Konzeption und Implementierung eines Portfolio-Managements

Bachelorarbeit
Technische Universität Dresden
März 2025

Philipp Jonscher

Betreuer: Prof. Dr.-Ing. Klaus Meißner
Hochschullehrer: Prof. Dr.-Ing. Klaus Meißner, Prof. Dr.-Ing. habil. Rainer Groh

Fakultät Informatik
Institut für Software- und Multimediatechnik
Seniorprofessur für Multimediatechnik

EINGEGANGEN
18. OKT. 2024

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Prüfungsamt

Anmeldung zur Bachelorarbeit

Persönliche Angaben der/des Studierenden:

Name, Vorname: Jonscher, Philipp

geb. am: 03.07.2000 Matrikelnummer: 4971364

Studiengang: Informatik (2009)

E-Mail-Adresse: philipp.jonscher@mailbox.tu-dresden.de

☐ Antrag auf Erstellung der Arbeit in englischer Sprache (§21 (6) Prüfungsordnung)

Thema der Bachelorarbeit in deutscher Sprache (Kurztitel):

Konzeption und Implementierung eines Portfolio-Managements

Thema der Bachelorarbeit in englischer Sprache (Kurztitel): (Bitte unbedingt angeben!)

Conception and implementation of a portfolio management

Wir sind mit dem oben genannten Thema einverstanden und übernehmen je ein Gutachten:

1. Gutachter: Prof. Dr.-Ing. Klaus Meißner
(Akademischen Grad immer mit angeben)

(falls kein HSL, prüfungsberechtigt laut FR-Beschluss vom ...)

Professur: Seniorprofessur für Multimediatechnik

☒ Ich betreue die Arbeit 03.10.2024 .....
(Datum, Unterschrift)

2. Gutachter: Prof. Dr.-Ing. habil. Rainer Groh
(Akademischen Grad immer mit angeben)

(falls kein HSL, prüfungsberechtigt laut FR-Beschluss vom ...)

Professur: Professur für Mediengestaltung

☒ Ich betreue die Arbeit 16.10.24 .....
(Datum, Unterschrift)

Beginn: 28.10 2024 Abgabetermin: 13.01 2025

Die Bearbeitungszeit der Bachelorarbeit beträgt 11 Wochen (siehe §28 der Prüfungsordnung)

Datum und Unterschrift der/des Studierenden: 02.10.2024, Jonscher

Der Prüfungsausschuss stimmt dem Antrag zu:

Datum: 21. Okt. 2024 Vors. des Prüfungsausschusses: .....

Verteiler nach Rücklauf: Original Studierende/r, Kopie 1. Gutachter, Kopie 2.Gutachter, Kopie Prüfungsamt

Version 23.03.2023

TECHNISCHE UNIVERSITÄT DRESDEN

eingegangen
15. JAN. 2025

Fakultät Informatik/Prüfungsamt

Prüfungsausschuss des Studiengangs / Examination Board of the Degree Program

Informatik

Antrag auf Verlängerung der Bearbeitungszeit der Abschlussarbeit / Application for an extension of the processing time for the final thesis

Jonscher

Name / Family Name

Philipp

Vorname / First Name

4971364 — Matrikel-Nummer / Matriculation Number

Bachelor — Hochschulgrad / University Degree

Thema (Titel) der Abschlussarbeit / Topic (Title) of the final thesis:

Konzeption und Implementierung eines Portfolio-Managements

Prof. Dr.-Ing. Klaus Meißner

Betreuender Hochschullehrer /Supervising Professor

28.10.2024

Beginn der Bearbeitungszeit / Start of Processing Time

22.01.2025

Abgabetermin /Submission Deadline

6 Wochen

Dauer der Verlängerung / Duration of Extension*

05.03.2025

Neuer Abgabetermin / New Deadline

*Maximale Verlängerungsdauer siehe Prüfungsordnung/Maximum period of extension according to Examination Regulation
Begründung der Verlängerung siehe Rückseite / Reason(s) for the Extension see backpage

**Begründung der Verlängerung / Reason(s) for the Extension

Ausgangspunkt meiner Bachelorarbeit ist ein komplexes Softwaresystem, dass um die Funktion des Portfolio-Management erweitert werden sollte. Erst während der Bearbeitung wurde deutlich, dass die bestehende Implementierung zur Navigation substanziell geändert werden musste. Diese notwendigen Anpassungen waren zeitaufwändig und in der ursprünglichen Planung nicht berücksichtigt. Außerdem ergab sich substanzieller zusätzlicher Aufwand bezüglich der Vermeidung von Datenbank-Inkonsistenzen.

10.01.2025

Datum / Date

Unterschrift Student / Signature Student

Zustimmung durch betreuender Hochschullehrer /Approval by Supervising Professor:

10. 01. 2025

Datum / Date

Unterschrift betreuender Hochschullehrer / Signature Supervising Professor

Entscheidung des Prüfungsausschusses / Decision of the Examination Board:

Dem Antrag auf Verlängerung wird / The application for extension is

☒ stattgegeben / approved

☐ nicht stattgegeben / not approved

Datum / Date 5.1.2025

Unterschrift Prüfungsausschuss / Sign. Examination Board

Dieser Antrag ist mindestens 3 Wochen vor dem ursprünglichen Abgabetermin der Abschlussarbeit abzugeben. / This application must be submitted at least 3 weeks before the original deadline of the final thesis.

The English version is the translation of the German original. / Only the German version is legally binding.

Technische Universität Dresden

Fakultät Informatik

Aufgabenstellung für eine Bachelorarbeit

Name, Vorname des Studenten: Jonscher, Philipp
Matrikelnummer des Studenten: 4971364

Thema: **Konzeption und Implementierung eines Portfolio-Managements**

Zielstellung:

Im Forschungsprojekt „Intelligent Personalized Wealth Management" (SmartWM) werden Strategien der Vermögensverwaltung untersucht. Diese basieren auf sehr heterogenen Informationen, sowohl semantisch beschriebener unstrukturierter wie auch strukturierter Daten zu Anlageobjekten verschiedenster Anlageformen. Diese Informationen sollen personalisiert und intelligent so aufbereitet werden, dass „normale" Anleger Veränderungen von Anlagen erkennen und Anlageentscheidungen treffen können.

Ausgangspunkt dieser Arbeit sind die Ergebnisse bestehender Abschlussarbeiten zur Erfassung und grafischen Darstellung von Kursdaten sowie zur Konzeption der Benutzeroberfläche der Vermögensverwaltung SmartWM. In **dieser Aufgabenstellung** soll nun das bestehende Java-Programm um Funktionen zur Verwaltung von Portfolios, Depots und Konten erweitert werden. Zur Verwaltung gehören z. B. auch Funktionen für das Einrichten, Eröffnen und Deaktivieren von Portfolios, Depots sowie Konten, zum An- und Verkauf von Wertpapieren sowie zur Erfassung von Erträgen. Zudem sollen für die so erfassten Depots, Wertpapiere und Daten tabellarische Übersichten von Depotveränderungen konzipiert und implementiert werden.

