Dokumentation
für die Anwendung »Stocker«

1584 Programmierpraktikum, SoSe 2021

Matthias Rudolph

4. Juli 2021

Inhaltsverzeichnis

1 Einleitung

Im Rahmen des Programmierpraktikums habe ich ein Programm zur Verwaltung und Darstellung von Aktien und ihren Aktienkursen geschrieben. In der Aufgabenstellung wurden umfangreiche Anforderungen detailliert, die den Aufbau der GUI und der Geschäftslogik betrafen. Das »Stocker«-Programm ruft z. B. Daten von einem Netzwerk-Datenlieferanten ab, verwaltet die Aktien in einer Watchlist und stellt Aktienkurse in mehreren Fenster in einer *Multiple-document*-Oberfläche dar.

2 Beschreibung der Programmstruktur

Bei der Programmierung meines Stocker-Programm habe ich mich an einer für meine Zwecke angepassten Variante der *Model-View-Controller*-Architektur orientiert. Die Klassen-Dateien sind in entsprechende Pakete eingeteilt: `model` (& `model.dataWrappers`), `view`, `controller`. Diese Pakete werden ergänzt um Pakete, die eine bestimmte, klar umgrenzte Funktionalität zur Verfügung stellen: `network`, `preferences`, `persistence` und `json`, und das Paket `common` für alle gemeinsamen Schnittstellen und Enums.

Die verschiedenen Programmbestandteile werden von einem `MainController` zusammengehalten und alle Kindfenster der grafischen Oberfläche entsprechend der *Multiple-document interface*-Architekturvorgabe auf einem `MainFrame` dargestellt. Im Hintergrund ist ein zentrales Datenmodell für die Verwaltung der Daten zuständig.

2.1 *Model-View-Controller*-Architektur

Ein Problem dabei, die *Model-View-Controller*-Architektur auf ein grafisches Programm zu übertragen, das Javas Swing-Framework verwendet, besteht darin, dass Swing selbst auf einem vereinfachten *MVC*-Modell basiert, das nicht zwischen *View* und *Controller* unterscheidet. Das *Java Swing*-Buch von O'Reilly schreibt dazu: »Swing actually makes use of a simplified variant of the MVC design called the *model-delegate*. This design combines the view and the controller object into a single element that draws the component to the screen and handles GUI events known as the *UI delegate*.«[1]

Konkret bedeutet das, dass in Swing ein *View* selbst die GUI-Events verarbeiten kann, die auf ihm erzeugt werden. An einigen Stellen in meinem Programm, wo sich ein *View* eines Programmteils mehr oder weniger als alleinstehende Einheit implementieren ließ, habe ich deshalb darauf verzichtet, noch einen weiteren *Controller* zu erstellen, nur um die wenigen GUI-Events vom *View* dorthin durchzureichen. Das hätte an diesen Stellen nur weitere Abhängigkeiten und Methoden-Redundanzen erzeugt, ohne dass ein Gewinn an Übersichtlichkeit oder Modularität ersichtlich gewesen wäre. Zu nennen sind hier die *Views* der Funktionseinheiten Suche, Watchlist und Chart, die hauptsächlich als unterschiedliche Darstellungen der Daten des *Models* fungieren.

[1] `https://www.oreilly.com/library/view/java-swing/156592455X/ch01s04.html`, abgerufen am 30.06.2021.

In den Programmteilen, in denen eine deutlichere Trennung zwischen Programmlogik und Darstellung besteht, habe ich die Trennung in *View* und *Controller* beibehalten, z. B. bei den Alarmen oder den Nutzer:innen-Einstellungen.

