einer Anwendung vereinfachen. Sie kann auch die Möglichkeiten zur erneuten Verwendung von Code erheblich verbessern und ermöglicht Entwicklern und Benutzeroberflächen-Designern eine einfachere Zusammenarbeit bei der Entwicklung ihrer jeweiligen Teile einer Applikation. Das MVVM-Muster wird nicht nur bei WPF, sondern auch bei Xamarin und Universal Windows Platform (UWP) angewendet, was wiederum neu vereinfachte Anwendungsmöglichkeiten gibt, da der Code für verschiedene Plattformen nur einmal geschrieben werden muss. [1]

Der UI Code ist in den Projekten *WPF\_DesktopClient* und *WPF\_HelperClasses* zu finden. *WPF\_HelperClasses* stellt Code zur Verfügung, der für jede WPF Anwendung benötigt wird. Durch das Auslagern in ein eigenes Projekt, kann dieser Code problemlos wieder verwendet werden.

## 2.4 Domain Driven Design

### 2.4.1 Ubiquitous Language

Wir nutzen in unserer Applikation und unserer Domain die gleiche Sprache. Diese wurde zu keinem Zeitpunkt in der Applikation übersetzt. Da wir im Code die Domain Sprache nutzen, verhindern wir Verständnisprobleme.

#### **Velocity**

*Velocity* ist in unser Domain der Begriff, welcher die Geschwinigkeit einer Hantelstange zu einem definierten Zeitpunkt angibt.

#### **Acceleration**

*Acceleration* ist in unser Domain der Begriff, welcher die Beschleunigung einer Hantelstange zu einen definierten Zeitpunkt hat.

#### **Tracked**

Unter dem Wort *Tracked* verstehen wir in unser Domain die analysierte Position einer Hantelstange während einer gesamten Aufnahme.

#### **Processing**

Das Wort *Processing* beschrieb in unser Domain das weitere analysieren der Positionen einer Hantelstange.

### 2.4.2 Repositories

Mit Hilfe von Repositories abstrahiert man den Zugriff und die Suche von Fachobjekten. Ein Repository stellt hierbei die Methoden zur Verfügung mit den man an die gewünschten Objekte herankommt. Meist befindet sich ein solches Repository im *Domain Code*, wir haben uns jedoch entschieden unsere verschiedenen Repositories nicht in den *Domain Code* zu packen, sondern in weiter außen liegenden schichten. Der Grund hierfür ist, dass das Fachobjekt, welches für das Repository zuständig ist erst in der *Adapter Schicht* definiert wird. Unser Beispiel hierfür, ist der *PluginManager*. Der *PluginManager* ist ein Repository für alle Plugins, welche in unser Applikation verwendet werden. Der

*PluginManager* befindet sich in der *Adapter Schicht* und damit auf der selben Ebene wie die Interfaces welche der *PluginManager* verwaltet.

### 2.4.3 Aggregates

Aggregates sind Klassen welche eine Hülle um eine oder mehrere Entitäten/ Value Objects bilden, hierbei schützen sie die internen Eigenschaften von externen unerlaubten Veränderungen. Wenn eine externe Entität einen Wert aus dem Aggregat anfragt, wird meist nur eine Kopie zurück geben.

Ein Beispiel für ein Aggregat in unserem Code ist die *TrackedInformation* Klasse, diese schützt den Zugriff auf die internen *Value Object* vor externen Veränderungen.

### 2.4.4 Entities

Entities sind Objekte welche von ihre Identität definiert werden und nicht durch ihre Werte. Ein Beispiel hierfür ist eine Person auch wenn sie ihr Gewicht ändert oder andere Eigenschaften wie der Nachname oder die Haarfarbe. Zudem sind zwei Personen trotz selben Namen unterschiedliche Individuen. Ein Beispiel eines Entities ist bei uns die Klasse *TrackedInformation*. In dieser Klasse wird die Identität durch eine einmalige ID festgestellt.

### 2.4.5 Value Objects

Unter Value Objects versteht man Objekte deren Gleichheit nicht auf Identität beruht. Zwei Value Objects sind gleich, wenn sie denselben Wert haben ohne notwendigerweise dasselbe Objekt zu sein. Zudem sind Value Objects nicht veränderbar.

