2021

Advanced Software Engineering

STOCK TRACKER

SEBASTIAN MOMANN

Inhalt

Technologieauswahl

TypeScript

JavaScript ist eine sehr einfache und dynamische Skriptsprache. Für den Anwendungsfall einer Webanwendung ist JavaScript hervorragend geeignet. Diese Sprache wurde eben für diese Art von Anwendung entwickelt. Sie hat ihren Ursprung im Web.

Aus anderen Programmiersprachen wie Java bin ich es gewöhnt mit Typisierung zu arbeiten. Typisierung bedeutet, dass Variablen ein Typ zugewiesen werden kann. Es ist nicht möglich einen Wert eines anderen Typs der Variablen zuzuweisen. Hierdurch kann man eine hohe Sicherheit und Fehlerresistenz sicherstellen, da Falschzuweisungen abgefangen und behandelt werden können.

Um nun eine solche Typisierung auch in JavaScript verwenden zu können, habe ich mich dazu entscheiden TypeScript zu verwenden. TypeScript erweitert JavaScript und bietet neben der Typisierung weitere Features, um das Programmieren mit der Sprache JavaScript angenehmer zu gestalten.

Webpack

TypeScript selbst kann nicht vom Browser ausgeführt werden. Um eine in TypeScript geschriebene Anwendung auszuführen, muss der Quellcode zunächst in JavaScript übersetzt werden. Browser können nur dieses übersetzte Script ausführen. Das Übersetzen übernimmt in dieser Anwendung die Bibliothek webpack. Webpack ist in erster Linie dazu gedacht, die vielen Quellcode Dateien in ein Bündel zu konvertieren und wo möglich auch zu komprimieren. Dies ist sinnvoll, um lange Ladezeiten zu vermeiden. Ebenso kann bei Bedarf auch Lazy Loading eingesetzt werden. Diese Bündelung ist für eine Webanwendung sehr vorteilhaft, weswegen ich von diesem Feature Gebrauch mache.

Des Weiteren bietet webpack die Möglichkeit Stylesheets (CSS) zu übersetzen. In diesem Projekt wird mit einer SCSS Datei gearbeitet, um HTML Elemente zu stylen. SCSS ist wie TypeScript nur ein overlay und muss in ein für den Browser verständliches Format gebracht werden. Bei styling ist dies CSS.

Webpack ist somit eine optimale Wahl um die 3 Anforderungen zu lösen. Das Übersetzen von TypeScript in JavaScript, das Übersetzten von SCSS in CSS und das kompakte Bündeln der JavaScript Dateien.

LocalStorage

Für die Datenspeicherung wird in dieser Anwendung die lokale Speicherung auf dem Gerät des Benutzers zurückgegriffen. Mittels JavaScript kann man ein Key-Value Storage (localStorage) ansprechen. Hier werden alle Daten gespeichert. Bei Aktiendaten handelt es sich um sehr kritische Daten. Möchte man diese auf einer externen Datenbank speichern, muss eine sichere Verbindung und im Optimalfall eine verschlüsselte Speicherung stattfinden. Dieses Projekt ist jedoch zu klein für einen solchen Aufwand, weswegen hierfür zunächst die lokale Speicherung als wirtschaftliche und sichere Methode verwendet wird. In Zukunft kann dank der

angewandten Programmierprinzipien die Speichermechanik ohne großen Aufwand ausgetauscht werden.

Jest

Um die Anwendung zu testen ist ein einfaches Framework sinnvoll. Da der Quellcode in TypeScript geschrieben ist, ist es auch sinnvoll, diese Sprache beizubehalten. Bei der Auswahl des Frameworks habe ich zwischen Mocha und Jest entschieden. Die Entscheidung fiel auf Jest. Zwischen den Frameworks gibt es keine signifikanten Unterschiede in der Syntax. Ebenso sind die bereitgestellten Funktionalitäten bis auf Einzelheiten dieselben. Der große Vorteil von Jest gegenüber Mocha ist, dass Jest keinen Browser für die Ausführung benötigt. Ebenso können Tests parallel ausgeführt werden. Mocha hingegen setzt auf sequenzielle Ausführung. Im scope dieses Projekts (10 Test) bringt die parallele Ausführung keinen signifikanten Mehrwert, ist aber bei Fortführung des Projekt eine sinnvolle Eigenschaft.

Ts-mockito

Jest ist nicht, beziehungsweise nur sehr eingeschränkt in der Lage Objekte zu mocken. Hierfür wird eine extra Bibliothek benötigt. Ts-mockito bietet sich als Lösung an. Mit dieser Bibliothek können mocks simpel und deklarativ erstellt werden. Ebenso können die mocks mehrfach verwendet und somit ausgelagert werden. Der Code der mocks ist ebenso Framework und Test-Framework unabhängig.

SOLID

Single Responsibility Principle

Jede Klasse hat einen spezifischen Aufgabenbereich. So gibt es Die Renderer Klassen, welche eine einzige Informationsquelle haben, um diese in einer HTML Struktur darzustellen. Ein Renderer kann weitere Renderer beinhalten. Der *HistoryRenderer* ist dazu gedacht, alle Transaktionen einer Aktie aufzulisten. Gleichzeitig muss auch eine Transaktion hinzufügbar sein. Das eigentliche Hinzufügen ist jedoch eine andere Aufgabe, welche nicht vom *HistoryRenderer* ausgeführt werden soll. Hierzu gibt es einen eigenen Renderer, welcher das HTML Layout für das Hinzufügen einer Transaktion erstellt und die Logik übernimmt.