Konkret umfasst die Arbeit u. a. folgende Aufgaben:

- Überprüfung und Überarbeitung des Konzeptes zur Portfolio-Verwaltung.
- Konzeption der wesentlichen Funktionen des Portfolio-Managements, der Inhaber-, Depot- und Konto-Verwaltung sowie des Order- und Transaktions-Managements.
- Implementierung ausgewählter Funktionen des Portfolio-Managements, der Inhaber-, Depot- und Konto-Verwaltung sowie des Order- und Transaktions-Managements.
- Test und Evaluation der Erweiterung des vorhandenen Java-Programms SmartWM sowie der MySQL-Datenbank um die zuvor beschriebenen Konzepte und Funktionen.

Insbesondere sind in dieser Abschlussarbeit folgende Teilziele zu erreichen:

- Einarbeitung in die vorhandenen Vorarbeiten, insbesondere in den Quellcode des Java-Programms SmartWM.
- Entwicklung der zuvor genannten Konzepte zur Verwaltung von Portfolios, Depots und Konten.
- Implementierung der Konzepte als Erweiterung des Programms SmartWM.
- Test der Implementierung und Überprüfung der Konzeption mit noch festzulegenden Wertpapieren und Web-Seiten.

Betreuer, verantwortlicher Hochschullehrer: Prof. Dr.-Ing. Klaus Meißner
Zweitprüfer: Prof. Dr.-Ing. habil. Rainer Groh
Institut: Software- und Multimediatechnik
Beginn am: 28.10.2024
Einzureichen am: 13.01.2025

Verteiler: Prüfungsamt
HSL
Student

03.10.2024 ..... Jonscher

Unterschrift des Hochschullehrers Studenten

Erklärung

Hiermit erkläre ich, Philipp Jonscher, die vorliegende Bachelorarbeit zum Thema

Konzeption und Implementierung eines Portfolio-Managements

selbstständig und ausschließlich unter Verwendung sowie korrekter Zitierung der im Literaturverzeichnis angegebenen Quellen und Hilfsmittel verfasst zu haben.

Dresden, 2. März 2025 Jonscher

Inhaltsverzeichnis

1 Einführung

Eine strukturierte und effiziente Verwaltung von Finanzanlagen ist essenziell, um komplexe Vermögensstrukturen übersichtlich zu organisieren. Digitale Lösungen spielen dabei eine zentrale Rolle, indem sie Prozesse automatisieren und wichtige Informationen schnell zugänglich machen. Insbesondere in einem dynamischen Finanzumfeld müssen Nutzer jederzeit einen intuitiven Überblick über ihre Investitionen haben, neue Strukturen einfach anlegen und bestehende Daten effizient verwalten können. Darüber hinaus ermöglichen definierte Richtlinien eine gezielte und zukunftsorientierte Steuerung von Depots.

In dieser Arbeit wird das Programm SmartWM um spezifische Funktionen zur Verwaltung von Inhabern, Portfolios, Konten und Depots erweitert. Dazu zählen unter anderem das Anlegen und Bearbeiten von Inhabern sowie eine übersichtliche Darstellung relevanter Kenndaten, einschließlich zugehöriger Portfolios, Konten und Depots. Zudem sollen die Verwaltungskomponenten eine tabellarische Übersicht bieten, die beispielsweise alle registrierten Inhaber auflistet. Darüber hinaus werden Mechanismen zur Vermeidung sowie der Erkennung und Behebung von Datenbank-Inkonsistenzen realisiert.

Während der Implementierung kommen KI-Tools zum Einsatz, die in einer früheren Arbeit von Herrn Sömisch ([Söm24]) untersucht wurden. Ziel ist es, die Praxistauglichkeit und den Mehrwert ausgewählter Werkzeuge anhand der Implementierung von Erweiterungen zu SmartWM zu testen. Dies soll nicht nur die Effizienz der Implementierung in dieser Arbeit und künftigen Projekten erhöhen, sondern auch die Struktur und Qualität des Quellcodes optimieren.

Im weiteren Verlauf dieser Arbeit werden zunächst grundlegende Begriffe geklärt, die als Basis für das Verständnis der folgenden Kapitel dienen. Anschließend werden Defizite aufgearbeitet, die mit dem Stand der Bachelorarbeit von Herrn Görg bestehen. Daraufhin erfolgt eine Analyse der definierten Anforderungen, die durch die Entwicklung von Mockups veranschaulicht werden. Im nächsten Schritt wird die Konzeption des Quellcodes mit Hilfe von UML-Diagrammen dargestellt, um die Struktur und Funktionalität der Software zu planen. Die anschließende Implementierung des Konzepts erfolgt in einer Full-Stack-Umsetzung. Dabei wechseln sich Konzeption und Implementierung periodisch ab, um schrittweise vorzugehen. Zum Beispiel wurde zunächst die Inhaber-Verwaltung konzipiert und anschließend implementiert, bevor bspw. die Portfolio-Verwaltung zunächst konzeptioniert und anschließend implementiert wurde. Dieses iterative Vorgehen ermöglicht eine enge Verzahnung von Planung und praktischer Umsetzung, was die Entwicklung effizient und zielgerichtet gestaltet.

2 Begrifflichkeiten

2.1 Inhaber

Der Begriff *Inhaber* bezeichnet eine natürliche Person, die Eigentümer von Portfolios, Konten und Depots sind (vgl. [Mei24, Seite 8]).

2.2 Portfolio

Ein Portfolio kombiniert Konten und Depots zu einer auswertbaren Einheit und kann als Container betrachtet werden, der diese Elemente zusammenfasst (vgl. [Mei24, Seite 8]).

2.3 Konto

Ein Konto dient als Basis für den Geldverkehr eines Kunden und enthält dessen Guthaben. Es spielt in der Vermögensverwaltung eine zentrale Rolle, unterscheidet sich jedoch je nach Funktion. Typische Konten sind unter anderem Girokonten, Sparkonten, Festgeldkonten, Tagesgeldkonten, Darlehenskonten sowie Depot-Verrechnungskonten (vgl. [Mei24, Seite 8]).

2.4 Depot

Ein Depot, häufig als Wertpapierdepot bezeichnet, ist die Aufbewahrungs- und Verwaltungseinheit für Wertpapiere wie Aktien, Anleihen und Fonds. Jedem Depot ist mindestens ein Verrechnungskonto zugeordnet, auf dem die zugehörigen Cash-Reserven verwaltet werden. (vgl. [Mei24, Seite 8])

2.5 Entität

Eine Entität (z. B. Datenbank-Entität) ist ein datenbezogenes Konzept, das die konkreten Werte für die definierten Attribute und Beziehungen speichert bzw. bereitstellt. (vgl. [Wik24a])

2.6 Fake-Inhaber/-Portfolio/-Konto

Der Begriff „Fake“ bezeichnet im Kontext dieser Arbeit eine bewusst unvollständige bzw. vereinfachte Instanz eines Objekts, die nur die essenziellen Attribute enthält und nicht alle Eigenschaften oder Verknüpfungen der Entität abbildet. Diese Fake-Instanzen dienen dazu, eine minimale, aber dennoch funktionale Darstellung eines Objekts zu ermöglichen.

2.7 Listeners

Ein Ereignis-Listener (kurz: Listener) ist eine Funktion, die ein bestimmtes Ereignis überwacht und bei dessen Eintreten entsprechend reagiert. (vgl. [AWS])

3 Anforderungen an diese Arbeit

Zu Beginn dieser Arbeit bestand die Aufgabe darin, sich in den Quellcode des Programms einzuarbeiten und bestehende Defizite [Mei24, Seite 28] zu beheben.