In unserem Code haben wir die Klasse *Vector2D* als ein Value Object umgesetzt, welches alle Kriterien erfüllt. So wird bei der Überprüfung ob zwei Vektoren gleich sind ausschließlich auf ihre internen x und y Koordinaten geschaut und nicht auf ihre Identität. Zudem ist das Ergebnis einer mathematischen Operation zweier Vektoren ein neuer Vektor während die alten Vektoren unverändert bleiben.

## 2.5 Unit Tests

Unser Code verfügt über ca. 100 Unit Tests. In der folgenden Auflistung sind 10 Beispiele für unsere Unit Tests:

1. Vector2DTester.Copy\_AVector\_WillReturnAVectorWithANewReference()
2. Vector2DTester.Add\_TowVectors\_ReturnSumOfVecors()
3. ServiceCacheTester.AddItem\_AtMaxCacheSize\_willRemoveFirstItem()
4. ServiceCacheTester.AddItem\_WithAKeyThatAllreadyExist\_willThrowKeyAlready ExistExeption()
5. VelocityCSVTranslatorTester.GetCSV\_WithSameKey\_WillReturnTheSameIntance()
6. PluginManagerTester.Add\_Aplugin\_WillSendTheAddedPluginOverTheEventsystem()
7. AdapterTests.Copy\_ofAVectorCSVItem\_willReturnASimilarItem()
8. AdapterTester.Equals\_OfTwoSimilarVectorCSVItems\_WillReturnTrue()
9. ServiceCacheTester.AddItem\_AtMaxCacheSize\_willRemoveFirstItem()
10. Vector2DTester.Copy\_AVector\_WillReturnAEqualVector()

### 2.5.1 ATRIP-Regeln

Die Anwendung der Automatic, Thorough, Repeatable, Independent and Professional (ATRIP)-Regeln können zu guten Unit Tests führen. Die ATRIP-Regeln wurden in "Pragmatic Unit Testing" beschrieben. Im folgenden Abschnitt werden wir die Umsetzung der ATRIP-Regeln anhand unseres Codesubstrat verdeutlichen.

#### **Automatic - Eigenständig**

All unsere Test laufen automatisch durch, sodass keine manuelle Eingriffe nötig sind. Genauso haben alle Test nur zwei mögliche Endergebnisse, entweder sie haben bestanden oder nicht. Nach jedem Commit auf das Gitrepo, werden auch alle Tests durch gelaufen und wir werden benachrichtigt wenn mindestens ein Test fehlschlägt.

#### **Thorough - Gründlich (genug)**

Bei Thorough geht es um die Gründlichkeit der Unit Test, diese kann nicht mit 10 Test erreicht werden. Um hier gut abzuschneiden muss eine hohe Code Coverage erreicht werden, jedoch ist eine hohe Code Coverage nicht gleichzusetzen mit einem guten Thorough. Wichtig ist dabei, dass die wichtigsten Funktionalitäten der Applikation getestet werden. Da sich Fehler meist Gruppieren, testen wir bestimmte Bereiche stärker ab. Dies ist im Bereich des *Vectors* nötig. In unserem Codesubstrat weisen wir eine hohe Code Coverarge auf, nähere Details dazu in Abschnitt Code Coverage. Um zu überprüfen ob unsere Tests

auch wirklich gründlich testen haben wir auch zusätzlich Mutation Testing betreiben, mehr dazu im Abschnitt Mutation Tests.

### **Repeatable - Wiederholbar**

Alle Tests sollten jederzeit automatisch durchführbar sein und immer das gleiche Ergebnis liefern. Schlägt ein Test ohne Veränderungen spontan fehl, ist der Test fehlerhaft. Solch ein Problem hatten wir bei einem

Test (`Vector2DTester.ToString_OfAVector_WillReturnTheCorrespondingString()`). Hier hatten den Fall, dass je nach Betriebssystem Einstellungen Gleitkommazahlen entweder mit Komma oder mit Punkt dargestellt wurden. Jedoch hat zu diesem Zeitpunkt unser Test nur die Komma-Schreibweise akzeptiert. Dies ist aktuell nicht mehr der Fall. Jetzt prüft der Test die Einstellungen des Betriebssystems und passt die Daten des Tests an. All unseren anderen Test sind nach unserem aktuellen Wissen wiederholbar. Da wir beim Großteil der Test die Testdaten statisch hinterlegt haben, gibt es wenig Raum für solche Fehler.