Neben den Renderern gibt es Router, welche initial - abhängig von der URL - das Laden der Website übernehmen und die nötigen Renderer ansprechen.

Open / Closed Principle

Je nach dem, in welchem Land das Unternehmen einer Aktie beheimatet ist, kann sich die Währung einer Aktie unterschieden. Die Trading API gibt den Preis immer in der Heimatwährung zurück. Somit ist es notwendig, dass der Preis einer Aktie für die Anzeige in die Währung des Euro umgerechnet wird. Eine eigene Klasse für die Währung gewährleistet so, dass immer der Preis in Euro zurückgegeben wird. Das Summieren aller Transaktionskosten ist somit leicht möglich. Ebenso können neue Aktien mit anderen Währungen hinzugefügt werden, ohne das Summieren anpassen zu müssen.

Ebenso gibt es mehrere Transaktionsarten. Aktuell beschränkt sich eine Transaktion auf das Kaufen und Verkaufen. Beim Zusammenrechnen aller Transaktionen muss somit unterschieden werden, ob sich die Transaktion positiv oder negativ auf das Konto auswirkt. Diese Auswirkung ist in jeder Transaktionsklasse spezifiziert, wodurch die Transaktionen ohne weitere Verarbeitung verrechnet werden können. Auch hier lässt ich eine neue Transaktionsart, die sich ggf. anders auf das Konto auswirkt, leicht einfügen.

Liskov Substitution Principle

Die Basisklasse *Stock* bildet eine Aktie, beziehungsweise ein Aktienunternehmen ab. Da die Unternehmen in unterschiedlichen Ländern beheimatet sind, unterschieden sich die Währungen. Um nun mit den unterschiedlichen Währungen umgehen zu können, gibt es eine abgeleitete Klasse je Währung. So wird z.B. eine amerikanische Aktie als eine *USD_Stock* angelegt. Diese Klasse überschreibt die Basisklasse *Stock*. Bei der Abfrage des Preises wird die Währung in Euro umgerechnet. Somit kann bei Berechnungen die Funktion der Basisklasse angesprochen werden. Die Berechnung kann aber auch ohne Probleme mit den abgeleiteten Klassen *USD_Stock* und *EUR_Stock* durchgeführt werden.

Interface Segregation Principle

Jede Klasse, welche in Interface implementiert, benutzt diese Funktionen auch. Selbst die Klassen, die eine Klasse mit einem Interface erweitern, benötigen diese Klassen.

Dependency Inversion Principle

Keine Klasse baut auf eine andere Klasse auf. Kein Konstruktor beinhaltet eine festgelegte andere Klasse. Abhängigkeiten können als Parameter übergeben werden. Somit können Funktionalitäten ohne Probleme ausgetauscht werden.

GRASP

Coupling

Durch das Unterteilen in unterschiedliche Rendermodule, hängen die einzelnen Klassen nicht voneinander ab. Hält sich eine Klasse an die vorgegebenen Interfaces, so ist es für das Rendermodul kein Problem, die übergebene (oder aus der Datenbank geladene Klasse) darzustellen. Jedes Rendermodul lässt sich eigens aufrufen und anzeigen, sofern eine passende Referenz erstellt wurde, an welche der Renderer seinen Inhalt anheften kann.

Cohesion

Hohe Kohäsion wird hier durch das Unterteilen in unterschiedliche Klassenarten gesichert. Die Haupttypen sind hierbei Renderer, welche sich um das Erstellen von HTML Strukturen kümmern, Datenmodellen, und Datenstrukturen. So sind Datenmodelle hauptsächlich Atomare Modelle und Datenstrukturen eine Sammlung von Datenmodellen. Eine Aktie ist hierbei ein Modell, eine Historie jedoch eine Struktur. Natürlich gibt es auch Mischformen wie zum Beispiel die Transaktion, welche eine Referenz auf einem Modell beinhaltet.

DRY

Durch das Auslagern von Code in extra Methoden oder Funktionen kann Duplizierung vermieden werden. Ein Renderer muss in der Lage sein, Basis HTML Elemente zu generieren. Hierzu wird häufig derselbe Code benötigt. Dies hat zur Folge, dass bei einer Umstellung auf andere Basiselemente oder eine allgemeine Anpassung dieser, nur eine Stelle angefasst werden muss. Ebenso muss man den Code nicht an mehrere Stellen kopieren. Der Code wird hierdurch strukturierter und übersichtlicher. Für das Konkrete Beispiel ist es dahingehend sinnvoll, da ich mir vorbehalten möchte, das Design grundlegend zu ändern. Hierbei kann es passieren, dass ein HTML Element ähnlich wie bei Material Components aus mehreren Unterkomponenten besteht. Anstelle das nun bei jeder Verwendung anzupassen, habe ich die Möglichkeit nur eine Stelle zu adaptieren, um die Änderung durchzusetzen.

Wichtig ist, dass sich hier nur auf Basiselemente bezogen wird. Elemente wie *table-div* kommen wiederholt im Code vor. Diese habe ich aus Gründen der Leserlichkeit und der notwendigen hohen Parameteranzahl nicht generalisiert.

Entwurfsmuster

Factory-Pattern

Zur Erleichterung und Lesbarkeit des Codes, habe ich mich dem Factory Pattern bedient. Mit dem Factory Pattern ist es mir möglich, eine Transaktion zu erstellen. Basierend auf den Übergebenen Parametern wird ein Objekt der Unterklasse einer Transaktion angelegt. Bei Transaktionen wird aktuell zischen einer Kauf- und einer Verkaufstransaktion unterschieden.