Der Hauptfokus liegt jedoch auf der Konzeption und Implementierung zentraler Funktionen für das Portfoliomanagement. Dazu zählt die Inhaber-Verwaltung, die eine Übersicht aller registrierten Inhaber bereitstellt und die Möglichkeit bietet, neue Inhaber zu erstellen. Zudem sollen für jeden Inhaber spezifische Informationen abrufbar sein, darunter eine Übersicht der eingegebenen Parameter sowie Ansichten der zugehörigen Portfolios, Konten und Depots. (vgl. [Mei24, Seiten 34–37]) Ein weiteres Modul ist die Portfolio-Verwaltung, die eine Liste aller registrierten Portfolios umfasst und die Anlage neuer Portfolios ermöglicht. Analog zur Inhaber-Verwaltung wird für jedes Portfolio eine detaillierte Ansicht bereitgestellt, in der die eingegebenen Parameter eingesehen werden können. (vgl. [Mei24, 37f.]) Die Konto-Verwaltung bietet eine Übersicht über alle registrierten Konten und ermöglicht die Erstellung neuer Konten. Auch hier gibt es eine spezifische Ansicht für jedes Konto, in der die relevanten Parameter dargestellt werden. (vgl. [Mei24, Seiten 41–42]) Abschließend umfasst die Depot-Verwaltung eine Übersicht aller registrierten Depots sowie die Möglichkeit, neue Depots anzulegen. (vgl. [Mei24, Seiten 44–46])

4 Vorbereitende Arbeiten

Um sich einen umfassenden Überblick über den bestehenden Quellcode zu verschaffen, bestand die erste Aufgabe – wie im vorherigen Kapitel erwähnt – darin, die Defizite aufzuarbeiten, die mit dem Stand der Bachelorarbeit von Herrn Görg bestehen.

4.1 Unstimmigkeit in der Benutzeroberfläche

Beim Auswählen des Menüpunktes *Daten* wurde dieser dunkler dargestellt, während die Menüpunkte der nächsten Menüebene, wie beispielsweise *Wertpapier-Kurs*, bei Auswahl heller dargestellt wurden (vgl. Abbildung 4.1).

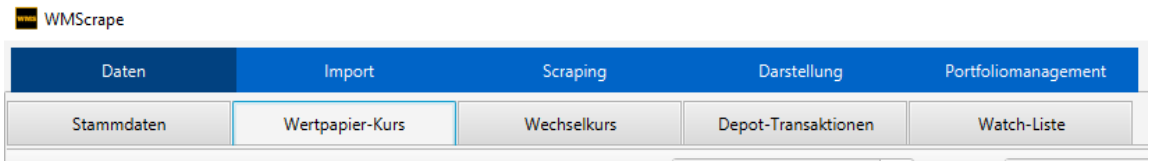

Abbildung 4.1: Menü-Darstellung vor der Behebung

Bislang wurden Menüpunkte, wie *Daten*, entweder in der *initialize*-Methode der jeweiligen Controller-Klasse[1] (z. B. *PrimaryTabController* für die erste Menüebene mit den Tabs *Daten*, *Import*, usw.) erstellt oder direkt in der FXML-Datei definiert und anschließend im Controller referenziert. Um die Tab-Erstellung zu vereinheitlichen und ein konsistentes Design zu gewährleisten, wurde zunächst die Methode *createStyledTab* in der Klasse *PrimaryTabController* um zwei neue Parameter ergänzt, die die Farbwerte für die Darstellung eines Tabs bei Auswahl und Nicht-Auswahl als hexadezimale Werte definieren. Diese Methode dient zur Erzeugung von Tabs, bei denen die Hintergrundfarbe sich entsprechend des Auswahlstatus anpasst.
Zusätzlich wurden der Klasse zwei neue Methoden hinzugefügt: *createPrimaryTab* und *createSubTab*. Beide rufen die *createStyledTab*-Methode auf, verwenden jedoch unterschiedliche Farbwerte:

- *createPrimaryTab*: Erstellt Tabs der ersten Menüebene, wie z. B. *Daten*.
- *createSubTab*: Generiert Tabs der zweiten Menüebene, wie *Stammdaten*, mit einer etwas helleren Farbgebung im Vergleich zur ersten Ebene.

[1] Klassen die als Controller fungieren.

Die Controller-Klassen, die die Tabs selbst nicht erstellten, sondern lediglich referenzierten, wurden entsprechend angepasst, sodass Tabs nun über die Methoden *createPrimaryTab* bzw. *createSubTab* erstellt werden. Diese Änderungen führten dazu, dass auch die Tabs der zweiten Menüebene, die zuvor grau gefärbt waren, nun eine bläuliche Färbung erhielten, was das Design insgesamt harmonisierte (vgl. Abbildung 4.2).

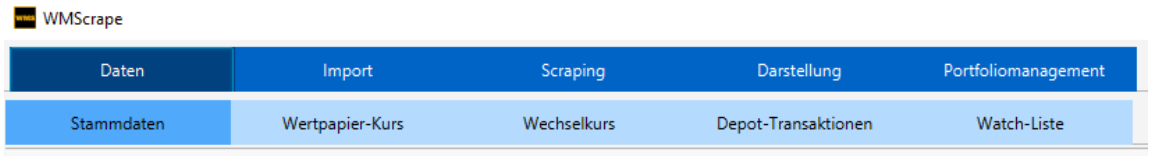

Abbildung 4.2: Menü-Darstellung nach der Behebung

4.2 Scrapen: Unterschiedliche X-Path-Ausdrücke bei verschiedenen Wertpapiertypen

Es stellte sich heraus, dass zum Beispiel der X-PATH-Ausdruck der Elemente auf Webseiten, wie beispielsweise auf der Frankfurter Börse[2], je nach Wertpapiertyp (Aktien, ETFs, usw.) unterschiedlich ist. Daraus folgt, dass die Webseiten-Konfiguration für unterschiedliche Wertpapiertypen individuell angepasst werden muss. (vgl. [Mei24, Seite 28])

Dieses Problem wurde nur rudimentär von Herrn Görg [Gör23, Seite 67] gelöst. Dabei wurden in der Klasse *WebsiteHandler* verschiedene Methoden wie *boerseFrankfunkfurtLoadHP* und *replaceXPATHFB* implementiert, die festcodierte Identifikatoren verwenden, um auf der Webseite nach Elementen zu suchen oder bestimmte Stellen in X-PATH-Ausdrücken aus der Konfiguration durch vordefinierte Terme zu ersetzen, falls ein Element mittels des Ausdrucks aus der Konfiguration nicht gefunden werden kann.

