Kucoin Desktop Interface

Programmentwurf

für die Prüfung zum

Bachelor of Science

des Studiengangs Informatik
Studienrichtung Angewandte Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Jonas Etzbach

16.05.2022

Matrikelnummer 4880261

Kurs TINF19B4

Ausbildungsfirma cjt Systemsoftware AG, Karlsruhe

Gutachter der Ausbildungsfirma Mirko Dostmann

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

DDD	Domain Driven Design
GUI	Graphical User Interface
DTO	Data Transfer Object
WPF	Windows Presentation Foundation
API	application programming interface
MVVM	Model-View-Viewmodel

1. Domain Driven Design

1.1 Analyse der Ubiquitous Language

Die Ubiquitous Language ist Bezeichnung für die Fachsprache, welche bei der Softwareentwicklung dazu verwendet wird, um sich über die Domäne zu verständigen und diese zu konkretisieren. Sie wird sowohl von den Softwareentwicklern als auch von den Domänenexperten gesprochen.

Innerhalb des Kucoin Desktop Interface gibt es die Fachdomäne der Kryptowährungen. Diese umfasst dabei jedoch nicht viele Fachbegriffe, da die Komplexität der Domäne recht gering ist.

Der zentrale Begriff hierbei ist der Exchange (dt. Austausch, in der Finanzwelt auch Börse). Bei Kryptowährungen werden zwei Dinge als Exchange bezeichnet, einmal Handelsplattformen wie Kucoin und zum anderen die Wechselkurse der Kryptowährungen selbst, da diese Aufgrund der fehlenden Zentralität eigene Ökosysteme und damit Börsen bilden. Im Projekt wird der Begriff Exchange für den Wechselkurs zwischen zwei Kryptowährungen verwendet. Ein Exchange besitzt dabei ein Symbol, sowie einen Verlauf.

Das Symbol, dient als Identifizierung eines bestimmten Wechselkurses und besteht aus den Abkürzungen zweier Kryptowährungen, einer Basiswährung, in welcher der Preis angegeben wird und einer Fremdwährung, welche auf der Börse „ge- und verkauft“ werden kann.

Der Kursverlauf wird durch sog. Candles (Kerzen) dargestellt, welche den Kurs in bestimmten, regelmäßigen Zeitabständen darstellen. Eine Candle besteht hierbei aus den folgenden Werten:

- Opentime/Timestamp: markiert den Anfang des Intervalls
- Open: Eröffnungspreis des Intervalls
- Close: Schlusspreis des Intervalls
- High: höchster Preis des Wechselkurses innerhalb des Intervalls
- Low: niedrigster Preis des Wechselkurses innerhalb des Intervalls
- Base volume: der insgesamt gehandelte Wert, gemessen in der Basiswährung

- Foreign volume: der insgesamt gehandelte Wert, gemessen in der Fremdwährung

Des Weiteren kann ein Nutzer der Applikation ein Aktualisierungsintervall für die geöffneten Wechselkurse auswählen, oder die automatische Aktualisierung deaktivieren. Das Aktivierungsintervall wird als Updateinterval bezeichnet.

1.2 Analyse und Begründung der Verwendeten Muster

1.2.1 Value Objects

Ein Value Object stellt einen festen und nicht veränderbaren Wert dar. Sie werden rein über ihre Werte miteinander verglichen und besitzen kein ID als Vergleichseigenschaft. Value objects werden im Projekt durch records umgesetzt, welche automatisch sicherstellen, dass zur Erstellung eines Value Objects stets eine neue Instanz erzeugt werden muss.