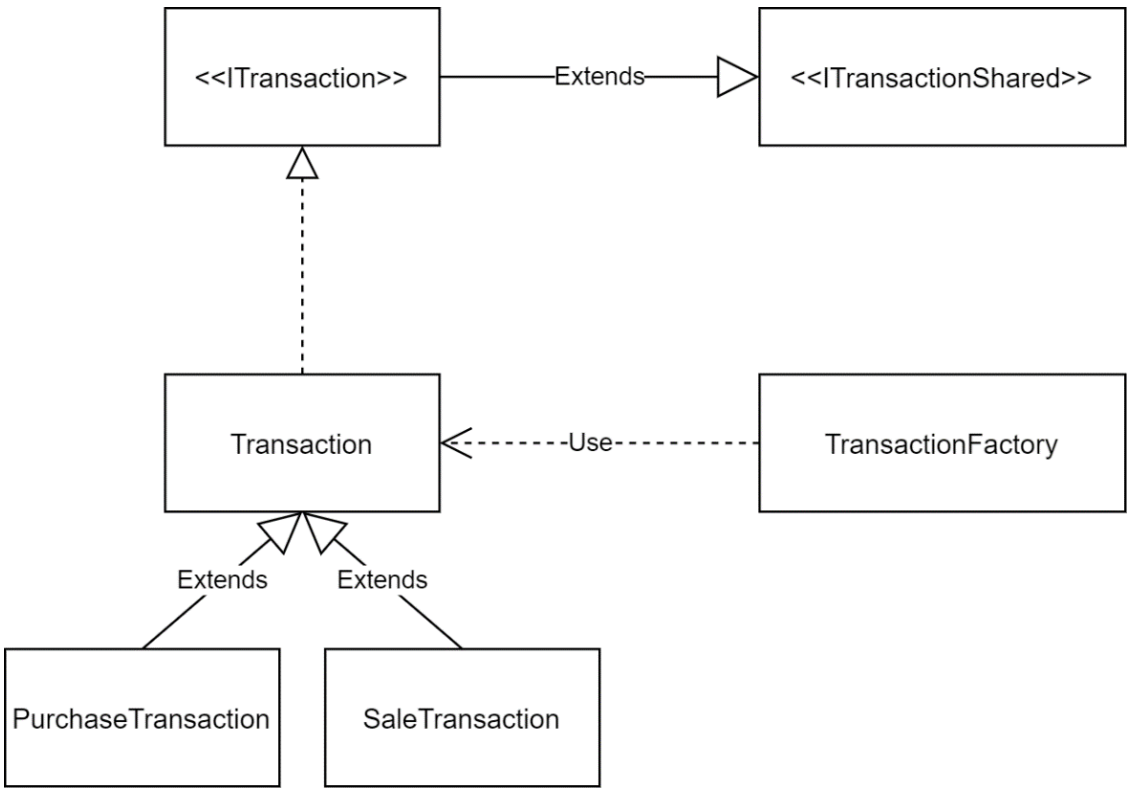
Die Factory wird aufgerufen, wenn der Benutzer eine neue Transaktion zu seiner Liste hinzufügen möchte. Hierbei wählt der Benutzer in einem Dropdown Menü aus, ob es sich bei der Transaktion um einen Kauf oder einen Verkauf handelt. Im Basierend auf

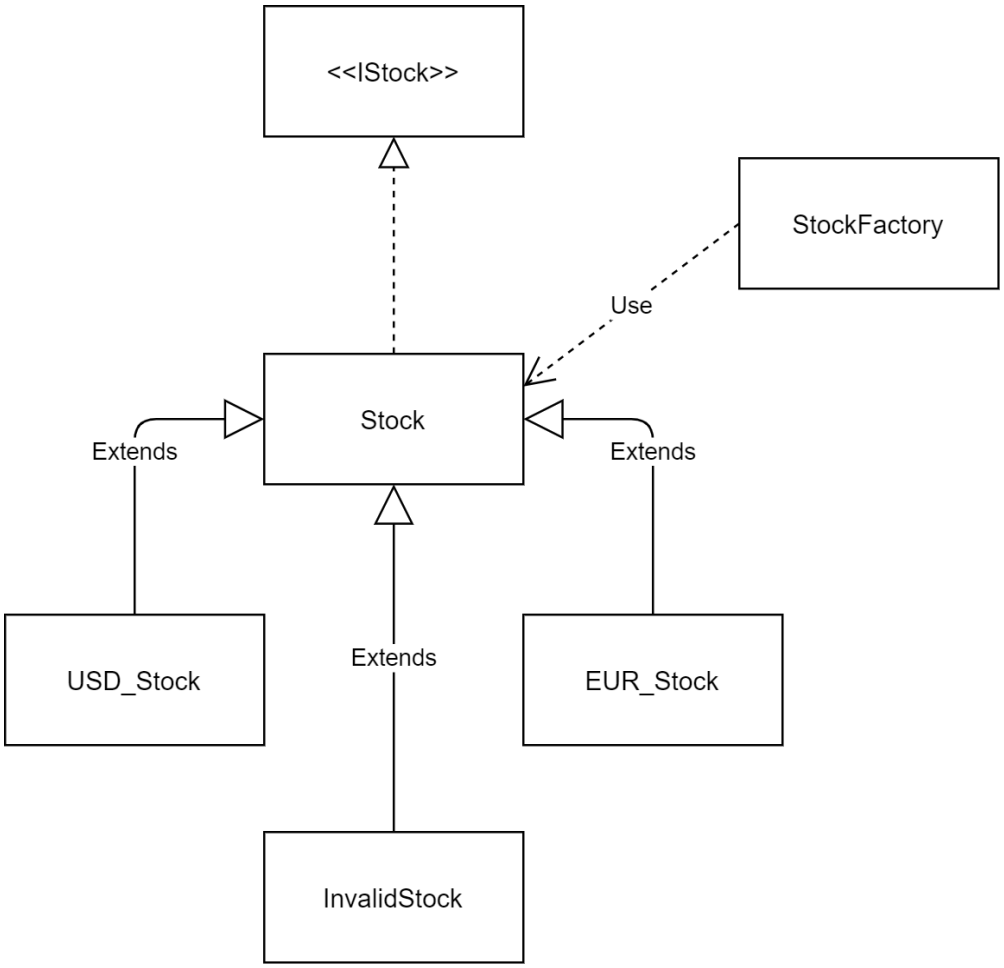
der Selektion wird ein Parameter (1 für Kauf, 2 für Verkauf) and die Factory weitergegeben.

