Stocker - Dokumentation

Grundpraktikum Programmierung (1584), Sommersemester 2021

Marc S. Schneider, 3254631

26. Juni 2021

1 Einleitung

Stocker ist eine Anwendung zur Beobachtung von Aktienkursen und weiteren Börsenwerten, die im Rahmen des Grundpraktikum Programmierung der FernUniversität in Hagen im Sommmersemester 2021, veranstaltet vom Lehrgebiet Programmiersysteme, entstanden ist. Die Daten werden bei Bedarf von einem Datenanbieter über ein Netzwerk oder das Internet beschafft. Standardmäßig sind dies `finnhub.io` und der `Kursdatengenerator`; weitere Anbieter können von Nutzer:innen hinzugefügt werden, insofern diese dieselbe API zur Kommunikation anbieten. Zur Beobachtung stehen alle Werte zur Verfügung, die der Datenanbieter in den Rubriken `quote` und `stock` unterstützt. Im Falle von `finnhub.io` sind dies Aktien, Währungen und Kryptowährungen. Die Beobachtung erfolgt in einer Watchlist (Tabelle) oder in Charts, wobei in den Charts zusätzlich technische Indikatoren eingezeichnet und Alarmwerte, bei deren Über- oder Unterschreitung eine Benachrichtigung ausgelöst wird, definiert werden können.

Dieses Dokument gibt eine kurze Einführung in die Bedienung der Anwendung (Abschnitt 2), soweit diese nicht aus der Aufgabenstellung hervorgeht oder selbsterklärend ist, gefolgt von einem kurzen Überblick über die Struktur des Programms (Abschnitt 3) und einer Beschreibung der Implementierung zentraler Programmfunktionen, nämlich der Berechnung der technischen Indikatoren und der Darstellung der Daten in Charts (Abschnitt 4).

2 Funktionsumfang und Bedienung

Die Anwendung bietet alle Funktionen und erfüllt alle Anforderungen, die aus der Anforderungsliste und den Szenarien in Abschnitt 4 der Aufgabenstellung hervorgehen. Hier werden nur die Funktionen beschrieben, die nicht in den Anforderungen der Aufgabenstellung erwähnt sind und / oder deren Bedienung eventuell nicht selbsterklärend ist.

2.1 Sitzungsverwaltung

Stocker ermöglicht die Darstellung einer großen Zahl von Daten, wodurch leicht die Übersichtlichkeit verloren gehen kann. Zudem ist davon auszugehen, dass ein:e Nutzer:in sich zu verschiedenen Zeiten für verschiedene Börsenwerte interessiert. Zum Beispiel könnten werktags sowohl Aktien also auch Kryptowährungen von Interesse sein, aber am Wochenende, wenn die Aktienmärkte geschlossen sind, ausschließlich Kryptowährungen. Außerdem könnte es für Nutzer:innen sinnvoll sein, verschiedene Rubriken wie zum Beispiel Tech-Werte, Rohstoff-Werte oder Immobilienaktien zu unterscheiden und jeweils nur eine davon gleichzeitig zu betrachten. Um dies zu ermöglichen, unterstützt Stocker das Erstellen und Verwalten von Sitzungen (Sessions). Eine Sitzung umfasst Informationen über die Watchlist-Einträge und geöffnete Charts mit allen Einstellungen und Indikatoren, sowie die Positionen und Größen aller geöffneten Kind-Fenster. Beim ersten Programmstart wird eine Standard-Sitzung mit dem Namen *default* geöffnet. Beim Beenden des Programms wird automatisch die aktuelle Sitzung gespeichert, und beim nächsten Start der Anwendung wird die zuletzt aktive Sitzung wieder geladen. Ansonsten können die Sitzungen folgendermaßen verwaltet werden:

- `Sitzung` → `Unter neuem Namen speichern`: Die Sitzung in ihrem aktuellen Zustand wird unter einem neuen Namen gespeichert.
- `Sitzung` → `Aktuelle Sitzung speichern` (Shortcut: `Strg + S`) speichert die aktuelle Sitzung.
- Wechsel der aktiven Sitzung durch Klick auf den Namen der Sitzung im Menu `Sitzung`. Nach dem Bestätigen des Wechsels wird gefragt, ob die aktuell aktive Sitzung (die durch den Wechsel verlassen wird) gespeichert werden soll.
- `Sitzung` → `Aktive Sitzung löschen` löscht die aktuell aktive Sitzung (d.h. alle Watchlist-Einträge werden entfernt und alle Charts geschlossen) und eine der anderen Sitzungen wird geladen
- `Sitzung` → `Sitzung zurücksetzen` setzt die aktive Sitzung zurück. Dadurch werden alle Watchlist-Einträge entfernt und alle Charts geschlossen. Die Sitzung selbst bleibt aber unter ihrem Namen bestehen und aktiv.

Die Sitzungen werden programmintern gespeichert und beim Beenden des Programms im JSON-Format in die Datei `stocker_3254631_session.json` im Programmverzeichnis geschrieben. Sie werden nicht mit in die Properties-Datei `stocker_3254631.json` geschrieben, um die Programmeinstellungen und die Sitzungen voneinander zu trennen (so kann zum Beispiel eine Sitzungs-Datei an eine andere Instanz von Stocker auf einem anderen Rechner übertragen werden, ohne die Einstellungen der anderen Instanz zu überschreiben).

2.2 Werte von Nicht-US-Börsen ausblenden

Mit dem kostenlosen Zugang bei `finnhub.io` ist nur der Zugriff auf Werte, die an den US-Börsen *NYSE* und *NASDAQ* gehandelt werden, erlaubt. Die Suchfunktion liefert jedoch Werte von diversen Börsen weltweit, die durch einen Punkt, gefolgt vom Kürzel der jeweiligen Börse im Tickersymbol gekennzeichnet sind (z.B. `.DE` für die deutsche *XETRA*). Meistens werden viele dieser Nicht-US-Tickersymbole gefunden, und die maximale Anzahl an Suchergebnissen, die `finnhub.io` liefert, ist begrenzt. Dadurch kann es vorkommen, dass sinnvolle Ergebnisse nicht auftauchen, stattdessen aber viele Ergebnisse von Nicht-US-Börsen, die den Nutzer:innen eines kostenlosen Accounts gar nicht zugänglich sind. Um die Übersichtlichkeit zu verbessern, besteht die Möglichkeit, durch Setzen eines Hakens in den Einstellungen → Tab *Datenanbieter* alle Werte mit Punkt (.) im Tickersymbol auszublenden. In der Regel sind dies die Symbole, die mit dem kostenlosen Account nicht zugänglich sind. Leider gibt es einige wenige Tickersymbole, die regulär, also auch an der *NYSE* bzw. *NASDAQ*, einen Punkt im Namen haben, und die nicht mehr gefunden werden, solange diese Option aktiviert ist. Falls Werte in der Suche zu fehlen scheinen, sollte diese Funktion zur Sicherheit abgeschaltet werden. Standardmäßig ist sie abgeschaltet.

2.3 Gitterlinien

In der Menüleiste jedes Chartfenster gibt es eine Checkbox, mit der gestrichelte Gitterlinien (entsprechend der automatisch gewählten Ticks an den Koordinatenachsen) in den Chart eingezeichnet werden können, um die Ablesbarkeit der Daten zu verbessern. Die Gitterlinien sind standardmäßig aktiviert. Ihr Zustand wird mit der Sitzung gespeichert.