Innerhalb der Domäne gibt es die folgenden Value Objects, welche sich zum Teil schon aus den, zuvor genannten Domänen Begriffen, ergeben:

- ExchangeSymbol: Besteht aus dem eigentlichen Symbol des Wechselkurses, sowie den Abkürzungen der beiden Kryptowährungen des Wechselkurses. Dabei darf das Symbol nicht leer sein und muss einem bestimmten Regulären Ausdruck entsprechen, die Abkürzungen ergeben sich dann aus dem Symbol selbst, sofern dieses dem Regulären Ausdruck entspricht.
- Candle: Stellt eine einzelne Candle aus der Domäne dar und dient damit als Werte-Tupel für die einzelnen Werte des Kursverlaufs. Dabei dürfen die einzelnen numerischen Werte, also Open, Close, High, Low, BaseVolume und ForeignVolume nicht negativ sein.
- UpdateInterval: Stellt das Aktualisierungsintervall für die geöffneten Kurse dar, und besitzt kann nur die Festgelegten Werte 30 Sekunden, 1, 2, 3, 5, 10, 15 oder 30 Minuten annehmen. Besitzt eine Eigenschaft Interval, welche das eigentliche Interval durch den C# internen Typ TimeSpan darstellt, sowie eine Name Eigenschaft, welche die Instanz beschreibt.
- ExchangeRequest: Stellt eine Request für einen Exchange dar und besteht aus einem ExchangeSymbol sowie einem Update Callback, wenn der Verlauf eines Exchange ändert

1.2.2 Entities

Entities sind im Gegensatz zu Value Objects durch, eine Identität unterscheidbare, Objekte. Des Weiteren besitzen Entities einen Lebenszyklus, was bedeutet, dass diese sich über die Zeit hinweg verändern können.

Innerhalb der Anwendung gibt es nur die Entity Exchange, welche wie in der Ubiquitous Language beschrieben ein Symbol, welche als ID dient, sowie einem Enum KlineInterval, welches die Intervalldauer der Candles beschreibt und dem Kursverlauf bestehend aus den eigentlichen Candles.

Exchanges können auf zwei varianten betrachtet werden, die erste Variante wäre Entities und die zweite wären Aggregates. Im Projekt wird nun der Exchange als Entity definiert, da sich der Verlauf (History) eines Exchange mit der Zeit durch regelmäßige Aktualisierung, sofern nicht vom User deaktiviert, was sich mit einem Lebenszyklus dieser vergleichen lässt.

1.2.3 Aggregates

Aggregates sind eine Gruppierung von Value Objects und Entities. Sie fassen diese zu einer gemeinsam verwalteten Einheit zusammen, um die Komplexität dieser zu reduzieren.

Innerhalb des Projekts gibt es nur die ExchangeHistory, welche eine Zusammenfassung von Candles in Kombination mit einem KlineInterval darstellt. Dadurch werden diese direkt mit einem entsprechenden Intervall verknüpft, was die Handhabung dieser Einheit erleichtert.

1.2.4 Repositories

Repositories schirmen die Domäne vom Datenmodell ab und stellen eine Schnittstelle zwischen den beiden Teilen dar. Sie bieten Methoden zur Verwaltung von und Interaktion mit den Datenobjekten, Entitäten und Aggregaten der Domäne im Speichermodell an.

Innerhalb der Applikation lässt sich das IExchangeRepository identifizieren. Dieses besteht aus Methodenhüllen zur Verwaltung von Exchanges und ermöglicht das asynchrone Erhalten und Löschen von Exchanges. Dieses Verschleiern dann die Verwendung der Kucoin API, auch ist es dadurch möglich die Kucoin API

beispielsweise durch eine andere zu ersetzen, ohne dass eine Anpassung der Codestellen, welche darauf zugreifen notwendig ist.

2. Clean Architecture

Innerhalb des Kucoin Desktop Clients ist die Clean Architekt in die folgenden Ebenen unterteilt:

- Domain-Schicht
- Application-Schicht
- Adapter-Schicht
- Plugin-Schicht

Diese sind in der oben dargestellten Reihenfolge aufgebaut und

2.1 Domain-Schicht

Die Domain Schicht enthält die fachlichen Regeln, welche innerhalb des Projektes gelten.

Innerhalb des Projektes sind das die in Kapitel 1.2 vorgestellten Objekte, sowie die, bei den Value Objects vorhandenen Factories. Hierbei sei erwähnt, dass aufgrund der smarten GUI, welche eigentlich in die Plugin-Schicht gehört, diese teilweise auf die Value-Objects der Domain-Schicht zugreift. An dieser Stelle wäre jedoch eine Trennung der beiden Schichten ein größerer Aufwand, als sich lohnen würde. Was sich am Beispiel des UpdateIntervals zeigt.