Auf dem gleichen Weg werden Transaktionsobjekte mit den Daten aus der Datenbank erzeugt. Beim Speichern der Transaktionen in die Datenbank, werden je nach Typ unterschiedliche Werte gespeichert, um im Nachhinein zwischen den zwei Arten unterscheiden zu können. Da die Daten der Transaktionen immer gleich sind, ist es nicht sinnvoll z.B. eine Tabelle für Käufe und eine für Verkäufe anzulegen. Die Transaktionen können in eine Tabelle geschrieben werden und unterschieden sich durch den genannten Parameter (1 für Kauf, 2 für Verkauf). Sollen nun die Transaktionen aus der Datenbank geladen werden, werden die Rohdaten in die Factory hineingegeben. Basierend auf dem Parameter (1 oder 2) erstellt die Factory eine Kauf- oder Verkauf Transaktion

Dieses Pattern ist insofern sinnvoll, dass Komponenten die Unterscheidung zwischen Kauf und Verkauf nicht kennen müssen. Soll eine Transaktion in einer Komponente erzeugt werden, so wird die Unterscheidung welche Art von Transaktion erzeugt werden soll in die Fabrik ausgelagert. Bei Bedarf kann somit auch eine weitere Transaktionsart eingefügt werden. Normalerweise müssten man hierzu die Unterscheidung in jeder Komponente anpassen. Mit dem Factory Pattern kümmert sich jedoch die Factory um eine solche Unterscheidung. Die Komponente muss nur das Handling einer allgemeinen *Transaktion* beherrschen, die einzelnen Arten erben von dieser Klasse und haben ihre spezifische Implementierung.

Dasselbe Prinzip ist auch bei Stocks im Zusammenhang mit *USD_Stock*, *EUR_Stock* und *Invalid_Stock* zu erkennen.