2.2 Weitere Architektur-Muster

Um GUI-Events zu verarbeiten, die über die jeweilige Funktionseinheit hinausweisen, habe ich die *MVC*-Architektur um das Mediator-Muster ergänzt. Der bereits angesprochene `MainController` fungiert als Mediator und ermöglicht die Kommunikation zwischen verschiedenen Programmbestandteilen, die sich untereinander nicht kennen.[2]

Konkret für mein Stocker-Programm bedeutet das, dass der Mediator ins Spiel kommt, wenn etwa aus der Suchfunktion heraus ein neues Chart-Fenster geöffnet oder alle laufenden Programmbestandteile über eine Veränderungen der Nutzer:innen-Einstellungen informiert werden sollen. Um die Abhängigkeiten möglichst gering zu halten, sind alle diese Mediator-Methoden in einem `IMainController`-Interface zusammengefasst.

Zusätzlich dazu verwende ich an diversen Stellen meines Programms das für die *MVC*-Architektur übliche Beobachter-Muster. So verfügt etwa der Netzwerk-Controller über einen `PushSubscriber`, der die Daten aus neuen Push-Updates erhält, das `StockerModel` verfügt über einen `AlarmListener`, der informiert wird, wenn dem Modell neue Alarme hinzugefügt werden, und jedes `StockItem` verfügt über eine Liste von `StockListenern`, die jedes Mal informiert werden, wenn die Aktiendaten durch ein Push-Update aktualisiert wurden.

2.3 Datenhaltung

Die zentrale Datenhaltung meines Programms ist zweistufig organisiert. Zum einen gibt es das Datenmodell in der Klasse `StockerModel`, das als zentrale Datenschnittstelle aller Funktionseinheiten des Programms fungiert und entsprechende Operationen zur Manipulation der Daten zur Verfügung stellt. Insbesondere verläuft über dieses zentrale Model der Datenabruf von den Datenprovidern über die Schnittstelle des Netzwerk-Controllers.

Auf der zweiten, niedrigeren, Stufe sind die Daten des Programms in Objekten der Klasse `StockItem` aufbewahrt – ein Objekt für jede betrachtete Aktie. Diese `StockItems` speichern die allgemeinen Angaben zu jeder Aktie (Name, Beschreibung usw.), sowie die Kurswerte. Außerdem verfügen sie über zwei verschiedene Status: `available` und `loading`. `Available` wird gesetzt, wenn der Datenprovider überhaupt Daten zur entsprechenden Aktie liefern kann. Relevant wird das etwa bei einem Wechsel des Datenproviders, so dass Aktien eines anderen Datenproviders danach als `not available` geführt werden können, ohne jedoch Fehler zu verursachen oder aus dem Programm entfernt werden zu müssen. So können die Aktien bei einem Wechsel zum ursprünglichen Provider

[2] Vgl. `https://en.wikipedia.org/wiki/Mediator_pattern`, abgerufen am 30.06.2021.

wiederhergestellt werden. Der zweite Status, `loading`, zeigt an, ob bereits eine Datenanfrage beim Provider läuft. So soll verhindert werden, dass zu viele HTTP-Anfragen hintereinander herausgeschickt werden, was durch die stellenweise langsamen Reaktionszeiten bei der Abfrage historischer Daten über die HTTP-API sonst verursacht werden und zu HTTP-Fehlern führen könnte.

Neben der Darstellung historischer Kursdaten, die über die HTTP-API des Datenlieferanten abgerufen werden, kann das Stocker-Programm auch die Echtzeit-Daten verarbeiten, die vom Datenlieferanten über eine Web-Socket-Verbindung zur Verfügung gestellt werden. Die Aufgabenstellung bot einen Algorithmus, wie aus einer Liste von Times-Sales-Daten eine Menge neuer Chart-Kerzen berechnet werden können. Ich habe diesen Algorithmus für mein Programm dahingehend angepasst, dass nach jedem Aufruf des Algorithmus die historischen Kerzendaten mit den Kerzendaten aus den Echtzeit-Aktualisierungen verschmolzen werden. Das hat den Vorteil, dass das Programm die Liste von Times-Sales-Daten nicht vorhalten muss, sondern direkt nach ihrer Verarbeitung verwerfen kann. Der Algorithmus wurde so geändert, dass er zunächst immer auf der letzten Kerze der im Modell vorhandenen Kerzen-Liste operiert und über die Zeit-Differenz des gewählten Chart-Intervalls ermittelt, ob diese letzte Kerze mit neuen Daten aktualisiert werden muss oder ob ihr Intervall abgeschlossen ist, in welchem Fall dann eine neue Kerze mit neuen Daten angehängt wird.