Im ersten Schritt wurde die Webseiten-Konfiguration angepasst. Die Eingabefelder von „Historische-Link“ bis „Seitenanzahl-Feld“, die im Abschnitt „Navigations-Automatisierung“ der Webseiten-Konfiguration zu finden waren, wurden aus der Controller-Klasse der Webseiten-Konfiguration (*HistoricWebsiteTabController*) in die neue Controller-Klasse für die TitledPanes (*SecuritiesTypeDataContainer*) verlegt. TitledPanes sind aufklappbare und schließbare Panels. Anschließend wurde die Ansicht der sogenannten TitledPanes implementiert, die es ermöglichen, für jeden Wertpapiertyp eine individuelle Konfiguration der Parameter vorzunehmen (vgl. 4.3). Um die Konfigurationsdaten zu speichern, wurden die entsprechenden Attribute (z. B. für den „Historische-Link“-Identifikator) aus der Entitäts-Klasse *Website* in die neue Entitäts-Klasse *HistoricWebsiteIdentifiers* überführt. Während erstere die allgemeinen Daten einer Webseitenkonfiguration enthält, repräsentiert die letztere die Parameter eines Wertpapiertyps. Darüber hinaus wurde der Enum *SecuritiesType* (Deutsch: Wertpapiertyp) implementiert. Dieser Enum wird verwendet,

[2] https://www.boerse-frankfurt.de

um u.a. in der Controller-Klasse der Webseiten-Konfiguration die Logik so anzupassen, dass über alle Literale des Enums iteriert wird und für jedes ein TitledPane mit der neuen Ansicht erzeugt wird.

Abschließend wurde die Testfunktion über den Button „Testen“ angepasst. Hierfür wurde die Klasse *HistoricWebsiteTester* um eine Liste erweitert, die für jeden Wertpapiertyp ein Beispielwertpapier anhand des ISBN-Wertes enthält. Diese Klasse beinhaltet die Logik der Testfunktion. Zudem wurde die Funktion so modifiziert, dass der Testprozess erst abgeschlossen wird, wenn alle Beispiele getestet wurden. Kann der Test zu einem Typ nicht erfolgreich durchgeführt werden, wird der Typ übersprungen und es beginnt der Testprozess mit dem nächsten Wertpapiertyp.

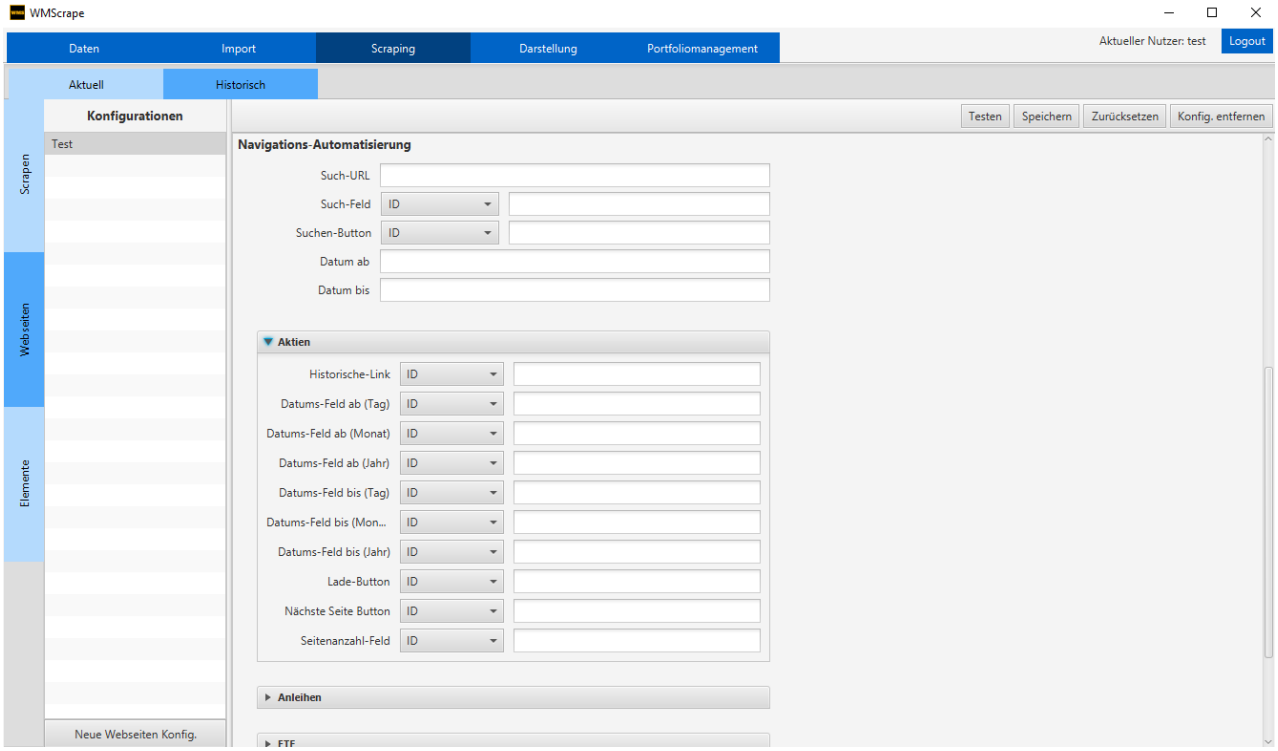

Abbildung 4.3: Neue Ansicht der Webseitenkonfiguration nach der Implementierung

Im nächsten Schritt wurde die Element-Konfiguration angepasst. Dazu wurde die Ansicht für die Zuordnung der Zeilenelemente um das Eingabefeld für den Identifikator der Tabelle mit historischen Daten ergänzt. Zusätzlich wurde eine TabPane in die Ansicht integriert, die für jeden Wertpapiertyp ein Tab enthält, welches die Ansicht der typ-spezifischen Konfiguration bereitstellt. Entsprechend wurde die Controller-Klasse der Elementenkonfiguration (*HistoricWebsiteElementTabController*) angepasst, um über alle Literale des Enums *SecuritiesType* zu iterieren und für jedes einen Tab in der TabPane zu erzeugen. Als Controller der einzelnen Tabs dient hier die *SecuritiesTypeCorrContainer*-Klasse. Außerdem wurde die Entitäts-Klasse *ElementIdentCorrelation* um drei

Attribute erweitert, um den Wertpapier-Typ, den Identifikator-Typ sowie den Identifikator-Wert für die Tabelle mit historischen Daten zu speichern. Diese Klasse repräsentiert u.a. ein Eintrag in der Tabelle zur Zuordnung der Zeilenelementen (vgl. Tabelle unten rechts in Abbildung 4.4). Beim Scrapen soll so aus einen der Einträge der Identifikator für die Tabelle der historischen Daten abgerufen werden können. Diese Lösung ist zwar nicht sehr elegant, brachte jedoch den Vorteil mit sich, dass ein umfangreiches Refactoring dadurch vermieden werden konnte.

Zuletzt wurde der Scrapingprozess angepasst. Die Methoden, die die einzelnen Schritte des Prozesses darstellen (z. B. *doLoadHistoricData* zum Scrapen historischer Kurse), welche in der Klasse *WebsiteScraper* zu finden sind, wurden so überarbeitet, dass sie die erforderlichen Parameter aus der *HistoricWebsiteIdentifiers*-Klasse beziehen. Falls für einen Wertpapiertyp keine Konfiguration angegeben ist, können Elemente dieses Typs nicht gescrapet werden. Um dem zu scrapenden Wertpapier den richtigen Literal aus *SecuritiesType*-Enum zuzuweisen, wurde in Absprache mit dem Betreuer eine neue Spalte *Stype* in die zu importierende Excel-Datei aufgenommen, die den Typ des Wertpapiers (z. B. „ANL“ für Anleihen) beschreibt. Zur Persistierung des Typs wurde die Wertpapier-Klasse (*Stock*) um ein entsprechendes Attribut erweitert. Für die Zuordnung des Spalteninhalts zu einem Literal dient dabei die Methode *getMapped* der Enum-Klasse.