### **Independent - Unabhängig voneinander**

Unit Test sollten generell von so wenig wie möglich abhängen. Unsere Tests sind alle unabhängig, dies wird zum Beispiel an dem Test

*`ServiceCacheTester.AddItem_AtMaxCacheSize_willRemoveFirstItem()`* veranschaulicht. Dies haben wir erreicht, indem kein Test den andern aufruft und wir keinen Funktionalität doppelt testen.

### **Professional - Mit Sorgfalt hergestellt**

Da wir erst am Anfang unserer Karriere stehen ist diese Frage nicht leicht zu beantworten. Für Informatik Absolventen weisen die Test eine hohe Professionalität auf. Dies wird unterstrichen, da wir alle Regeln der ATRIP-Regel erfüllen. Ebenfalls besitzt unsere Applikation eine sehr gut Testabdeckung und die Mutaions haben bei den Mutationtests eine geringe Überlebenschance.

## **2.5.2 Code Coverage**

Code Coverage ist die Testabdeckung des Codes. Hier gibt es Line Coverage, dabei wird jede Zeile Code durch die Tests entweder durchlaufen oder nicht. Wobei durchlaufen nicht

gleich gesetzt werden kann mit abgesichert. Dann gibt es noch Branch Coverage, hier wird jede Entscheidung im Code (if-statement) auf verschiedenen "Abzweigungen" durchlaufen bzw. verlassen.

Eine hohe Testabdeckung ist hilfreich, jedoch ist eine Abdeckung größer 80% sehr aufwendig und somit auch kostspielig. Daher ist ein wirtschaftliches Optimum zwischen Aufwand und Kosten wichtig. Eine Testabdeckung von 100% bedeutet nicht automatisch, dass alles getestet wurde. Dabei wurden zum Beispiel bei der Line Coverage nur alle Zeilen durchlaufen.

Wir haben eine Line Coverage von 87% und eine Branch Coverage von 71%, genauer Zahlen befinden sich in der Abbildung 2.4. Wir sind sehr zufrieden mit diesen Zahlen, da sie uns ein gutes Gefühl geben. Da der Großteil unserer Anwendung regelmäßig getestet wird, sollten auftretende Fehler schnell erkannt werden.

Name	Covered	Uncovered	Coverable	Total	Line coverage	Covered	Total	Branch coverage
+ BarbellTracker.AbstractionCode	101	0	101	163	100%	14	14	100%
+ BarbellTracker.AbstractionCodeTests	184	3	187	359	98.3%	3	4	75%
+ BarbellTracker.Adapter	164	49	213	408	76.9%	44	64	68.7%
+ BarbellTracker.AdapterTests	347	0	347	681	100%	16	16	100%
+ BarbellTracker.ApplicationCode	12	103	115	333	10.4%	1	18	5.5%
+ BarbellTracker.DomainCode	17	4	21	56	80.9%	4	4	100%
+ BarbellTracker.DomainCodeTests	40	0	40	74	100%	0	0	
+ BarbellTracker.Services	132	16	148	317	89.1%	21	22	95.4%
+ BarbellTracker.ServicesTests	468	0	468	801	100%	6	6	100%

Figure 2.3: Screenshot der Code Coverage unserer Anwendung (Barbell Tracker)

### 2.5.3 Einsatz von Mocks

Mock-Objekte sind einfache Stellvertreter für "echte" Objekte. Diese werden genutzt um Abhängigkeiten während eines Tests zu ersetzen. Diese Mocks sind nötig, um eine Klasse ohne ihre realen Abhängigkeiten isoliert testen zu können. Hierfür gibt es bestimmte Mock-Tools, die eine einfache Schnittstelle zur Verfügung stellen. Diese Tools können für den jeweiligen Einsatzzweck trainiert werden. Hierfür durchläuft jedes Mock drei Phasen. Die erste Phase ist das Einlernen danach kommt das Abspielen und zuletzt die Überprüfungsphase.