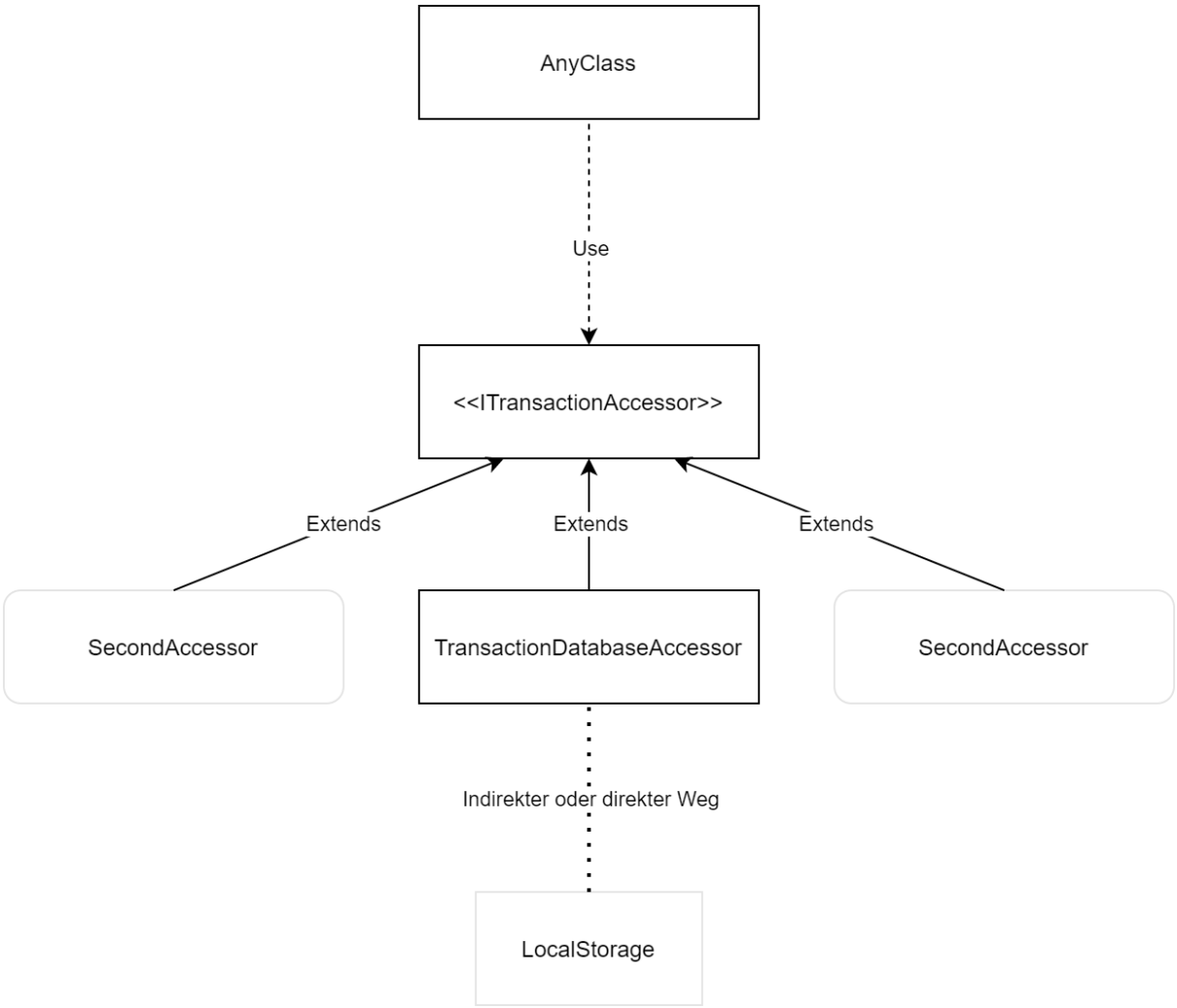


Repository-Pattern

Um die Datenbank von der Logik zu kapseln ist das Repostiory Pattern eine valide Struktur. In diesem Pattern wird für eine bestimme Gruppe von daten ein Repository als Interface definiert. In diesem Beispiel das *`ITransactionAccessor`* Interface. In diesem Interface sind alle nötigen Methoden definiert, welche die Datenbank zurückliefern soll. Die anderen Klassen in diesem Projekt können sich an diesem Interface Orientieren. Dies ist die Definition, wie die Klassen an die Daten gelangen. Wie die Datenbeschaffung letztendlich funktioniert, ist der verwendenden Klasse egal.

Um nun die Datenbeschaffung zu erstellen wird eine Klasse angelegt, welche die dieses Interface implementiert. Hier: *`TransactionDatabaseAccessor`*. Dieser Accessor liest die Daten aus der Datenbank aus und verarbeitet sie weiter, dass sie wie in dem Interface definiert zurückgegeben werden können.

Ein Interface kann von mehreren Klassen implementiert werden. Möchte eine Klasse also das Interface verwenden, können viele unterschiedliche Klassen des Accessors übergeben werden, denn jede Klasse hat die Eigenschaft, das Interface zu implementieren. Wird nun also die Speichermethode ausgetauscht, kann eine neue Klasseangelegt werden, welche das Interface implementiert. Anstelle des alten Accessors kann nun der neue an die verwendende Klasse übergeben werden. Die Schnittstellen sind wie beschrieben die gleichen. Durch dieses Pattern erhält man sehr viel Flexibilität und muss bei Änderungen nicht die verwendenden Klassen anpassen.


AnyClass
Use
<<ITransactionAccessor>>
Extends
Extends
Extends
SecondAccessor
TransactionDatabaseAccessor
SecondAccessor
Indirekter oder direkter Weg
LocalStorage

Architekturmodelle

Einfaches Schichtenmodell

Das einfache Schichtenmodell passt sehr gut zu dieser Anwendung. Hierbei wird zwischen Presentation Layer, Domain Layer und Data Layer unterschieden. Das Presentation Layer ist dazu da, Interaktionen mit dem Benutzer durchzuführen. So werden Daten zur Anzeige aufbereitet, aber auch benutzereingaben aufgefangen und verarbeitet. Die Renderer der Anwendung befinden sich auf dieser Ebene. Ihre Aufgabe ist es eine HTML Struktur zu erzeugen und diese mit den daten der Objekte zu füllen. Teilweise beinhalten die Renderer auch Formulare (z.B. der *`TransactionDialogRenderer`*). Dieser Renderer ermöglicht dem Benutzer das Hinzufügen einer Transaktion in die Datenbank.

Neben dem Presentation Layer gibt es das Domain Layer. Dieses Layer wird benutzt, um Daten zu verarbeiten, Berechnungen durchzuführen oder die Daten aus den Datenbanken zu laden. Die *`History`* Klasse befindet sich auf dieser Ebene. Ihre Aufgabe ist die Verwaltung von mehreren Transaktionen der der gleichen Aktie. Ebenso ist sie in der Lage ihre Daten von der Datenbank [Klasse] anzufragen.

Auf der untersten Ebene befindet sich die Datenbank. Die Datenbank kümmert sich um das Speichern und Laden von Datensätzen aus dem *`localStorage`*. Ebenso auf dieser Ebene ist der *`StockInfoService`*. Ähnlich wie die Datenbank kümmert sich der Service um die Beschaffung von Daten. Konkret frägt dieser Service eine API nach Informationen zu einem spezifischen Parameter.

Das einfache Schichtenmodell ist sehr gut für den Stock Tracker geeignet. Die Kernaufgaben der Anwendung bestehen auf exakt den im Schichtenmodell beschriebenen Aufgaben. Daten werden aus der Datenbank geladen, aufbereitet und angezeigt. Der Aufbereitungslogik ist es egal, wie die Daten angezeigt werden. Somit besteht keine Abhängigkeit zur höheren Schicht. Ebenso hat die Datenbank kein Interesse an der Aufbereitungslogik, sondern kann sich auf die effiziente Speicherung der Daten konzentrieren. Auch sie hängt somit nicht von der höheren Schicht ab.