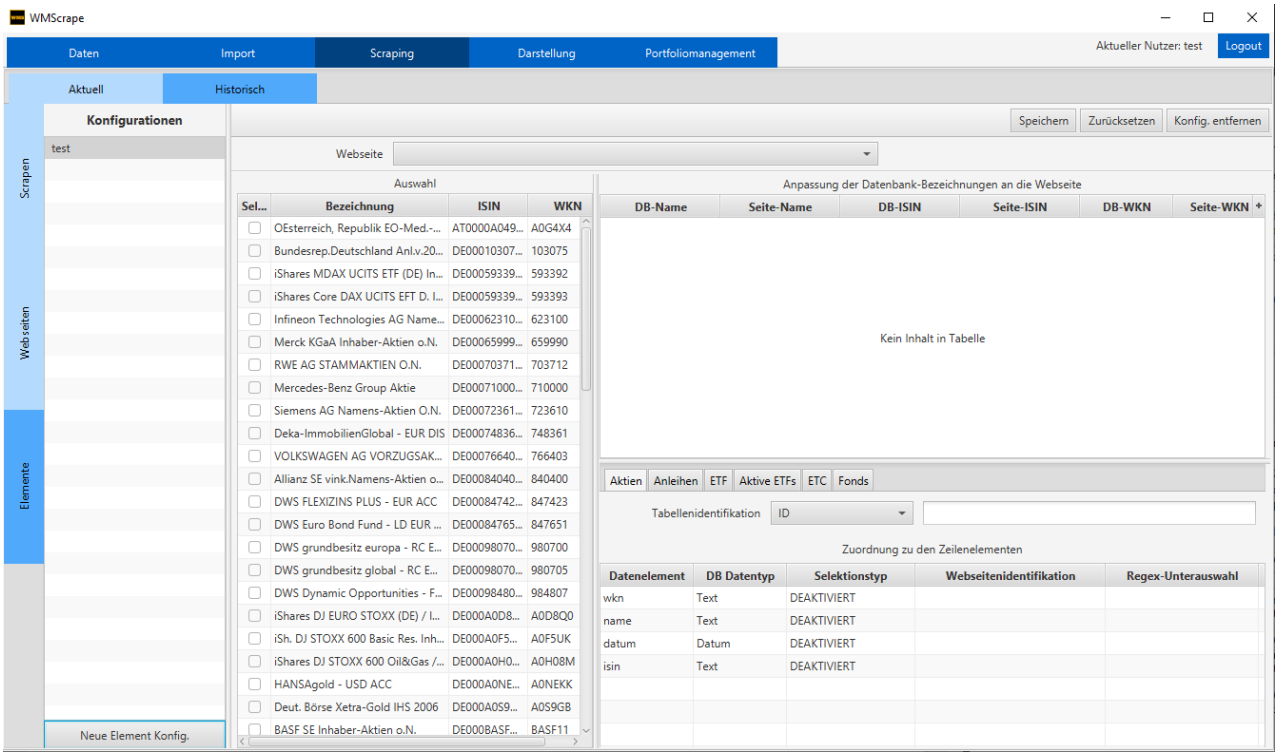

Abbildung 4.4: Neue Ansicht der Element-Konfiguration nach der Implementierung

5 Konzeption

In diesem Kapitel werden die Lösungskonzepte der zu entwickelnden Erweiterungen erläutert.

In der Konzeption wurden auch die Strukturen der Pakete berücksichtigt, unter denen die Objekte (wie Klassen usw.) organisiert werden sollen. Dies bietet den Vorteil, dass die Systemstruktur und Modularität optimiert werden, indem verwandte Klassen sinnvoll gruppiert werden. Dadurch wird die Wartbarkeit und Erweiterbarkeit des Codes erleichtert. Ein weiterer Vorteil besteht darin, dass zukünftige Studenten sich einfacher in die komplexe Erweiterung des Portfoliomanagements einarbeiten können, da sie die relevanten Objekte schneller finden. Dies basiert auf eigener Erfahrung während der Einarbeitung in den Quellcode früherer Arbeiten.

Da der verfügbare Platz begrenzt ist, werden in den folgenden Abschnitten nur die wesentlichen Aspekte dargestellt, wobei die Inhalte vereinfacht und auf das Wesentliche reduziert sind.

5.1 Inhaber-Verwaltung

Die Inhaber-Verwaltung soll dem Benutzer ermöglichen, neue Inhaber anzulegen und eine Übersicht der registrierten Inhaber bereitzustellen. Zudem soll das „Öffnen eines Inhabers“ eine spezifische Ansicht bieten, in der sowohl die Parameter des Inhabers als auch die zugehörigen Vermögen, Portfolios, Konten und Depots eingesehen werden können.