2.4 Statische Methoden

Trotz ihrer generellen Gegenläufigkeit zu einer objektorientierten Programmstruktur habe ich mich entschieden, an zwei Stellen des Programms statische Methoden beizubehalten. Zum einen bei der JSON-(De-)Serialisierung, die ich im Rahmen einer `JsonFactory`-Klasse implementiert habe, die mir auf Methodenaufruf hin in ihrer »Factory« Strings im JSON-Format bzw. Java-Objekte »produziert«.

Zum anderen kommen statische Methoden beim Berechnen von Indikatoren und dem Aktualisieren der Aktien-Daten nach einem Push-Update zum Einsatz. Semantisch naheliegend wäre es im Rahmen meines Programms gewesen, konkrete Objekte einer solchen Berechnungs-Klasse an die `StockItem`-Objekte anzubinden, die die Daten selbst enthalten, so dass jede Aktie über ihre eigenen Rechen-Fähigkeiten verfügt.

Die Methoden zur Berechnung der Indikatoren sollten jedoch dem von der Kursleitung vorgegebenen Test-Interface offengelegt werden und von dort mit Arrays beliebiger Kursdaten bestückt werden. Deshalb hätte sich ein Vorgehen als unpraktikabel erwiesen, bei dem jede Instanz der Klasse `StockItem` für ihre eigenen Aktienwerte bestimmte Indikatoren berechnet, ohne dass die Werte noch einmal explizit übergeben werden, wie im Test-Interface gefordert. Deshalb habe ich die benötigten Berechnungs-Methoden schließlich als statische Methoden einer `StockCalcHelper`-Klasse implementiert, die aufgerufen werden können, ohne zuerst eine Instanz der Berechnungsklasse zu erzeugen, die keine weitere Funktionalität bieten würde.

2.5 Weitere Bestandteile

Zum Datenabruf verfügt das Stocker-Programm wie schon erwähnt über einen zweiteiligen Netzwerk-Controller, der zum einen die HTTP-Requests händelt und zum anderen eine WebSocket-Verbindung zum Datenprovider öffnet, um die Echtzeit-Updates zu empfangen. Die Nachrichten des Datenproviders kommen als Strings im JSON-Format und werden vom Netzwerk-Controller an das Datenmodell weitergeleitet. Dort werden sie verarbeitet und in Java-Objekte übersetzt, die von den Methoden meines Programms verarbeitet werden können.

Um die Daten verarbeiten und verwalten zu können, habe ich eine Reihe von Wrapper-Klassen geschrieben, die die Daten zu den verschiedenen Datenpunkten gliedern und die im Paket `model.dataWrappers` zusammengefasst sind. Dort gibt es etwa Klassen zu den Alarmen (mit ID der Aktie, zu der er gehört, Threshold-Wert usw.), den Kerzen (mit Höchst-, Tiefst- und allen weiteren Werten) oder den möglichen Indikatoren (Bollinger Band, Gleitender Durchschnitt). Im Vergleich zur eins-zu-eins-Übertragung der Formate der Datenlieferanten ermöglichen die Wrapper-Klassen eine deutlich übersichtlichere Verarbeitung.

Die Nutzer:innen haben bei der Bedienung die Möglichkeit, eine Reihe von Einstellungen ihren Wünschen entsprechend anzupassen. Diese Einstellungen müssen sowohl beim Start des Programms eingelesen (oder aus den Defaults wiederherstellt) werden als auch vom `PreferencesController` an alle notwendigen Programmteile weiterkommuniziert werden, wenn eine Veränderung zur Laufzeit auftritt. Die grundsätzlichste Einstellungsveränderung betrifft dabei den Wechsel des Datenproviders, der einen Neuaufbau des Programms mit einem neuerlichen Abrufen aller Daten (sofern möglich) von diesem neuen Provider notwendig macht.