Für das Mocking nutzen wir das Framework *Moq*. Wir haben das Mocking Tool auf verschiedene Art und Weisen genutzt.

Fine Code Coverage	
Coverage	Summary
	Risk Hotspots
	Coverage Log
<b>Assemblies:</b>	9
<b>Classes:</b>	42
<b>Files:</b>	40
<b>Covered lines:</b>	1488
<b>Uncovered lines:</b>	217
<b>Coverable lines:</b>	1705
<b>Total lines:</b>	3524
<b>Line coverage:</b>	87.2% (1488 of 1705)
<b>Covered branches:</b>	111
<b>Total branches:</b>	156
<b>Branch coverage:</b>	71.1% (111 of 156)

Figure 2.4: Screenshot der Unit Test Summary unserer Anwendung (Barbell Tracker)

1. In der Klasse *PluginManagerTester* haben wir Mocks genutzt um ein anonyme Implementierung des Interfaces *IPlugin* zu erstellen, welches in der Lage ist ein vor definierten Namen zurückzugeben. Dies haben wir genutzt um den *PluginManager* mit Plugins zu füllen.
2. In der Methode *Add\_Aplugin\_WillSendTheAddedPluginOverTheEventsystem* welche in der Klasse *PluginManagerTester* liegt, haben wir ein Mock benutzt, welches ein Eventsystem simuliert. Das wichtige ist hierbei das wir überprüfen, ob die Methode *Fire* mit den richtigen Parameter aufgerufen wird. Ist dies nicht passiert schlägt der Test fehl.
3. In der Methode *RequestCSVWithTheGetCSV\_FromATrackedInformationObject\_WillReturnTheRigthCSVObject* in der Klasse *VelocityCSVTranslatorTester* nutzen



wir ein Mock, welches das *ICalculator* Interface darstellt und die Methode *GetCalculatedValue* implementiert. Hierbei definieren wir einen Input und einen Rückgabewert, welcher zurückgegeben wird wenn der vorher definierte Input hinein gegeben wird.

### 2.5.4 AAA-Normalform

Arrange, Act und Assert (AAA)-Normalform eines Unit Tests besitzt drei Phasen Arrange, Act und Assert. Bei Arrange wird die Test-Welt initialisiert und bei Act werden die testende Aktion ausgeführt. Die letzte Phase mit Assert wird die Aktion geprüft und das Ergebnis ausgegeben. Als Ergebnis kann Success, Failure oder Error erfolgen. Success bedeutet der Test wurde bestanden. Das heißt die Testmethode läuft durch und alle Assertions wurden erfüllt. Bei Failure wurde die Assertion nicht bestanden. Als letztes Ergebnis kann Error auftreten. Dies tritt auf, wenn ein Fehler eintritt der nichts mit der Assertion zu tun hat. Wie zum Beispiel eine Nullpointer Exception im zu testenden Code. Dies sollten aber keine gewollte Exceptions sein, diese müssen anders deklariert werden. Das bedeutet Errors sind zu vermeiden und zeigen einen Entwicklungsfehler im Code. Jeder Test in unserem Code ist in der AAA-Form. Wir haben uns dazu entschlossen die einzelnen Phasen mit Kommentaren zu kennzeichnen.

### 2.5.5 Welches Framework und warum

Für C# in .NET gibt es verschiedene Unit Test Frameworks. Diese sind MSTest, NUnit und xUnit. Wir haben uns für xUnit entschieden, da xUnit weit verbreitet ist und dadurch ein hohen Bekanntheitsgrad aufweist und auch für uns ein bekanntes Framework ist.