2.2 Use Case-Schicht

Die Use Case-Schicht (alternativ auch Application-Schicht) enthält die Anwendungsfälle (Use Cases) der Anwendung. Hier befindet sich die Geschäftslogik auf Basis der Projektanforderungen. Wodurch sich die Steuerung der Domäne ergibt.

Ein Beispiel hierfür innerhalb des Projektes wären die LoadAvailableExchangeSymbols und die LoadExchange Klassen, welche die gleichnamigen Use Cases umsetzen.

Innerhalb der Applikation wurde sich dagegen entschieden die einzelnen Use Cases in Form von einer Klasse zusammenzufassen, da die fasst alle Use Cases auf die einzige Entität der Domäne zugreifen, wodurch die Klasse sehr groß werden würde.

Durch den Verzicht auf die Zusammenlegung der Use Cases bleiben die Klassen übersichtlich und der Code ist leichter wartbar.

2.3 Adapter-Schicht

Die Adapter-Schicht bildet die Schnittstelle zwischen der Plugins- und Use Case- und Domain-Schicht.

Dies wird durch Adapter-Klassen umgesetzt, welche mit Mappern gleichgesetzt werden können. Hierbei sei erwähnt, dass aufgrund der fehlenden, direkten Serialisierung der Daten die Konvertierung nur in eine Richtung stattfindet, wodurch es auch keine Data Transfer Objects (DTO) gibt die Implementiert werden müssen. Das einzige, vorhandene DTO - KucoinKline - wird durch die verwendete Bibliothek bereitgestellt. Innerhalb der Anwendung wird das DTO zuerst in Value-Objects der Domäne konvertiert und dann für die Darstellung durch entsprechende Adapter in ein, für die GUI Verarbeitbares Objekt konvertiert. Die hierfür notwendigen Adapter befinden sich im Adapters Namespace.

2.4 Plugin-Schicht

Die Plugin-Schicht greift auf die entsprechenden Adapter und verwendeten in dieser Schicht verwendeten Frameworks zu. Hierdurch werden die Hintergrund-Operationen implementiert. Die verwendeten Frameworks stellen dabei Methoden und Funktionen zur Verfügung, um den Programmieraufwand zu reduzieren und die Anwendung von den direkten technischen Details abzukapseln.

Innerhalb des Projekts werden hierbei die folgenden Projekte verwendet:

- WPF: GUI Framework für C#
- Livecharts2: zur Darstellung der Kurse mittels WPF
- Kucoin.NET: C# Bibliothek für den Zugriff auf die Kucoin-Web-API

Die Frameworks befinden sich alle vollständig in der Plugin-Schicht. Die einzigen Abhängigkeiten der Frameworks bestehen zwischen Livecharts und WPF, diese Befinden sich jedoch auch beide, da sie zur GUI gehören in der Plugin-Schicht.

3. Programming Principles

3.1 SOLID

3.1.1 Single-Responsibility Principle

Nach diesem Prinzip haben verschiedene Module/Klassen/Objekte eine einzige Verantwortung oder Aufgabe.

Ein Beispiel hierfür innerhalb des Projektes sind die Adapterklassen, deren einzige Aufgabe die Konvertierung eines Objektes einer Klasse in ein Objekt einer anderen Klasse ist. Ein konkretes Beispiel hierfür wäre die FinancialCandleConverter Klasse.

3.1.2 Open/Close Principle

Das Open-Close Prinzip gibt vor, dass Klassen für Erweiterungen (Veränderungen) offen aber für Modifikationen geschlossen sind. Bestehender Code soll also nach Möglichkeit nicht verändert werden, eine Erweiterung vorhandener Funktionen soll aber dennoch möglich sein ohne, dass ein Bruch in der Applikation entsteht.

Im Projekt wurde dies innerhalb des Value Objects UpdateInterval erreicht, um ein neues Aktualisierungsintervall für die Aktualisierung der Wechselkurse zu erstellen, muss nur eine entsprechende Instanz in den Array hinzugefügt werden.

3.1.3 Liskov Substitution Principle

Das Liskov Substitution Prinzip besagt, dass eine Basis Klasse immer durch ihre Ableitung ersetzbar sein soll, ohne dass es dabei zu einem logischen Bruch der Applikation kommt.