Das Programm verfügt außerdem über die Möglichkeit, seinen aktuellen Zustand zu speichern und nach einem Neustart wiederherzustellen. Die Persistenz habe ich in drei Teile aufgeteilt, die zusammen in einer Datei gespeichert, aber einzeln wiederhergestellt werden. Erstens die Einstellungen, zweitens das Datenmodell und drittens der Zustand der grafischen Oberfläche.

3 Beschreibung der Implementierung der Indikatoren und der grafischen Darstellung

Teil der Aufgabenstellung war neben der Verwaltung der Aktien ihre grafische Darstellung und die grafische Darstellung bestimmter Indikatoren der technischen Aktienanalyse, konkret von Bollinger Bändern und Gleitenden Durchschnitten.

3.1 Objekt-eigene `draw`-Methoden

Im Sinne einer objektorientierten Programmstruktur bestand mein erster Ansatz darin, jedes zu zeichnende Element als eine Klasse zu implementieren, die über eine eigene Zeichenmethode verfügt (vgl. SVN revision 2808). Dieser Ansatz ist schließlich an Threading-Problemen gescheitert, die ich im Rahmen der gesetzten Zeit nicht im Stande zu lösen war. Zu Dokumentationszwecken soll dieser Ansatz hier dennoch kurz erläutert werden, bevor ich vorstelle, wie ich die Implementierung schließlich stattdessen umgesetzt habe.

Für den ersten Ansatz habe ich ein Interface `IPaintableChartComponent` erstellt, das über eine abstrakte Methode `drawOn(ChartPanel)` verfügte und von allen zu zeichnenden Elementen (Bundles aus Kerzen oder Linien-Punkten, den Indikatoren oder Alarmen) implementiert wurde. So konnte die Chart-Funktionseinheit die notwendigen zeichenbaren Elemente verwalten, denen jeweils nur im Methodenaufruf das Panel übergeben werden musste, auf das sie sich selbst zeichnen sollten. Die Implementierung des Zeichenvorgangs wäre dementsprechend in die einzelnen Datenstrukturen verlegt und dort auf die jeweiligen Erfordernisse zugeschnitten worden.

Das Problem, das sich mir bei diesem Vorgehen gestellt hat, betraf das Threading-Verhalten von Swing. Wenn die `drawOn`-Aufrufe nicht über `invokeLater` erfolgt sind, dann führte das nur zu einem Flackern der gezeichneten Elemente und zu weiteren Darstellungsfehlern. Wenn jedoch mithilfe von `invokeLater` sichergestellt wurde, dass die Zeichenaufrufe korrekt auf dem *Event Dispatch Thread* erfolgten, dann führte das zu einer starken Verlangsamung des Programmablaufs. Die Kerzen wurden sichtbar eine nach der anderen gezeichnet, bei den Indikatoren war teilweise das Zeichnen jedes Datenpunkts einzeln wahrzunehmen.

Da das `ChartPanel` samt all seiner Elemente durch häufige Push-Übertragung neuer Aktienwerte sehr regelmäßig, teilweise sogar mehrmals in der Sekunde, neugezeichnet werden muss, führte die Verlangsamung zu einem konstanten Flackern des gesamten Bildes. Im Ergebnis war das ein nicht akzeptabler Zustand, weshalb ich von diesem Ansatz abgerückt bin.

3.2 `draw`-Methoden im Chart Panel

Meine alternative Herangehensweise bestand darin, eine `ChartPanel`-Klasse zu erstellen, die für jedes zu zeichnende Chart-Element eine angepasste Zeichen-Methode besitzt (`drawCandles`, `drawBollingerBands` usw.). Dieses Vorgehen war zwar erfolgreich, ist in Hinsicht auf die Erweiterbarkeit um neue Indikatoren oder andere Chart-Inhalte aber natürlich nachteilig gegenüber dem zuerst versuchten Ansatz. Vom vorherigen Ansatz beibehalten habe ich dabei, die Daten jedes zu zeichnenden Elements in Wrapper-Klassen zu verpacken (`BollingerBand`, `AlarmUnit` usw.), die sich die abstrakte Superklasse `PaintableChartComponent` zur Bereitstellung gemeinsamer Methoden und Attribute teilen. Die Wrapper-Klassen stellen über Getter-Methoden alle zum Zeichnen notwendigen Werte zur Verfügung und das Chart Panel verarbeitet die Werte in ihren Methoden.