5.1.1 Hauptmenü

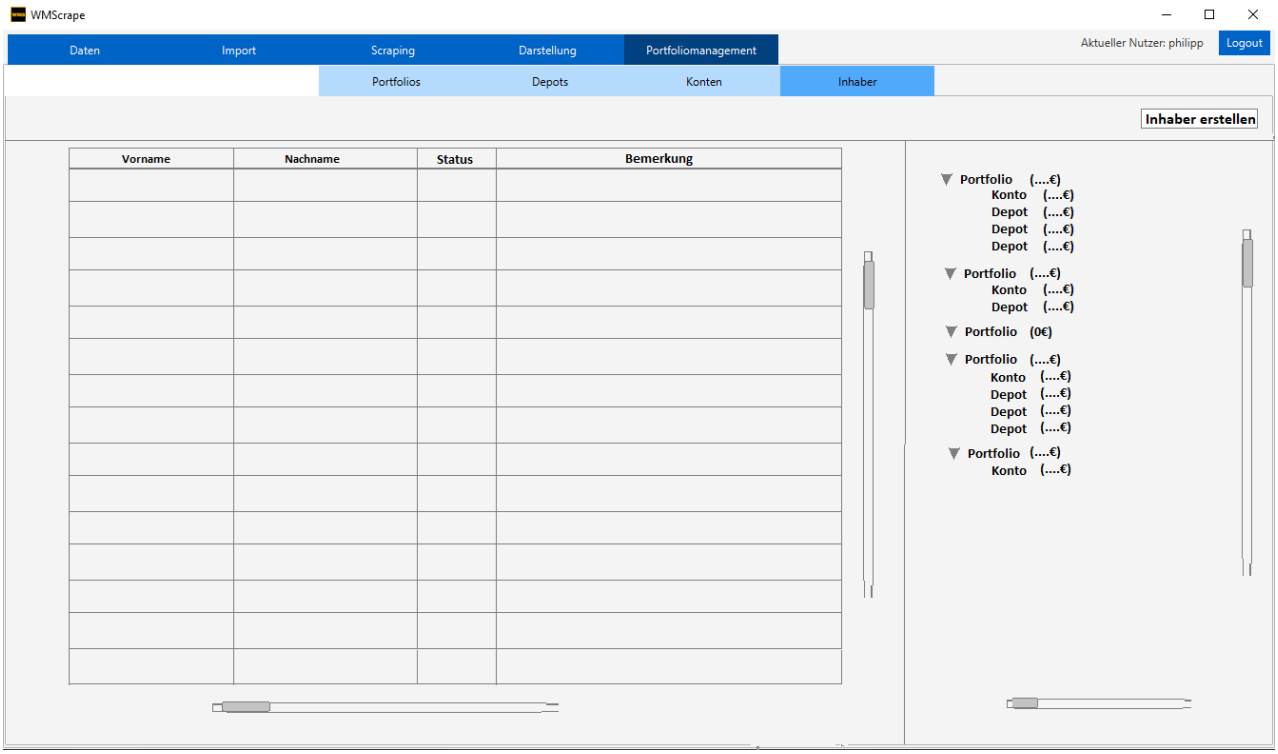

Abbildung 5.1: Mockup des Hauptmenüs der Inhaber-Verwaltung

Das Hauptmenü bietet dem Benutzer einen Überblick über die in SmartWM registrierten Inhaber, deren Portfolios sowie die darin enthaltenen Konten und Depots (vgl. Abbildung 5.1). Die Portfolios werden angezeigt, sobald ein Inhaber ausgewählt wird.

Über ein Kontextmenü ist es möglich, Inhaber zu löschen oder deren Status zu verändern. Zudem erfolgt das Öffnen der inhaber-spezifischen Ansicht durch einen Klick auf die entsprechende Tabellenzeile. Analog dazu öffnet ein Klick auf ein Portfolio, Konto oder Depot in der Baumansicht die jeweilige Übersicht (z. B. die Konto-Übersicht).

Die Pfeilsymbole ermöglichen das Ein- und Ausklappen der Portfolios, sodass Details zu den enthaltenen Konten und Depots bei Bedarf einsehbar oder ausgeblendet werden können. Zusätzlich ermöglicht der Button „Inhaber erstellen“ das Hinzufügen neuer Inhaber.

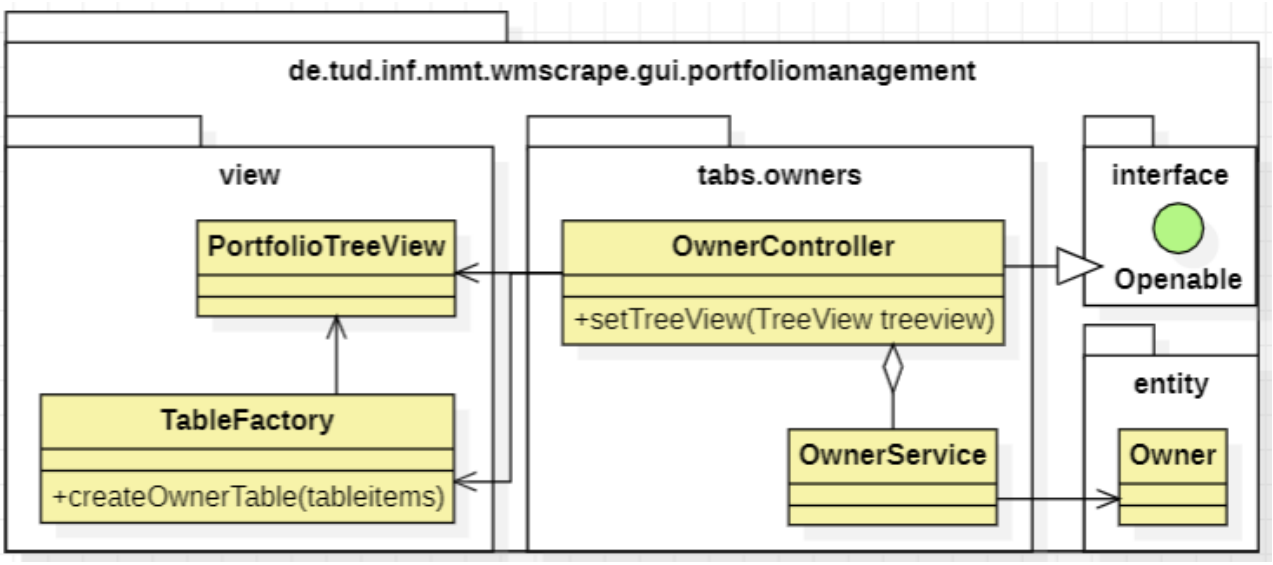

Abbildung 5.2: Grober Überblick zum Konzept des Hauptmenüs

Im Wesentlichen sorgt die Controller-Klasse (*OwnerController*) aus Abbildung 5.2 für die Ausgabe der Daten sowie die Registrierung von Ereignissen wie dem Klick auf den Button zum Erstellen eines Inhabers. Die Daten lädt sie über die Service-Klasse (*OwnerService*) aus der Datenbank, erstellt die Tabellenansicht mit Hilfe der *TableFactory*-Klasse und die initiale (leere) Baumansicht der Portfolios mit der *PortfolioTreeView*-Klasse. Zusätzlich enthält die Controller-Klasse eine Methode, welche aus der *createOwnerTable*-Methode heraus aufgerufen wird. Die *createOwnerTable*-Methode erstellt die Tabelle aus Abbildung 5.1 und erzeugt beim Auswählen eines bestimmten Inhabers die Baumstruktur-Ansicht (*PortfolioTreeView*) der dem Inhaber zugehörigen Portfolios und stellt sie anschließend über die Methode des Controllers dar.

Die *Openable*-Schnittstelle dient zur Kennzeichnung von Controller-Klassen, die ihre Daten dynamisch laden müssen – etwa nach der Erstellung eines neuen Inhabers oder bei jedem Tab-Klick. Sie definiert eine Methode, deren Implementierung darauf abzielt, spezifische Ansichtsdatensätze neu zu laden, so zum Beispiel die Inhaber-Tabelle.

Die Klassenstruktur folgt dem Prinzip des Controller-Service-Repository-Musters[1], wodurch sich die Aggregation zwischen der Controller- und der Service-Klasse ergibt. Die Service-Klasse stellt die Geschäftslogik der Inhaber-Verwaltung bereit. Zu ihren Aufgaben gehören u.a. das Abrufen und Speichern von Inhabern. Gemäß dem bereits genannten Prinzip aggregiert sie die Repository-Schnittstelle[2] *OwnerRepository*, um mit der Datenbank zu interagieren (vgl. Abbildung 5.3).

[1] https://tom-collings.medium.com/controller-service-repository-16e29a4684e5
[2] In Spring ist ein Repository eine spezielle Schnittstelle, die für den Datenzugriff zuständig ist.