Aufgrund der fehlenden Verwendung tieferer Vererbung innerhalb des Projektes wird konnte dieses Prinzip nicht vollständig, ein Ansatz hierfür findet sich jedoch mit dem DelayedInitialisionViewModel und dem ExchangeSelectionViewModel, würde dieses nicht per Binding statisch in die GUI eingebunden werden, sondern über eine dynamische Bindung an die GUI gekoppelt werden, würde es nicht zu einem logischen Bruch kommen.

3.1.4 Interface Segregation Principle

Das Interface Segregation Prinzip gibt vor, dass statt einem großen generischen Interface viele kleinere, auf den spezifischen Fall bezogene Interfaces zu erstellen, um eine bessere Wartbarkeit und Modularität zu erreichen. Des Weiteren wird hierdurch eine klare Aufgabenverteilung erreicht.

Im Projekt wurde dies beispielsweise die Interfaces IUseCase, IContextBasedUseCase<TContext> und IQueryUseCase<TParam, TResult> erreicht. Dadurch ist es möglich Use Cases zu definieren die genau für einen spezifischen Fall wichtig sind, auch wenn diese generisch sind, da sie in ihrer Funktionalität noch recht gering sind:

- IUseCase: Aktion ohne Parameter oder Rückgabewert
- IContextBasedUseCase<TContext>: von einem Kontext abhängige Aktion
- IQueryUseCase<TParam, TResult>: Query-Anfrage mit einem entsprechenden Parameter und einem Ergebnis

Ein weiteres Beispiel für das Interface Segregation Prinzip sind die verschiedenen CallBack Interfaces ISubscription- und IExchangUpdate- und IExchangeSymbolsUpdateCallBack durch die es möglich ist für die individuellen Fälle speziell ausgewählte CallBacks zu programmieren, ohne, dass eine generische Klasse für alle Fälle geschrieben werden muss.

3.1.5 Dependency Inversion Principle

Das Dependency-Inversion-Prinzip besagt, dass Abstraktion von nie von konkreten Details abhängig sein dürfen, sondern nur die konkrete Implementation von den Abstraktionen.

Dieses Prinzip wurde durch die Schichtarchitektur der Clean-Architecture in der Applikation umgesetzt. Ein konkretes Beispiel, ist dass die Use Cases, sofern sie ein Repository benötigen, nur von diesem Abhängig sind, jedoch keinerlei Abhängigkeit zu der GUI besitzen, welche die Use Cases in entsprechenden Fällen direkt oder indirekt aufruft.

3.2 GRASP

3.2.1 Information Expert

Information Expert besagt, dass für eine Aufgabe derjenige zuständig sein soll, der das meiste Wissen darüber besitzt.

Ein Beispiel hierfür ist die ExchangeIsOpen Klasse, welche das ISpecification Interface für den Typ ExchangeSymbol implementiert. Wenn man wissen will, ob ein Exchange offen ist, wie das beim SelectExchange-Use Case der Fall ist.

3.2.2 Creator

Nach dem Creator Pattern wird festgelegt, wer für die Instanz-Erzeugung einer Klasse zuständig ist.

Beispiele hierfür sind die Factory-Klassen der Value Objekte Candle und ExchangeSymbol, welche diese Klassen als einzige instanziieren können.

3.2.3 Controller

Der Controller ist zuständig die GUI Events an die nicht GUI-Klassen weiterzuleiten und stellt damit die erste Klasse nach der GUI dar.

Im Projekt wurde dies durch die Implementation des Model-View-Viewmodel (MVVM) Architektur-Patterns, sowie die darüber angebundenen Commands, umgesetzt, da WPF dieses Architektur-Pattern besonders gut unterstützt. Die konkreten Beispiele sind die MainViewModel und die ExchangeSelectionViewModel Klassen, sowie alle nicht abstrakten Klassen im Commands Namespace.

3.2.4 Indirection

Indirection als Pattern verringert die Kopplung zwischen Klassen, durch den Einsatz eines Vermittlers zwischen den Klassen. Dadurch erhält man auch eine größere Flexibilität als bei der Vererbung.