### 2.5.6 Mutation Tests

Um die Qualität unserer Tests noch einmal zu überprüfen, haben wir uns dazu entschlossen unsere Tests Mutation Testing zu unterziehen. Hierfür haben wir das Tool *stryker-mutator* benutzt. Die Ergebnisse der Mutationstests liegen auf unserem Github Repo unter dem Ordner *MutationTestResults*. Der prozentuale Anteil der getöteten Mutationen beträgt zwischen 51% bis 84%. Mit diesem Ergebnis sind wir sehr zufrieden, besonders da die überlebten Mutationen kaum eine Relevanz auf die Test Qualität haben.

In der folgenden Grafik 2.5 finden sie ein Beispiel für unsere Ergebnisse der Mutationstests.






File / Directory	i	Mutation score	# Killed	# Survived	# Timeout	# No coverage	# Ignored	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
 All files		<div><div>82.95%</div></div> 82.95	73	8	0	7	0	0	0	73	15	88
 Implementation		<div><div>82.46%</div></div> 82.46	47	5	0	5	0	0	0	47	10	57
 Interface		N/A	0	0	0	0	0	0	0	0	0	0
 KeyAlreadyExist.cs		N/A	0	0	0	0	0	0	0	0	0	0
 ServiceCache.cs		<div><div>83.87%</div></div> 83.87	26	3	0	2	0	0	0	26	5	31

Figure 2.5: Mutationtests der Service Schicht

## 2.6 Refactoring

Das Refactoring unseres Codes erfolgt in Pair-Programming. Das hat bestimmte Vorteile, zum einen können wir so zusammen über Code Smells und Naming Fragen diskutieren und eine gemeinsame Lösung finden und zum Anderen übersehen wir weniger Code Smells. Wir hatten zwei große Refactorings in unsern Projekt, welche im folgenden erläutert werden.

### 2.6.1 Beispiele für Refactoring

#### Von Singleton zu Dependency Injection

Zu Beginn der Planung unserer Applikation, haben wir uns drauf geeinigt ein paar Klassen als ein Singleton zu implementieren. Die zwei Singleton Klassen waren *BarbellTracker.Adapter.PluginManager* und *BarbellTracker.Adapter.UIAdapterManager*. Beides sind Klassen, welche für das Managen wichtiger Daten im Application Code zuständig sind. Hier war es notwendig, dass alle Klassen ein und die selbe Instanz nutzen, damit keine Informationen verloren gehen. Im ersten Moment erschien das Nutzen eines Singleton für diese Anforderung sinnvoll, besonderes da Microsoft es bei ihrer C# Dokumentation selbst als gutes Beispiel nennt. Beim Testen ist uns jedoch aufgefallen, dass es mit unserer aktuellen Umsetzung zu großen Problemen kam. Daher haben wir uns dazu entschieden, uns von dem Singleton Design Pattern zu trennen und es mit Dependency Injection zu lösen. Hierfür deklarieren wir die oben bereits benannten Singletons im Dependency Injection Container. Das Entfernen der Singletons wurden mit dem Commit ID: 0387ad74c092211ffe92ad8bbd4f87a998727e41 umgesetzt.

## Eventsystem

Die erste Umsetzung unseres Eventsystems, basierte auf einem Enum welches das Event angab. Das Enum hat mitgeteilt um welches Event es sich handelt und wurde mit den Argument (vom Typ Object) über das Eventsystem verschickt. Interessenten konnten hierbei bestimmte Enum Values (Events) abonnieren und wurden dann informiert, wenn dieses Enum losgeschickt wurde. An dieser Umsetzung gibt es gleich mehrere Aspekte zu kritisieren.

1. Das Hinzufügen neuer Events ist nur schwer zu realisieren, da hierfür ein neues Element beim Enum angelegt werden muss. Dadurch muss "häufig", also bei neu benötigten Events, der Applikation Code verändert werden.
2. Es gibt keine Überprüfung welcher Typ neben dem Enum über das System geschickt wird. Dies kann bei der alten Implementierung nur während der Runtime heraus gefunden werden. Dies führt dazu, dass das Erweitern oder Ändern des Verhaltens extrem schwierig zu implementieren ist.