2.4 Tastatur-Shortcuts

`Strg + F`	öffnet den Such-Dialog
`Strg + W`	zeigt die Watchlist an (bringt sie nötigenfalls in den Vordergrund)
`Strg + E`	öffnet den Einstellungs-Dialog
`Strg + S`	speichert die aktuelle Sitzung
`Strg + I`	im Chart-Fenster: öffnet die Indikator-Verwaltung
`Strg + A`	im Chart-Fenster: öffnet die Alarm-Verwaltung

3 Beschreibung der Programmstruktur

Die Anwendung folgt dem Entwurfsmuster *Model-View-Controller*. Danach richtet sich auch die Aufteilung in Pakete. Die Aufgaben und der Inhalt der einzelnen Pakete werden in den folgenden Unterabschnitten beschrieben.

3.1 Das Modell: `stocker.model`

Das Paket `stocker.model` enthält die Datenstrukturen, die zur lokalen Zwischenspeicherung der Daten vom Datenanbieter dienen. Eine echte lokale Datenhaltung existiert nicht: Alle Daten werden bei Bedarf vom Datenanbieter geholt, vorübergehend lokal gespeichert, solange sie gebraucht werden, und anschließend verworfen. Die abstrakte Klasse `WatchItem` definiert elementare Eigenschaften von Datenelementen wie einen Schlüssel (Tickersymbol) und einen Namen. Von ihr abgeleitet ist die Klasse `WatchlistItem`, die geeignet ist, um einen Börsenwert in der Watchlist darzustellen, und die Klasse `ChartWatchItem`, die zur Darstellung eines Börsenwerts in einem Chart gedacht ist. Außerdem gibt es die abstrakte Klasse `ChartIndicator` zur Beschreibung beliebiger Chartindikatoren und ihre Implementierungen `ChartIndicatorSMA` (für den gleitenden Mittelwert, *Simple Moving Average*) und `ChartIndicatorBollingerBands` für die Bollinger-Bänder. Um Alarme abzubilden, gibt es die sehr einfach gehaltene Klasse `ChartAlarm`. Der `AlarmManager` dient als zentrale Stelle zur Verwaltung der Alarme über alle Charts und die Watchlist hinweg. Die Alarme sind jeweils einem Tickersymbol zugeordnet (und nicht einem einzelnen Chart). Eng verbunden mit den Modell-Klassen ist die Klasse `Candle` aus dem Paket `stocker.util`, die einen einzelnen Datenpunkt mit den dazugehörenden Werten beschreibt und von allen Modell-Klassen benutzt wird.

3.2 Die Ansicht: `stocker.view`

Die Rolle der Ansicht wird übernommen vom `StockerFrame` (beschreibt das Hauptfenster) sowie von der `StockerWatchlist` (beschreibt das Kindfenster, das die Watchlist enthält) und `StockerChart` (ein Kindfenster, das einen Chart darstellt). Sowohl die Watchlist-Klasse als auch die Chart-Klasse implementieren das Interface `IStockerDataListener`, das eine Methode zur Entgegennahme einer Push-Benachrichtigungen über einen neuen Preis vorschreibt. Innerhalb der `StockerWatchlist` werden einige spezialisierte Versionen von Standard-Swing-Klassen verwendet, um die Tabelle zur Darstellung der Werte zu definieren (`WatchlistTable`, `WatchlistTableModel`, `WatchlistTableCellRenderer`). Innerhalb des `StockerChart` dient das `ChartPanel` als Zeichenfläche und übernimmt die Datendarstellung im engeren Sinne. Die Klassen der Ansicht benutzen die Datenformate, die in `stocker.model` definiert sind, um die Daten zu empfangen und zu verarbeiten. Geliefert werden die Daten vom `StockerDataManager` aus `stocker.control` (siehe nächster Unterabschnitt).