Das Panel sammelt zunächst alle zu zeichnenden Objekte (Kerzen, im Frame hinzugeschaltete Indikatoren oder der Aktie hinzugefügte Alarme). In einem weiteren Schritt fixiert es die Perspektive, d. h. den Ausschnitt des Wertebereichs, in dem alle zu zeichnenden Elemente liegen. (Ein absichtlich viel zu hoch oder viel zu tief gesetzter Alarm führt dementsprechend zu einer Nivellierung aller anderen Elemente. Die Alternative wäre aber, dass solche Alarme im Panel schlicht unsichtbar blieben.) Danach ruft das Panel nacheinander die jeweiligen `draw`-Methoden auf. Auf die Reihenfolge kommt es dabei insofern an, dass für die Darstellung relevant ist, welche Elemente über andere Elemente gezeichnet werden (Kerzen etwa über Bollinger Bänder und nicht etwa andersherum, weil sonst alles verdeckt wäre).

3.3 Daten-Pipelines

Voraussetzung dafür, dass etwas gemalt werden kann, ist natürlich, dass die Daten zu einer Aktie überhaupt vorliegen, d. h. entweder abgerufen oder generiert wurden. Für das Stocker-Programm ist grundsätzlich eine Kombination notwendig. Die historischen Daten werden beim Datenprovider abgerufen und von meinem Programm mit Echtzeit-Daten kombiniert, die ebenfalls vom Datenprovider kommen. Insbesondere ist es notwendig, alle Aktien-Indikatoren zu aktualisieren, wenn sich die Aktien-Werte verändert haben.

Wichtig ist, dass das Programm an verschiedenen Stellen Checks durchführt, ob die benötigten Daten vorliegen und sie sonst abfragt. Dazu legt das `ChartFrame` etwa eine maximale Anzahl von Kerzen fest, die gezeichnet werden sollen, muss aber auch mit weniger Kerzen umgehen können, wegen der Eigenheiten der Datenprovider für zumindest das Monats-Intervall weniger Kerzen zu liefern als für die anderen Intervalle.

Gleichzeitig muss das Programm bedenken, dass es für die Zeichnung der Chart-Indikatoren mehr *data points* benötigt, als tatsächlich gemalt werden, da für die Berechnung akurater Gleitender Durchschnitte in die Vergangenheit zurückgegriffen werden muss. Insofern muss das Programm nicht nur testen, ob es überhaupt genug Daten zum Malen hat, sondern auch, ob es darüberhinaus genug Daten zum Berechnen der Indikatoren besitzt. Die historischen Daten müssen über HTTP-Requests beim Daten-Provider abgefragt werden, wofür das Datenmodell zuständig ist.

Wie beschrieben, wird das eigentliche Zeichnen in meinem Programm von der `ChartPanel`-Klasse übernommen. Eine Instanz dieser Klasse wird von einem `ChartFrame` erzeugt, der im Sinne der oben diskutierten Überlegungen zur *MVC*-Architektur als *UI delegate* fungiert, d. h. *View*- und *Controller*-Funktionen in einem Objekt kombiniert. Das `ChartPanel` bezieht die Aktien-Daten aus dem *Model* und die Einstellungen (Intervall, Charttyp) von seinem `ChartFrame`. Änderungen dieser Einstellungen durch Nutzer:innen sorgen dafür, dass das `ChartFrame` in seiner *Controller*-Funktion mit dem Model kommuniziert und dort neue Daten anfragt, insbesondere wenn das Intervall verändert wurde und dafür noch keine Daten vorliegen.