Ein Beispiel innerhalb des Projektes hierfür sind die Adapterklassen, welche die zwischen den Klassen der Plugin- und der Domain-Schicht vermitteln und Konvertierungen durchführen können.

3.2.5 Low Coupling

Bei dem Low Coupling Prinzip geht es darum, die Abhängigkeit, zwischen den Klassen und Modulen zu verringern und möglichst gering zu halten.

Dies ist jedoch nicht immer einhaltbar, wie anhand der ExchangeSelectionViewModel-Klasse erkennbar ist, welche eine direkte, hohe Abhängigkeit von vielen Klassen besitzt. Auf der anderen Seite, wird durch das ViewModel auch Abhänigkeit zwischen dem Model (Daten) und dem View, reduziert.

Ein Beispiel innerhalb der Anwendung für Low Coupling wären die Adapter, welche für eine geringe Abhängigkeit zwischen den Schichten sorgt.

3.2.6 High Cohesion

Anhand der Kohäsion wird der logische Zusammenhang zwischen den Klassen innerhalb des Projektes gemessen. Eine hohe Kohäsion innerhalb eines Projekts sorgt für eine geringere Komplexität aufgrund von weniger, unterschiedlichen Funktionen und sorgt damit für eine höhere Wartbarkeit.

Ein Beispiel innerhalb des Projektes hierfür ist das KucoinExchangeRepository, welches in der GetExchange Methode die GetExchange Methode des IExchangeUpdaters, welcher Teil des ITickUpdaters ist, verwendet.

3.2.7 Polymorphism

Das Polymorphism-Prinzip besagt, dass Methoden mit identischer Signatur beim Zugriff auf diese, unterschiedliche Ergebnisse erzeugen. Methoden mit gleichem Namen Parameter und Rückgabewert werden also unterschiedlich realisiert.

Ein Beispiel innerhalb des Projektes hierfür sind die ExchangeUpdate- und die ExchangeSymbolsUpdate-Klasse, welche jeweils eine andere Implemtation für die OnTickUpdate Methode besitzen.

3.2.8 Protected Variations

Das Protected Variations-Pattern sorgt dafür, dass Klassen, Methoden und Module vor Änderungen anderer geschützt werden. Dies geschieht durch Interfaces, welche die Konkreten Implementationen verstecken.

Innerhalb des Projektes geschieht dies durch das IExchangeRepository-Interface, da dieses die Implementation über die Kucoin API versteckt. Dadurch wäre es auch möglich anstelle der Kucoin API die API einer anderen Krypto Börse zu verwenden, oder eine andere Verwendungs-/Zugriffsform der Kucoin API zu verwenden.

3.2.9 Pure Fabrication

Das Pure Fabrication-Pattern stellt das Konzept von Hilfsklassen dar, welche so nicht direkt in der Projektdomäne existieren, jedoch benötigt werden. Sie dienen zur dem low coupling und dem high cohesion Prinzip.

Ein ideales Beispiel hierfür ist das IExchangeRepository, welches die Funktionen, woher die Kursdaten bezogen werden, versteckt. Ein wirkliches Repository in der Art existiert nicht, es dient rein der Verschleierung der API-Aufrufe.

3.3 DRY

DRY steht für Don’t Repeat Yourself. Das DRY-Pattern hat das Ziel der Vermeidung von Redundanz durch Auslagerung von Code in eigene Methoden und Klassen.

Ein Beispiel für innerhalb der Anwendung ist die statische ThrowHelper-Klasse, welche statische Methoden zur Bedingungsprüfung und Werfen von Exceptions bei Fehlschlag dieser beinhalten, mit speziell auf die Bedingung zugeschnittenen Nachrichten.