3.3 Die Steuerung: `stocker.control`

Die Rolle der Steuerung (des Controllers) wird primär von einem Objekt des Typs `StockerControl` übernommen. Dieses übernimmt unter anderem auch die Sitzungsverwaltung und die Verwaltung der Programmeinstellungen. Sämtliche Aufgaben der Datenbeschaffung vom Datenanbieter wurden aber der Übersichtlichkeit halber in

die Klasse `StockerDataManager` ausgelagert, die funktional gewissermaßen zwischen den Rollen der Steuerung und des Modells steht, aber von der Arbeitsweise her eher zur Steuerung gehört. Der `StockerDataManager` benutzt einen Websocket-Client (`WSPushClient`), um sich beim Datenanbieter für Push-Updates anzumelden, und implementiert das Interface `IStockerDataReceiver`, was ihn befähigt, diese Push-Updates in Empfang zu nehmen und an die Datenempfänger (Watchlist, Charts) weiterzuleiten.

3.4 Die Dialogfenster: `stocker.dialog`

Die Anwendung kennt verschiedene Arten von Dialogfenstern. Diese erfüllen jeweils eine recht einfache und beschränkte Aufgabe. Um die Programmstruktur nicht unnötig aufzublähen, wurden die Dialoge jeweils in einer einzigen Klasse realisiert, die sowohl die Rollen der Ansicht als auch die der Steuerung übernimmt. Die Aufgabe des Modells (i.e. das Bereitstellen von Daten, die darzustellen sind, bzw. die Entgegennahme von Änderungen an diesen Daten) wird von anderen Programmteilen übernommen. Die Anwendung kennt hauptsächlich vier Arten von Dialogfenstern:

- Der Such-Dialog, Typ `StockerSearchDialog`, ermöglicht die Suche nach Börsenwerten und stellt das Ergebnis der Suche dar. Er implementiert das Interface `ISearchDataReceiver`, was ihn zum Empfang von Suchergebnissen qualifiziert
- Der Einstellungs-Dialog, Typ `StockerPropertyDialog`, stellt die Programmeinstellungen dar und ermöglicht Nutzer:innen, Änderungen vorzunehmen
- Der Indikator-Dialog, Typ `StockerIndicatorDialog`, ermöglicht es, in einem Chart technische Indikatoren hinzuzufügen und zu verwalten
- Der Alarm-Dialog, Typ `StockerAlarmDialog`, ermöglicht es, in einem Chart Alarmwerte hinzuzufügen und zu verwalten.

Daneben werden für sehr einfache Aufgaben direkt die Dialoge von `JOptionPane` (aus Swing) verwendet.

3.5 Die sonstigen Hilfsmittel: `stocker.util`

Dieses Paket umfasst diverse Hilfsmittel zur Beschreibung von Daten oder Einstellungen, die sich allerdings überwiegend passiv verhalten und selbst keine Aufgabe im Programmablauf übernehmen. Dazu gehören:

- Die Klasse `Candle`, die einen einzelnen Datenpunkt aus einem Candlestick-Chart beschreibt (sie dient nur der Zusammenfassung zusammengehöriger Werte im Sinne eines einfachen Records und bietet ansonsten keinerlei Funktionalität)
- Die (nicht-instanziierbare) Hilfsklasse `CandleParser`, die (statische) Methoden zum Erzeugen von `Candle`s aus JSON-Objekten bereitstellt

- Diverse Aufzählungstypen (Enums) zur Beschreibung von Eigenschaften wie z.B. Chart-Typen, Chart-Intervallen oder Chart-Farben (erkennbar daran, dass der Klassenname mit `E` beginnt)
- Weitere Hilfsklassen, zum Beispiel zur Validierung und automatischen Korrektur der Eingabe in Textfeldern (`TextfieldIntValidatorOnFocusLost`) oder um Ausnahmen (Exceptions) im Zusammenhang mit der Datenbeschaffung zu beschreiben (`StockerDataManagerException`)