Client-Server

Um an die neusten Daten der Aktien zu kommen, wird das Client-Server Pattern verwendet. Organisationen und Firmen stellen ihre Services oft online als z.B. API bereit, welche ein Entwickler gegen Bezahlung ansprechen kann. In diesem Projekt wird eine Anfragenlimitierte API eingebunden. Mit dieser API können Informationen wie der Preis der Aktie, aber auch Informationen zu einem Unternehmen abgefragt werden. Der Client, kann diese Daten aufbereiten und wie in diesem Falle in einem leserlichen Format anzeigen.

Ein Client ist nicht auf einen Server limitiert, sondern kann mehrere dieser Art verwenden. Um die Aktienpreise in Euro umzuwandeln wird in diesem Projekt eine weitere API verwendet.

Unit Testing

Vererbung

Wie im Kapitel „Entwurfsmuster – Facotry Pattern“ beschrieben, erzeugt eine Factory die unterschiedlichen Transaktionen. Basierend auf dem Transaktionstyp verändert sich auch die Implementierung der Funktionen. Möchte man zum Beispiel die Auswirkung einer Transaktion auf den Kontostand abrufen, so kann man dies mit `getTransactionPrice()` tun. Bei einem Verkauf bezahlt man nichts, sondern erhält eine Gutschrift. Aus diesem Grund wird eine positive Zahl zurückgegeben. Bei einem Kauf jedoch wird das Konto belastet, weswegen die Methode eine negative Zahl zurückgeben muss.

In der ersten Testklasse `transaction.test.ts` wird nun exakt dieses Verhalten geprüft. Über die Factory wird eine Transaktion angelegt. Anschließend wird geprüft, ob die Methode auch das erwartete Ergebnis einer positiven oder negativen Zahl liefert.

Man kann argumentieren, dass diese Test Suite nicht elementar genug ist. In dieser Suite wird indirekt getestet, ob die Factory die Transaktionen basierend auf dem Parameter erzeugt oder nicht. Ebenso wird getestet, ob die Implementierung der Kind-Klasse korrekt ist. Für diesen Anwendungsfall habe ich mich jedoch dazu entschieden dies so zu handhaben. Im Test wird davon ausgegangen, dass die Implementierung der Kind Klasse funktioniert. Es ist jedoch sinnvoll, eine dedizierte Testklasse zu erstellen, um dies zu testen. (außer es wird als getter/setter betrachtet) -> siehe ATRIP – Professional

Statistiken (mit Mocking)

In der zweiten Testklasse werden die Berechnungen der Transaktionshistorie getestet. Hierbei bedarf es Mocks, um Abhängigkeiten zu anderen Klassen zu kontrollieren. Dieses Vorgehen ist sinnvoll, da bei einem „Unit Test“ nur eine „Unit“ getestet werden soll. Konkret bedeutet dies, dass ein bestimmtes Modul oder eine bestimme Funktion getestet werden soll. Da die meisten Funktionen jedoch in irgendeiner Weise von anderen Klassen oder Modulen abhängen, müssen diese kontrolliert werden. Macht man dies nicht, testet ein Test diese Abhängigkeiten zwangsläufig mit. Der Test beschränkt sich somit nicht mehr nur auf die zu testende Funktion.

Zum Kontrollieren der Abhängigkeiten habe ich mich für die „ts-mockito“ Bibliothek entschieden. Mit dieser Bibliothek können Klassen aus dem Projekt erzeugt und kontrolliert werden. Wird benötigt die zu testende Funktion ein Objekt einer Klasse, kann hier ein Mock Objekt übergeben werden. Dieses Objekt verhält sich wie ein normales Objekt, jedoch können die Methoden und Funktionen dieses gemockten Objekts im Vorhinein festgelegt werden. Die Funktionen des Objekts werden in einem Test nun so angepasst, dass ein gewisser Pfad in der zu testenden Funktion eingeschlagen wird.

In der Testklasse `history.statistics.test.ts` werden die Berechnungen auf eine Transaktionsliste getestet. Um Berechnungen ausführen zu können, bedarf es zunächst einer Liste an Transaktionen.

Im Folgenden wird der Test *getTransactionQuantity - Should return correct amount of stocks owned* betrachtet. In diesem Test wird eine *History* (Transaktionsliste) benötigt. Die Methode *getTransactionQuantity()* soll die korrekte Summe der gekauften Aktien aller Transaktionen zurückliefern. Bei der Erstellung des *History* Objekts werden die Transaktionen aus der Datenbank geladen und in der angesprochenen Factory in die passenden Objekte konvertiert. Das *History* Objekt nimmt bei der Erstellung deinen *DataAccessor* und die zu betrachtende Aktie (*Stock*) an. Basierend auf diesen Parametern lädt die *History* die Transaktionen aus dem *DataAccessor*. Das Laden aus der Datenbank und das Anlegen des Objekts soll aber nicht getestet werden. Aus diesem Grund wird die Abhängigkeit des *DataAccessor* über die Library gemockt. Das bedeutet, dass ein Scheinobjekt des Accessors übergeben wird. Dieses Objekt wird durch die Bibliothek verwaltet. Hiermit können Funktionalitäten kontrolliert und simuliert werden. In diesem Beispiel wird die Methode des Ladens der Transaktionen simuliert. Das gemockte Objekt wird so manipuliert, dass es immer eine vorgegebene Anzahl an Elementen liefert.