4. Refactoring

4.1 Code Smell 1 – ExchangeSymbol Instanziierungskontrolle

Commits:

e4c472f037beabf2859fbb15fdcc1f52f0d61bb7

058b0b6d72ebe764c32d72ff7ba86501715a35c7

Vor dem Refactoring war es möglich eine Instanz des Value-Objects ExchangeSymbol zu erstellen, ohne dass hierfür die Factory verwendet werden musste, da die Factory Klasse außerhalb des Records definiert wurde und der entsprechende Konstruktor des ExchangeSymbols daher nicht als private deklariert werden konnte. Hierdurch war es möglich die Bedingungen, die für ein ExchangeSymbol gelten zu umgehen. Somit wären beispielsweise folgende Angaben für ein ExchangeSymbol valide gewesen:

- TradingCurrency = „Lieber“
- BaseCurrency = „Programmierer“
- Symbol = „das hier ist ein Bug!“

Was zu Fehlern in der Applikation führen würde und den in 1.Domain Driven Design definierten Anforderungen widersprechen würde. Um dieses Problem zu lösen, wurde die Definition der ExchangeSymbolFactory-Klasse in die Definition des Record-Typen verschoben. Um eine Trennung der beiden Typen dennoch möglichst gut zu ermöglichen wurde der Record-Typ als partial definiert, wodurch eine Dateitrennung der Typen ermöglicht wurde. Des Weiteren wurde der Konstruktor des ExchangeSymbol-Records als privat markiert, wodurch dieser nur noch von der Factory aufgerufen werden kann und die Anforderungen wieder gewährleistet werden können.

4.2 Code Smell 2 – Candle Instanziierungskontrolle

Commits:

cd9c6adf46fc141d56fd5d509ffc2617c9e83519

Vor dem Refactoring gab es keine Factory-Klasse für das Candle-Value-Object, und es fanden keine Werte-Überprüfungen für die Instanziierung der Candles statt, was zu einem logischen Bruch hätte führen können, in dem die Werte Fehlerhaft übermittelt worden wären, bspw durch einen Low-Price, der höher ist als der High-Price, was natürlich absolut sinnfrei wäre.

Um dies zu verhindern, wurde eine entsprechende Factory analog zu der in 4.1 erstellt, welche dafür sorgt, dass die entsprechenden Bedingungen eingehalten werden, um einen logischen Bruch in der zu vermeiden.

4.3 Code Smell 3 – Dupplicated Code

Commits:

ddbe218e4e7931e5b04f02326688f001df90e3c6

Das Refactoring des Code Smells 2 führte dazu, dass sehr viel redundanter Code geschrieben wurde, um die einzelnen Bedingungen zu überprüfen und Fehler zu werfen.

Um die Einhaltung des DRY Patterns zu gewährleisten, wurde daher eine Hilfsklasse ThrowHelper in dem Sub-Namespace Domain.Util erstellt, welche Methoden bereitstellt, um diese Bedingungen zu prüfen.

5. Unit-Tests

Unit-Tests tragen nicht direkt zur Applikation selbst bei, sorgen allerdings dafür, dass Anforderungen an die Applikation eingehalten werden können und die Funktionalität des Quellcodes auf Fehler überprüft wird und diese dadurch vermindert werden.

Die Tests wurden daher auch in einem separaten Projekt geschrieben, um die Test von der eigentlichen Anwendung zu Trennen.

5.1 Beachtung der ATRIP-Regeln

- **Automatic:** Gegeben, da alle Tests einzeln oder zusammen über die Entwicklungsumgebung Visual Studio gestartet werden können.
- **Throrough:** Gegeben,
- **Repeatable:** Gegeben, da die Tests immer wiederholt werden können und immer das gleiche Ergebnis haben, da frei von Zufallswerten sind.
- **Independent:** Gegeben, durch die strikte Trennung der Tests, in Klassen und Methoden, welche sich nicht gegenseitig referenzieren.
- **Professional:** Es werden nur relevante Stellen getestet (Nicht Getter und Setter/Properties) und auch keine Stelle doppelt getestet.

5.2 Mocks

Mocks werden bei den Use Case-Tests Mocks eingesetzt, um die Abhängigkeit von den benötigten Interfaces, wie den CallBacks und den Repositories zu entfernen.