4 Beschreibung der Implementierung ausgewählter Aspekte

4.1 Implementierung der Berechnung der technischen Indikatoren

4.1.1 Die abstrakte Basisklasse `ChartIndicator` und die Sicherstellung der einfachen Erweiterbarkeit

Die abstrakte Klasse `ChartIndicator` dient als Basisklasse für alle Indikatoren. Sie spezifiziert das Verhalten eines Indikators aus Sicht der sonstigen Teile der Anwendung vollständig. Das heißt insbesondere, im Modell, der Ansicht und der Steuerung nur Attribute und Parameter vom Typ `ChartIndicator` auftreten und nirgends solche von einem der Subtypen. Die verfügbaren Typen von Indikatoren werden im Aufzählungstyp (Enum) `EChartIndicator` beschrieben. Dieser besitzt eine Methode `EChartIndicator::getIndicator()`, die als Factory dient und einen Indikator von dem Typ, den das Enum repräsentiert, zurückliefert. Eine Erweiterung von Stocker um einen weiteren Indikatortypen ist sehr einfach: Dazu muss nur 1.) der neue Indikator in einer neuen Subklasse von `ChartIndicator` implementiert werden, und dann 2.) dieser neue Typ in der Definition von `EChartIndicator` hinzugefügt werden, und zwar einerseits als möglicher Wert des Enums, und andererseits in der Methode `EChartIndicator::getIndicator()`, die als Factory dient.

Verschiedene Indikatoren liefern verschieden viele Werte. Zum Beispiel liefert der gleitende Mittelwert nur einen Wert pro Zeitpunkt, während die Bollinger-Bänder drei Werte spezifizieren (oberes Band, Mittelwert, unteres Band). Um die Bollinger-Bänder (oder potenzielle weitere Indikatoren mit mehr als einem Wert) als einen einzigen Indikator abbilden zu können, besitzt `ChartIndicator` eine Liste von Listen von Werten (konkret eine `LinkedList<LinkedList<Double>>`): Die „äußere“ Liste umfasst die verschiedenen Indikator-Werte (im Falle des gleitenden Mittelwerts hat sie also nur ein Element, den Mittelwert; im Falle der Bollinger-Bänder drei Elemente für oberes Band / Mittelwert / unteres Band). Die „innere“ Liste enthält die Werte an den Zeitpunkten (die zugehörigen Zeit-Werte stehen in einer separaten Liste).

Des Weiteren sieht `ChartIndicator` Methoden vor, um sich die äußere oder eine beliebige innere Liste, oder Iteratoren auf eine dieser Listen zu beschaffen; um eine Liste von Kerzen (`LinkedList<Candle>`) zuzuweisen, auf der dieser Indikator

basiert; um den Indikator aktiv oder passiv zu setzen (angezeigt im Chart oder nicht); um die Berechnung durchzuführen (`calculate()`); um eine Liste von Textfeldern nebst beschreibender Texte zur Verfügung zu stellen, die zur Parametrisierung durch den:die Nutzer:in in einem Dialog von `JOptionPane` genutzt werden können (`getParametersMessage()`) und auch um den Indikator von ebendiesen Textfeldern zu parametrisieren (`parametrizeFromTextfields()`). Verschiedene Varianten von *String*-Repräsentationen bekommt man mit `toString()` und `getType()`. Die Methoden `getParameters()` und `setParameters(int[])` liefern bzw. setzen die Parameter als Integer-Array, was zur Serialisierung der Indikatoren im Rahmen der Sitzungsverwaltung nötig ist.