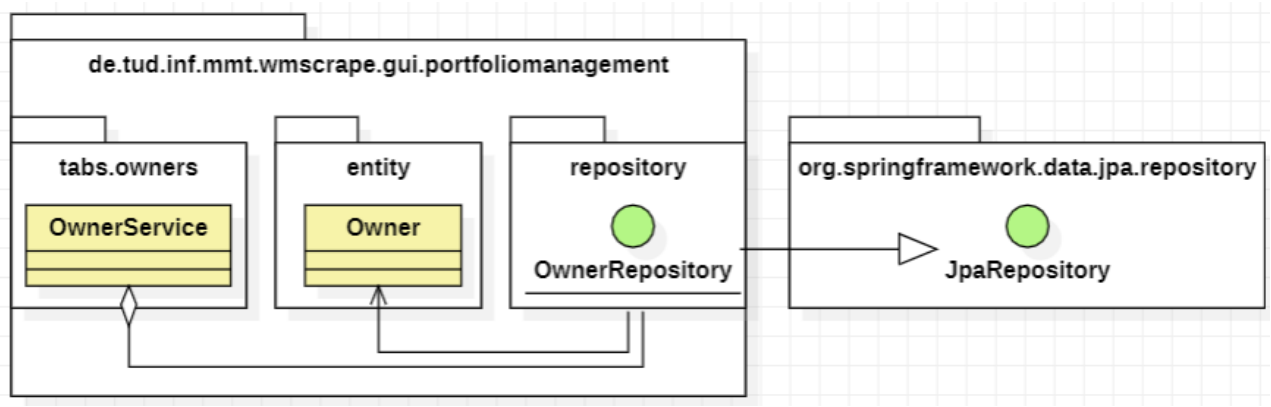

Abbildung 5.3: Konzept zur Service-Klasse

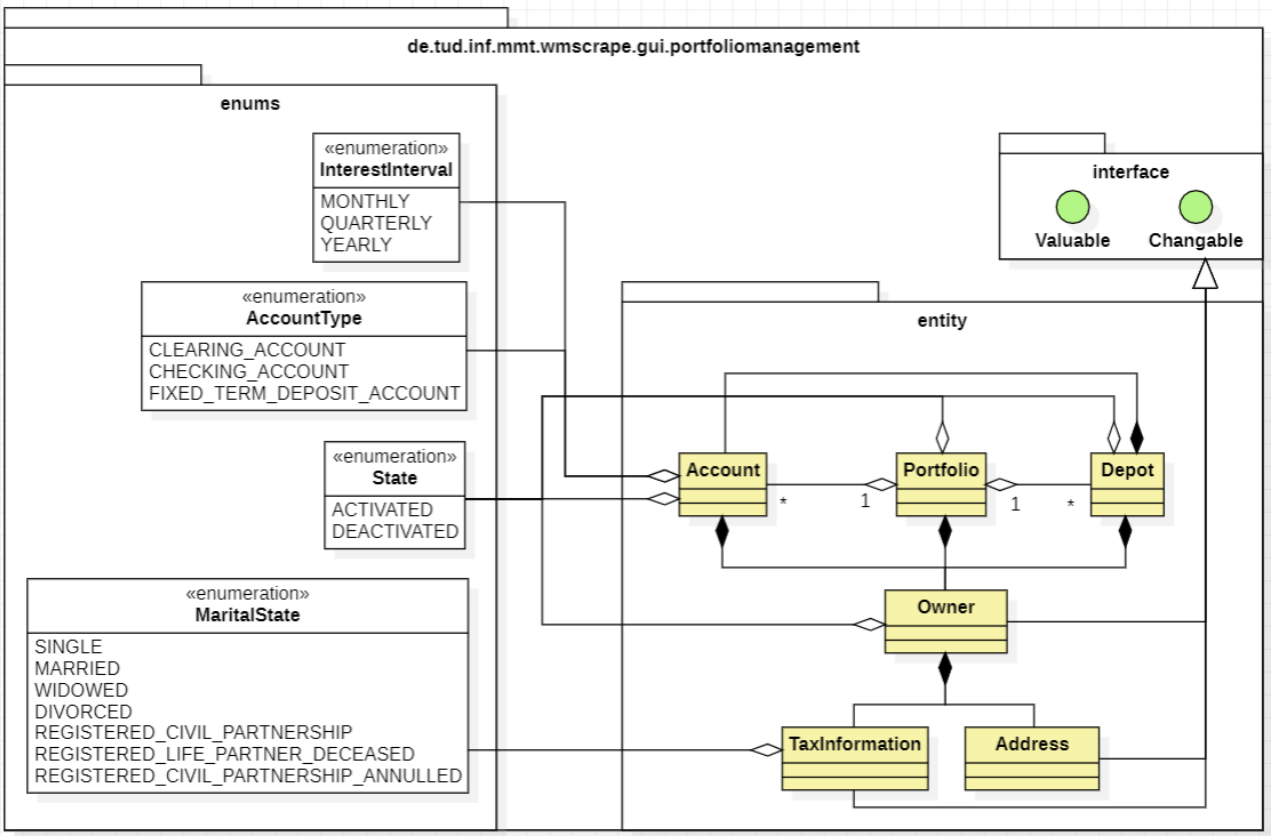

Abbildung 5.4: Konzept der benötigten Entitäten

Das *entity*-Paket aus Abbildung 5.4 enthält die zentralen Entitäts-Klassen, die als Grundlage für die Inhaber-Verwaltung sowie weitere Verwaltungskomponenten dienen.

Die *Owner*-Klasse repräsentiert einen Inhaber und umfasst folgende Attribute:

- Vor- und Nachname
- Status (aktiviert oder deaktiviert, dargestellt durch das Enum *State*)
- Anmerkungen zum Inhaber
- Erstellungs- und Deaktivierungsdatum
- Adresse (*Address*)
- Steuerinformationen (*TaxInformation*)

Die Adresse setzt sich aus folgenden Angaben zusammen:

- Land
- Postleitzahl
- Wohnort
- Straßenname und -nummer

Die Steuerinformationen umfassen folgende Daten:

- Steuernummer
- Familienstand[3] (z. B. ledig, dargestellt durch das Enum *MaritalState*)
- Steuer-, Kirchensteuer-, Kapitalertragssteuer- und Solidaritätssatz

Um tatsächliche Änderungen an den genannten Entitäten nachverfolgen zu können, implementieren diese die *Changable*-Schnittstelle. Diese Schnittstelle markiert eine Entitäts-Klasse als bearbeitbar und stellt Methoden wie *isChanged* bereit, um eine entsprechende Abfrage zu ermöglichen.

Die Adress- und Steuerinformationen wurden außerdem als eigenständige Klassen gestaltet, da sie separate Konzepte darstellen.

Ein Portfolio (*Portfolio*) wird durch folgende Merkmale beschrieben:

- Name bzw. Beschreibung

[3] https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Bevoelkerung/Haushalte-Familien/Glossar/familienstand.html

- Zugehöriger Inhaber
- Zustand
- Anlagenrichtlinie
- Erstellungs- und Deaktivierungsdatum

Jedem Portfolio können außerdem beliebig viele Depots und Konten zugewiesen werden.

Ein Konto (*Account*) weist folgende Eigenschaften auf:

- Bezeichnung bzw. Beschreibung
- Konto-Typ (*AccountType*)
- Zustand
- Kontostand mit Währung
- Zugehöriger Inhaber und Portfolio
- Anmerkungen
- Name der Bank, bei der das Konto geführt wird
- IBAN und Kontonummer
- Zinsinformationen: Zinssatz, Anzahl der Zinstage, Zinsintervall (*InterestInterval*)
- Erstellungs- und Deaktivierungsdatum

Ein Depot (*Depot*) ist durch folgende Parameter definiert:

- Name
- Zustand
- Depotbank (die Bank, bei der das Depot registriert ist)
- Zugehöriger Inhaber und Portfolio
- Verrechnungskonten
- Provisionsschema
- Anmerkungen

An dieser Stelle ist die bewusste Verwendung von Kompositionen anstelle von Aggregationen hervorzuheben. Sie macht deutlich, dass bestimmte Entitäten untrennbar miteinander verbunden sind und nicht unabhängig existieren können. Beispielsweise kann eine Adress-Entität nicht ohne einen Inhaber bestehen, ebenso wenig wie ein Konto ohne einen zugehörigen Inhaber.