6. Entwurfsmuster

6.1 Observer-Pattern

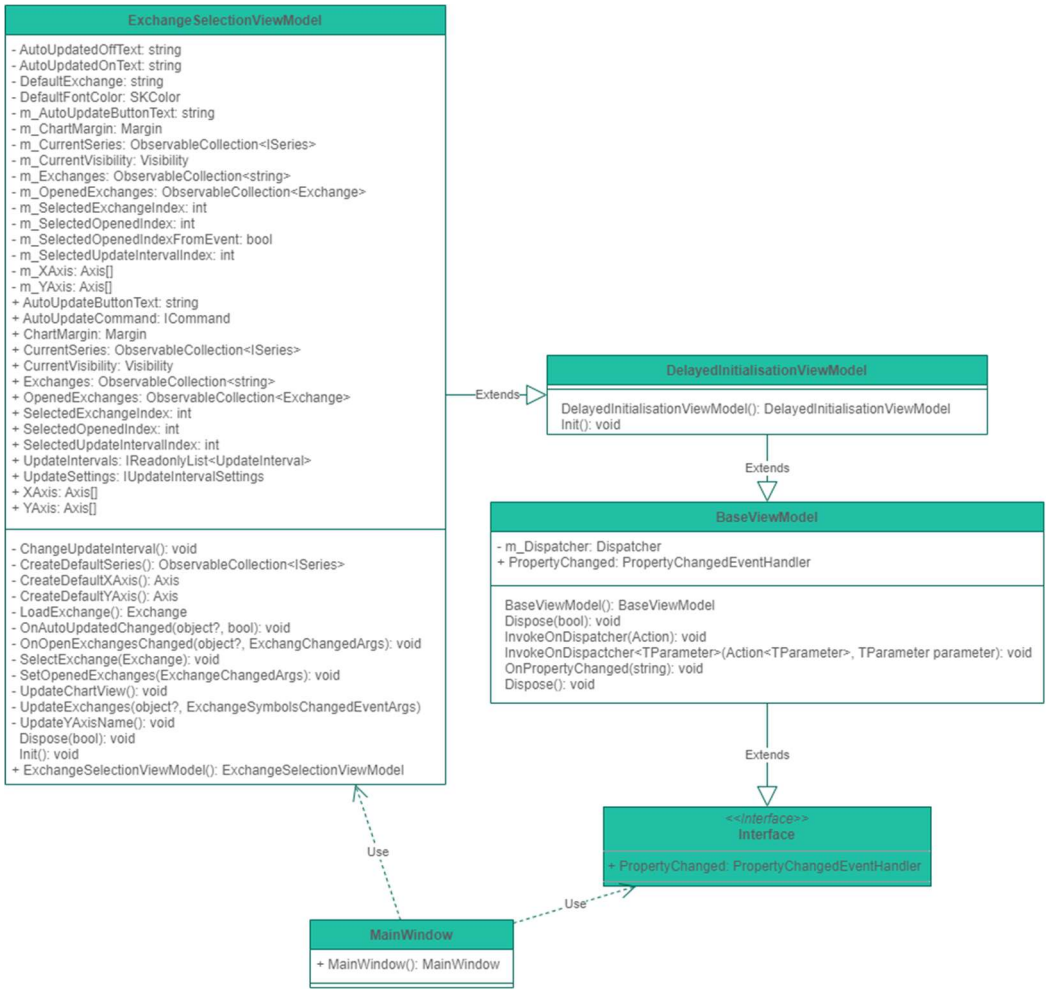

Abbildung 1: UML-Klassendiagramm des Observer-Patterns

Das Observer-Pattern (dt. Beobachter-Pattern) ist eine Form des Publish/Subscribe-Patterns bzw. des Ergeignisbeobachters. Objekte oder Methoden registrieren sich, um ein Ereignis zu erkennen.

Innerhalb des Projektes wird das Observer-Pattern beispielsweise bei der Verbindung zwischen der GUI-Klasse MainWindow und der ExchangeSelectionViewModel-Klasse verwendet. Dabei werden graphische UI-Elemente an die Properties des ExchangeSelectionViewModels gebunden, welches in der Basis Klasse das INotifyPropertyChanged Methode implementiert. Properties, welche die Methode OnPropertyChanged des Interfaces und damit das Event PropertyChanged aufrufen, welches vom WPF-Framework verwendet wird, um die Entsprechenden GUI Darstellungen zu aktualisieren, stellen dabei den Publisher dar. Der Observer ist dabei das Framework beziehungsweise die GUI-Klasse.