4.1.2 Die Klassen für den gleitenden Mittelwert und die Bollinger Bänder

Die konkreten Indikatortypen wurden in den Klassen `ChartIndicatorSMA` (SMA für *Simple Moving Average*) und `ChartIndicatorBollingerBands` umgesetzt. Beide sind selbstverständlich von `ChartIndicator` abgeleitet und implementieren dessen abstrakte Methoden, darunter insbesondere `calculate()` sowie die Methoden, die mit dem Setzen und Abfragen der Parameter zusammenhängen. Die Parameter werden in Attributen der Subklassen gespeichert. Die Werte der Indikatoren stehen in der Liste von Listen, die von der Superklasse `ChartIndicator` geerbt wurde (die „innere(n)“ Liste(n) mit den tatsächlichen Werten sind allerdings Attribute der Subklassen). Da für die Berechnung der Bollinger-Bänder ein gleitender Mittelwert erforderlich ist, benutzt `ChartIndicatorBollingerBands` intern eine Instanz von `ChartIndicatorSMA`. Die tatsächliche Berechnung in `calculate()` entspricht der Beschreibung aus der Aufgabenstellung. Da sowohl die zugrundeliegenden Candles, als auch die berechneten Werte in doppelt verketteten Listen (`LinkedList`) vorliegen, die keinen effizienten wahlfreien Zugriff unterstützen, wird mit Iteratoren gearbeitet, so dass immer nur schrittweise von einem zum nächsten oder vorherigen Element gegangen wird (vorwärts mit `next()` oder rückwärts mit `previous()` oder auch abwechselnd, je nach Bedarf).

Die Berechnung der Indikator-Werte erfolgt auf den diskreten Werten in den Candles so wie sie vorliegen, ohne Berücksichtigung der Zeitskala. Das bedeutet beispielsweise: Ein Gleitender Mittelwert mit Parameter $n = 20$ bezieht sich auf die zurückliegenden 20 Tage, wenn als Chart-Intervall „1 Tag“ ausgewählt ist, aber auf die zurückliegenden $40 = 20 \cdot 5$ Minuten, wenn „5 Minuten“ als Chart-Intervall gewählt ist. Dieses Verhalten ermöglicht eine sinnvolle Darstellung der Indikatoren auch auf den kurzen Intervallen und ist konsistent mit den Indikator-Werten von `finnhub.io`, wie sie über die REST-API mit `/indicator?...` geliefert werden.

4.2 Implementierung der Darstellung der Daten in Charts

Die Darstellung von Charts erfolgt in Chart-Kindfenstern, die die durch Objekte des Typs `StockerChart` beschrieben werden (abgeleitet von `JInternalFrame`). Die tat-

sächliche Darstellung darin wird vom `ChartPanel` (abgeleitet von `JPanel`) übernommen. Um die Daten bei beliebigen Fenstergrößen darstellen zu können, unternimmt das `ChartPanel` im Wesentlichen zwei Schritte der Skalierung:

4.2.1 Skalierungsschritt 1: Vom Daten-Koordinatensystem ins Referenz-System des Panels

Im ersten Schritt erfolgt eine Skalierung vom Daten-Koordinatensystem (x-Achse: Zeitstempel, Typ `long`; y-Achse: Preis, Typ `double`) ins Referenz-Koordinatensystem des Panels (x- und y-Achse: Koordinaten in Pixel, Typ `int`). Das Referenz-Koordinatensystem ist immer genau so breit und hoch wie das Panel, das heißt, alle Werte in diesem System sollten zwischen 0 und der Panelbreite bzw. -höhe liegen (es gibt leider Ausnahmen, siehe weiter unten). Die Berechnung erfolgt in der Methode `StockerChart::calculateScaledValues()`. Dort werden die Minima und Maxima in Richtung beider Koordinatenachsen bestimmt, und dann mittels einer linearen Abbildung die skalierten Koordinaten des Referenz-Koordinatensystems berechnet. Für die x-Achse:

$$x_i^s = x_{ref} \frac{t_i - t_{min}}{t_{max} - t_{min}} \tag{1}$$

wobei x_i^s der *s*kalierte Wert im Referenz-Koordinatensystem für den i-ten Datenpunkt ist (zwischen 0 und Panelhöhe), x_{ref} ist die Höhe des Panels, t_i ist der Zeitstempel des i-ten Datensatzes und t_{min} bzw. t_{max} sind der minimale bzw. maximale Wert des Zeitstempels in der Datenreihe. Die Skalierung der y-Werte erfolgt analog, und zwar für jeden y-Wert (also jeweils für die Werte *open*, *close*, *low* und *high*, die in den Candles der Datenreihe auftreten). Beispielhaft für die *low*-Werte:

$$y_i^{low,s} = y_{ref} \frac{y_i^{low} - y_{min}^{close}}{y_{max}^{close} - y_{min}^{close}} \tag{2}$$