Bei der Summierung aller Aktientransaktionen wird über alle Transaktionen gelaufen. Von jeder Transaktion wird die Anzahl an Aktien zurückgeben, um die es sich bei der Transaktion handelt. Auch hier können positive und negative Werte zurückgeliefert werden. Bei einem Kauf wird eine positive Zahl zurückgeliefert, beim Verkauf eine negative. Die Funktion *getTransactionQuantity()* der Transaktion liefert somit wie die Funktion *getTransactionPrice()* die Auswirkung auf die Aktiensumme. Die Aufgabe der zu testenden Funktion beschränkt sich auf die Zusammenrechnung dieser Werte. Es wird davon ausgegangen, dass die Funktion korrekt implementiert ist. Um keine Abhängigkeit zu haben wird diese Funktion auch über die Bibliothek kontrolliert. Im Test wird festgelegt, dass wenn die Funktion *getTransactionQuantity()* auf die von dem *DataAccessor* zurückgegebenen Transaktionen aufgerufen wird, eine bestimmte Zahl zurückgegeben wird. Im betrachteten Test wird bei jedem Aufruf der Methode eine andere zahl zurückgegeben. Da der gemockte Datenlieferant 5 Transaktionen zurückgibt, wird die Methode zum Erhalten der Aktienmenge der Transaktion insgesamt fünf Mal aufgerufen. Einmal je Transaktion. Die gemockten Transaktionen sind nun so manipuliert, dass die erste Transaktion eine Menge von 3 zurückliefert. Die zweite Transaktion hat eine Aktienmenge von 2 usw.. Die zu testende Methode geht zusammengefasst wie folgt vor.

Zunächst wird das zu testende Objekt (History) angelegt. Hier werden gemockte Funktionalitäten übergeben (Objekte). Die angelegte Historie beinhaltet eine Liste an gemockten Transaktionen. Die zu testende Methode betrachtet nun alle einzelnen (gemockten) Transaktion und frägt nach der Auswirkung auf die Aktienmenge. Die gemockten Transaktionen sind so manipuliert, dass diese Methode je nach Transaktion die Werte 3, 2, 2, 1, 2 zurückliefert. Ist die zu testende Methode korrekt implementiert, gibt diese die Summe der genannten Zahlen zurück. In diesem Fall 10.

Mocking ist ein gutes Mittel, um Abhängigkeiten zu kontrollieren. Im Test ist es nicht relevant, woher die Daten kommen. Es wird davon ausgegangen, dass alle Funktionen

der Abhängigkeiten funktionieren. Anstelle ein Komplettes System aufzustellen, damit der Test funktioniert, werden wie beschrieben alle Abhängigkeiten künstlich erstellt. Lediglich die zu testende Funktionalität wird nicht manipuliert. Hierdurch erhält man einen übersichtlichen und elementaren Test.

Die restlichen Tests verhalten sich ähnlich. Zusätzlich zum Manipulieren der Funktion kann auch getestet werden, ob die zu testende Funktion eine Abhängigkeit mit den korrekten Parametern aufruft.

ATRIP

Automatic

Durch das Unit Testing Framework Jest können die Unit Tests schnell und einfach ausgeführt werden. Durch das Konsolenkommando *`jest`* oder *`npm run-script test`* können alle vorhandenen Jest Unit Tests ausgeführt werden. Alternativ können auch einzelne Test Klassen oder einzelne Tests ausgeführt werden. Für mehr Informationen der selektiven Ausführung: https://jestjs.io/docs/. Alternativ können diese selektiven Ausführungen mithilfe der gängigen IDEs per Mausklick ausgeführt werden.

Die Tests prüfen nach dem Wahr oder Falsch Prinzip. Ist beispielsweise die erwartete Anzahl an gekauften Aktien abweichend vom eigentlichen Testergebnis, schlägt der gesamte Test fehl. Bei mehreren Vergleichen in einem Test müssen alle diese Vergleiche wahr sein, damit der gesamte Test bestanden ist.

Thorough

Bei Unit Tests wird wie genannt eine spezifische Funktionalität geprüft. Hierbei sind jedoch öfters teilschritte notwendig. Um einen korrekten Test mit optimaler Fehleridentifikation zu erhalten, müssen diese Teilschritte zusätzlich zum eigentlichen Ergebnis überprüft werden.

Wie bereits beschrieben, werden für Tests oftmals Mocks verwendet, um Abhängigkeiten zu manipulieren. Im Zuge des Tests zur Errechnung der Balance einer Transaktionsliste wird von jeder Transaktion die Balance abgefragt und zusammengerechnet. Neben dem Vergleich, ob das Ergebnis mit der Erwartung übereinstimmt, können weitere Vergleiche getätigt werden. In diesem Test wird zusätzlich geprüft, ob die Methode der Transaktion fünfmal aufgerufen wurde. Ist dies nicht der Fall hat man sofort einen Hinweis, was in der zu testenden Funktion nicht funktioniert.

Die meisten Tests müssen mit unterschiedlichen Parametern ausgeführt werden, um die Funktionalität der zu testenden Funktion bestätigen zu können. So muss im Optimalfall für jede mögliche Eingabe ein Test existieren. Das ist nicht möglich, weswegen mit Wertegruppen und Spezialfällen gearbeitet werden muss. Anstelle aller Werte von minus unendlich bis unendlich zu testen, kann eine Handvoll Werte verwendet werden.