Deswegen haben wir uns dazu entschieden das Eventsystem gegen ein generisches Eventsystem auszutauschen. Das Ziel ist, dass man sich auf ein beliebigen Typ abonniert. Wenn daraufhin ein Objekt mit genau diesem Typ über das Eventsystem geschickt wird, dann wird sein eingereichter Delegate aufgerufen. Diese Änderungen wurden im Commit ID: 5ae6aa5ecc0542942844ef3712f5032654b353dc umgesetzt.

### 2.6.2 Code Smells

Als Code Smell werden Programm-Abschnitte bezeichnet, die eine Überarbeitung des Quellcodes verlangen. Jedoch geht es nicht um Programm- oder Syntaxfehler, sondern um funktionierenden Code, der aber schlecht strukturiert ist. Meist drückt sich das durch schlecht lesbaren oder nur schwer verstehbaren Code aus. Durch Code Smells können sich bei Korrekturen oder Erweiterungen leicht neue Fehler einschleichen. Beispiele für Code Smell im Allgemeinen sind Code-Duplizierungen, lange Methoden oder große Klassen.

## Exception Handling

Wir haben ein Code Smell in der Klasse *FileManager* entdeckt. Hier wurden in einer Methode alle Exceptions abgefangen jedoch wollen wir nur die IOException abfangen. Dieser Code Smell wurde mit dem Commit ffba113a36aab5a7d365fc9d97467cb6abf08d78 behoben.

**Wert wird von Methode durchgereicht**

Ein weiteren Code Smell gibt es in der Methode *AddItemToCache* in der *Service* Schicht bei der Klasse *ServiceCache*. Hier wird ein Item dem Cache hinzugefügt. Der Methode wird dabei die *TrackedInformation* und das Item hineingereicht. Das Item wird benutzt und gleichzeitig ohne Veränderung wieder zurück gegeben. Dies erfolgt, da wir das Item in der Methode benötigen, in der wir auch die Methode *AddItemToCache* aufrufen. Das Problem dabei ist, dass wir nicht garantieren können, dass das Item von der *AddItemToCache* Methode auch wieder zurück geben wird. Ist das nicht der Fall, dann würde das Item Objekt fehlen. Dieser Code Smell wurde bis jetzt nicht behoben.

**Methoden klein geschrieben**

In der Klasse *Vector2D* wurden mehrere Methoden Namen klein geschrieben, diese sind unter anderem *Length()*, *Scalar()*, *DotProduct()* und *Normalize()*. Dieser Fehler ist in der Klasse *Vector2D* häufiger aufgetreten und wurde mit dem Commit 884809a82be2596d3b4d40ed78d834c38714fcff behoben.

# Chapter 3

## Extras

### 3.1 Build pipeline

Wir haben unser Projekt mit einer Build Pipeline ausgestattet. Diese startet jedes Mal, wenn ein neuer Commit auf dem Github Repo eingeht. Die Build Pipeline führt folgende Aktionen durch:

1. Bauen des Projektes auf der neusten Windows Version
2. Testen aller Tests
3. Deployen der WPF Applikation
4. Releasen der aktuellen WPF Applikation

Ein wichtiges Feature bei der Build Pipeline ist, dass wenn einer dieser Schritte fehlschlägt, alle eingetragenen Entwickler benachrichtigt werden. Das die aktuelle Version einen Fehler besitzt wird auf Github zusätzlich mit einem roten Pfeil angezeigt. Ebenfalls werden die Entwickler benachrichtigt sobald ein oder mehr Tests fehlschlagen.

Der Vorteil an unserer Build Pipeline ist, dass wir sicherstellen das unsere Versionen immer korrekt sind. Das ist wichtig, um zu überprüfen, dass die Versionen nicht nur auf den Entwickler PC's laufen und die Test regelmäßig laufen. Darüber hinaus gibt es einen weiteren Vorteil an der Build Pipeline, so können alle Interessierten die App testen, ohne selbst das Projekt bauen zu müssen. Dazu benötigt man lediglich die *exe*.

# Bibliography

- [1] Das model-view-viewmodel-muster. URL <https://docs.microsoft.com/de-de/xamarin/xamarin-forms/enterprise-application-patterns/mvvm>.