Wie man hieran sieht, wird für alle vier y-Werte einheitlich mit den min/max-Werten von y^{close} skaliert, da die Skalierung der vier y-Werte einheitlich sein muss, um sie gemeinsam Zeichnen zu können. Dies führt allerdings insofern zu Problemen, als beim Kerzen-Chart der höchste Wert von $y_i^{high,s}$ und der niedrigste Wert von $y_i^{low,s}$ deutlich außerhalb des Referenzsystems liegen können (also größer als die Panelhöhe bzw. kleiner als 0). Dieses Problem wird in Schritt 2 behoben.

Analog findet diese Skalierung auch für die Indikatoren und die Alarmwerte statt, allerdings in den separaten Methoden `ChartPanel::calculateScaledIndicators()` und `ChartPanel::calculateScaledAlarms()`.

4.2.2 Skalierungsschritt 2: Vom Referenz-System des Panels ins gezeichnete Koordinatensystem

Im zweiten Schritt wird vom Referensystem des Panels in das gezeichnete Koordinatensystem transformiert. Das gezeichnete Koordinatensystem ist das, welches für

Benutzer:innen über die eingezeichneten Koordinatenachsen sichtbar ist. Es unterscheidet sich vom Referenzsystem des Panels dadurch, dass rundherum ein Rand gelassen wird (um die Achsenbeschriftungen aufzunehmen) und dass die y-Achse nicht, wie in Swing üblich, von oben nach unten, sondern von unten nach oben geht. Beide Systeme sind ganzzahlig (Typ `int`). Zur Verringerung der Rundungsfehler werden allerdings Zwischenschritte mit Gleitkommazahlen (`double`) ausgeführt.

Dieser zweite Skalierungsschritt wird von den Methoden `ChartPanel::x(int x)` und `ChartPanel::y(int y)` übernommen (die Namen der Methoden sind sehr kurz gewählt, weil diese beim Zeichnen sehr häufig benutzt werden zum Zeichnen der einzelnen Datenpunkte). Die Berechnung der Zeichnungs-Koordinaten x_z und y_z geschieht folgendermaßen:

$$x_i^z = r_x + x_i^s \; s_x \tag{3}$$

$$y_i^z = y_{ref} - r_y - y_i^s \; s_y \; s_y^{min/max} - o_y \; y_{ref} \tag{4}$$

Dabei sind x_i^s und y_i^s die Werte im Panel-Referenzsystem, also das Ergebnis aus Schritt 1, r_x und r_y die Breite des linken bzw. oberen Randes und s_x bzw. s_y sind Skalierungsfaktoren, die das Referenzkoordinatensystem in den beiden Richtungen so „zusammendrücken“, dass links und rechts bzw. oben und unten genug Platz für die Ränder und< Achsenbeschriftungen bleibt. Diese Skalierungsfaktoren wurden zuvor schon im Rahmen von Schritt 1 mitberechnet als $s_x = (x_{ref} - 2 \; r_x)/x_{ref}$ bzw. $s_y = (y_{ref} - 2 \; r_y)/y_{ref}$. Außerdem ist $s_y^{min/max}$ ein zusätzlicher Skalierungsfaktor, der der Tatsache Rechnung trägt, dass bei Schritt 1 in Gleichung (2) einheitlich mit den min/max-Werten von y^{close} skaliert wurde, und somit das maximale y^{high} und das minimale y^{low} außerhalb des Zeichenbereichs liegen können. Er wird wie folgt berechnet:

$$s_y^{min/max} = \frac{y_{max}^{close} - y_{min}^{close}}{y_{max}^{high} - y_{min}^{low}}$$

und gibt an, um wie viel die Differenz der extremen *close*-Werte kleiner ist als die Differenz zwischen dem größten und kleinsten Wert insgesamt. Anschaulich betrachtet dient er in Gleichung (4) dazu, den y^s-Wertebereich aus Schritt 1 nochmal so zusammenzustauchen, dass auch der größte Wert von y^{high} und der kleinste Wert von y^{low} ins gezeichnete Koordinatensystem passt.