Betrachtet man die Tests für *totalTransactionBalance()* habe ich mich für die Wertegruppen (1) nur positiv, (2) nur negativ, (3) gemischt entschieden. Die Testgruppe verrechnet die Kosten aller Transaktionen. Im ersten Test wird davon ausgegangen, dass es sich nur um Verkäufe und somit nur um positive Werte handelt. Der Zweite Tests geht davon aus, dass in der Liste nur Käufe und somit nur negative Werte vorkommen. Der dritte Test ist realitätsnäher, da hier eine Mischung von positiven und negativen Werten und somit eine Liste von Käufen und Verkäufen betrachtet wird. Somit sind alle relevanten Möglichkeiten abgedeckt.

Repeatable

In der Anwendung ist es nicht nötig, Zufall zu erzeugen. Ebenso wird keine spezifische Datengrundlage benötigt. Kein Test manipuliert die Daten. Die Tests können ohne Probleme mehrfach ausgeführt werden, da alle relevanten Daten neu erzeugt und mittels Dependency Injection eingebunden werden. So wird beispielsweise die Liste an Transaktionen vor jedem Test neu erzeugt und beinhaltet immer dieselben Daten (für einen spezifischen Test)

Independent

Wie bereits beschrieben wird die Liste der notwendigen Transaktionen für einen Test vor jedem Durchlauf erzeugt. Diese Liste existiert ausschließlich für den einmaligen Durchlauf und wird anschließend verworfen. Auch wenn ein anderer Test die exakt gleiche Liste benötigt, wird diese nicht von Test zu Test weiterverwendet, sondern jedes Mal neu erzeugt. Dadurch entsteht eine Isolation der Tests. Jeder Test kann ohne Vorbereitung ausgeführt werden. Er basiert nicht auf einem vorherigen Ergebnis. Somit können Tests auch parallel und in beliebiger Reihenfolge ausgeführt werden.

Professional

In den Tests wird oftmals eine Liste an Transkationen verwendet. Um die Tests übersichtlicher zu gestalten kann die Erzeugung der Transaktionsliste ausgelagert werden. Beispielsweise in der Testklasse *history.split.test* wird die Erzeugung der Transaktionsliste in zwei unterschiedliche Funktionen ausgelagert. In der einen Funktion werden Transaktionen angelegt, welche fünf Tage in der Vergangenheit liegen. Die zweite Funktion legt Transaktionen an, welche in der Zukunft liegen. Über Parameter sind die Menge und die zu generierenden Identifikationsnummern steuerbar.

Diese Funktion ist ausschließlich für diese Testklasse vorgesehen und interagiert nicht mit dem Produktivcode.

Auf unnötige Tests wie *getter* und *setter* wird verzichtet, da diese keine weitere Komplexität beinhalten. Diese liefern lediglich die Felder des Objekts zurück. Man könnte argumentieren, dass die Methode *getTransactionPrice()* der *PurchaseTransaction* getestet werden soll, da diese das Feld nachträglich manipuliert. In diesem Projekt habe ich mich aber dagegen entschieden und mich komplexeren Tests zugewandt. (Weil 10 Tests)

Refactoring

Variable Declaration

In JavaScript und Typescript werden Variablen mit let, var oder const angelegt. Mit var kann man die Variable nachträglich verändern, mit const hingegen nicht. Möchte man also eine Variable vor Manipulation schützen, kann man diese mit const anlegen. Alle Variablen die nur angelegt und nicht manipuliert werden, sollen somit mit const angelegt werden. Manuelles Refactoring nötig. In IntelliJ kann mithilfe des Inspetion Tools angezeigt werden, wo man doch lieber const verwenden soll.

Unused Variable

Wird die Rückgabe einer Funktion in einer Variablen gespeichert wird erwartet, dass diese Variable verwendet wird. Ist dies nicht der Fall, kann Verwirrung entstehen. Aus diesem Grund werden unbenutzte Variablen entfernt, bzw. gar nicht erst zugewiesen. Im *`MainRouter`* der Anwendung wird eine Variable *`router`* angelegt. Der neue Router wird in der Variablen gespeichert. Diese Variable findet danach keine Verwendung. Sie kann somit entfernt werden.

Ebenso können nicht benötigte Parameter aus callback Funktionen entfernt werden. Der *`.addEventListener("click",`* … callback Parameter e (Event) wird nichtverwendet und kann somit entfernt werden.

Rename Method

Die Funktion *`makeid`* wird für die Generierung einer ID verwendet. Der Name der Funktion ist nicht wohlgeformt und sollte somit umbenannt werden. Ein passenderer Name is *`generateRandomIdentificationString`*.

Rename Variable

Der aus einem Codebeispiel kopierte und geänderte Code beinhaltet noch immer die Variable *`drinks`*. Dieser Name ist nicht passend für diese Anwendung und kann somit in *`transactionTypes`* umbenannt werden.

Switch (Factory)

In der *`TransactionFactory`* wurde bisher eine Unterscheidung basierend auf einem Parameter getroffen. Dieser Parameter ist eine Zahl > 0. Die Unterscheidung fand mit einem *`switch`* Statement statt. Anstelle diesen langen Code Blocks, kann der Transaktionstyp in einem Objekt hinterlegt werden. Der übergebene Parameter kann so als Attributname verwendet werden. Die Transaktion wird hierdurch dynamisch erstellt. Der Code ist lesbarer und die Methode um einiges kürzer.

Dasselbe Vorgehen ist auch bei der StockFactory angebracht.

Duplicate Code

Der Code wurde nachträglich so angepasst, dass Teile generalisiert werden können. Dadurch sind sehr viele Zeilen Code nachträglich verloren gegangen, was für dieses Projekt jedoch problematisch ist. Ein Beispiel sind die generellen Generierungen von HTML Grundbausteinen