Das ist aber leider immer noch nicht ganz ausreichend, denn die zusätzliche Skalierung mit $s_y^{min/max}$ berücksichtigt nicht, ob wegen y^{high} oder wegen y^{low} oder wegen beiden Werten skaliert wurde, und könnte es sein (wenn man den letzten Term in Gleichung (4) weg ließe), dass der gezeichnete Plot zwar die richtige Höhe hat, aber zu weit oben oder unten im Panel sitzt bzw. sogar oben oder unten über das Panel hinausragt. Dies wird in Gleichung (4) mit dem Offset-Faktor o_y korrigiert: Er beschreibt die Verschiebung nach oben oder unten, die nötig ist, um den Plot sauber in der Mitte des Panels sitzen zu haben. Ermittelt wird er ebenfalls einmalig im Zuge der Berechnungen für Schritt 1 (in `ChartPanel::calculateScaledValues()` durch Überprüfung

des Ergebnisses von `ChartPanel::y(`$y_{max}^{high,s}$`)` und `ChartPanel::y(`$y_{min}^{low,s}$`)`: Wenn diese außerhalb des Zeichenbereichs liegen, wird iterativ der Offset-Faktor o_y angepasst und erneut überprüft. In seltenen Fällen mit extremen Werten kann es nötig sein, zusätzlich noch $s_y^{min/max}$ zu verkleinern.

4.2.3 Das Zeichnen

Das Zeichnen im engeren Sinne ist auf verschiedene Methoden aufgeteilt, nämlich (alle in `ChartPanel`): `drawCoordinateLines(...)` (zeichnet die Koordinatenachsen), `drawTicks(...)` (Striche an den Koordinatenachsen sowie Achsenbeschriftungen), `drawDataLine(...)` (Linien-Plot), `drawDataCandles(...)` (Kerzen-Plot), `drawIndicators(...)` (zeichnet alle Indikatoren), `drawAlarms(...)` (Alarme) und `drawCrosslines(...)` (zeichnet das Fadenkreuz über das ganze Panel).

Allerdings hat sich herausgestellt, dass es ungünstig ist, alle diese Zeichenoperationen jedes Mal direkt in `paintComponent(...)` auszuführen, weil diese Methode relativ oft aufgerufen wird. Zum Beispiel wird sie von Swing auch dann aufgerufen, wenn eine anderes Chart-Fenster, was sich mit diesem Chart-Fenster überlappt, neu gezeichnet wird, oder wenn ein:e Nutzer:in mit der Maus über das Chart fährt und das Fadenkreuz ständig neu gezeichnet wird. Das ist in Summe doch recht aufwändig, kann zu signifikanter CPU-Auslastung führen und lässt die Anwendung für die Benutzer:innen spürbar träge werden. Deshalb wird als Zwischenschritt in der Methode `ChartPanel::paintImage()` ein Bild (`BufferedImage`) gezeichnet mithilfe der im vorangehenden Absatz erwähnten Zeichen-Methoden (bis auf `drawCrosslines(...)`). In `paintComponent(...)` wird dann nur noch dieses Bild gezeichnet mit `Graphics2D::drawImage(...)`. Darüber wird dann noch das Fadenkreuz gezeichnet. So geht das (häufige) Zeichnen des Panels in `paintComponent(...)` relativ schnell und ressourcenschonend, während das eher aufwändige Zeichnen des Bildes mit `paintImage` nur noch nötig ist, wenn sich die Daten oder die Panel-Größe geändert haben.