Um beispielsweise den Wert eines Kontos abrufen zu können, implementieren die drei zuvor genannten Klassen die *Valuable*-Schnittstelle. Zur besseren Übersicht wurde die Beziehung in der Abbildung weggelassen. Der Name der Schnittstelle leitet sich aus der Idee ab, dass Portfolios, Konten und Depots einen bestimmten Wert besitzen. Sie enthält zudem die Methode *getValue*, die eine Instanz der Service-Klasse zur Konto-Verwaltung als Parameter entgegennimmt. Die Service-Klasse ermöglicht wiederum den Abruf des Wechselkurses, um beispielsweise den Wert eines Kontos in Euro umzurechnen.

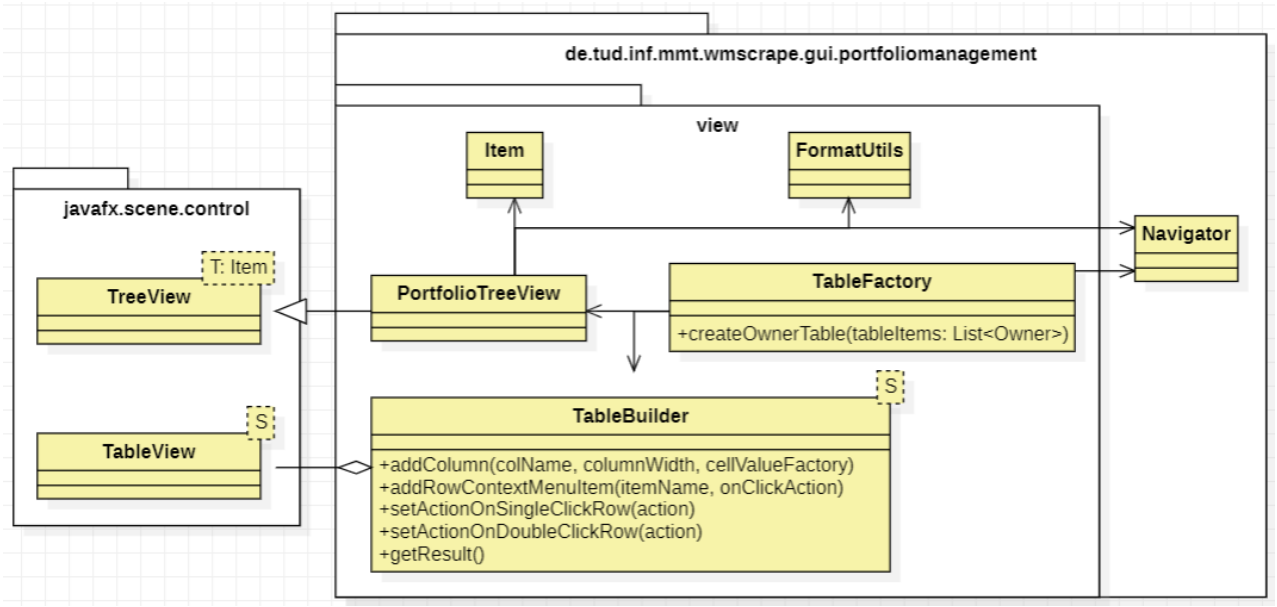

Abbildung 5.5: Konzept des Pakets „view“ im Kontext des Hauptmenüs

Das Paket *view* in Abbildung 5.5 umfasst zusätzliche Klassen zur Darstellung der Benutzeroberfläche.

Die *TableBuilder*-Klasse dient der Erstellung von Tabellen des Typs *TableView* und reduziert durch das Builder-Pattern den erforderlichen Code für die Tabellenerstellung. Dies fördert eine einfache Erweiterbarkeit und Wiederverwendbarkeit, insbesondere da Tabellen sowohl in dieser als auch in zukünftigen Arbeiten häufig verwendet werden. Gemäß dem Builder-Pattern aggregiert sie die *TableView*-Klasse, wobei der generische Typ *S* an den entsprechenden Parameter der *TableView* gebunden ist, sodass beide dasselbe Objekt repräsentieren. Die Methoden *addColumn*, *addRowContextMenuItem* und *setActionOn* ermöglichen eine flexible Erweiterung der Tabellenfunktionalität.

Mit *addColumn* können Spalten hinzugefügt werden, während *addRowContextMenuItem* das Einfügen von Kontextmenüeinträgen[4] erlaubt. Der Parameter *cellValueFactory* definiert die Darstellung des Zellinhalts in einer Spalte, während der Parameter *columnWidth* die prozentuale Breite einer Spalte in Relation zur Gesamtbreite der Tabelle festlegt. Die Methode *setActionOn* ermöglicht das Festlegen von Zeilenaktionen, beispielsweise das Anzeigen der Baumansicht per einfachem Klick oder das „Öffnen eines Inhabers“ durch einen Doppelklick. Abschließend kann mit der Methode *getResult* die konfigurierte Tabelle abgerufen werden.

Die *TableFactory*-Klasse basiert auf dem Abstract-Factory-Pattern (auch bekannt als „Factory Class“) und stellt Fabrikmethoden für spezifische Tabellen bereit, wie beispielsweise die Methode *createOwnerTable* in Abbildung 5.1. Dadurch wird das „Single Responsibility Principle“[5] (Prinzip der eindeutigen Verantwortlichkeit) gewahrt, da die Tabellenerzeugung an einer zentralen Stelle gekapselt wird. Die Erstellung der Tabellen erfolgt dabei mit Hilfe der *TableBuilder*-Klasse.

Zur Visualisierung der Baumansicht der Portfolios wurde die *PortfolioTreeView*-Klasse entwickelt. Diese erbt von der *TreeView*-Klasse und bindet den Parameter *T* an den Typ *Item*, der die anzuzeigenden Objekte repräsentiert. Zudem nutzt sie die *FormatUtils*-Klasse, um primitive Datentypen wie *float* (Fließkommazahlen mit Punkt als Dezimaltrennzeichen) für die Darstellung von Portfolio-Werten im Kommaformat umzuwandeln.

Die Initialisierung sowohl der *PortfolioTreeView*- als auch der *TableBuilder*-Klasse erfolgt über den jeweiligen Konstruktor, der die entsprechenden Einträge vom Typ *Portfolio* bzw. *S* erhält.

Die Klassen *PortfolioTreeView* und *TableFactory* greifen auf die *Navigator*-Klasse zurück, um zu den jeweiligen Übersichten zu navigieren (z. B. zur Inhaber-Übersicht). Die *Navigator*-Klasse stellt dafür geeignete Methoden bereit, sodass die Navigation zentral organisiert ist.

[4] Ein Kontextmenü ist ein Menü, das durch einen Rechtsklick auf ein Objekt geöffnet wird.
[5] https://de.wikipedia.org/wiki/Single-Responsibility-Prinzip

5.1.2 Erstellen eines Inhabers

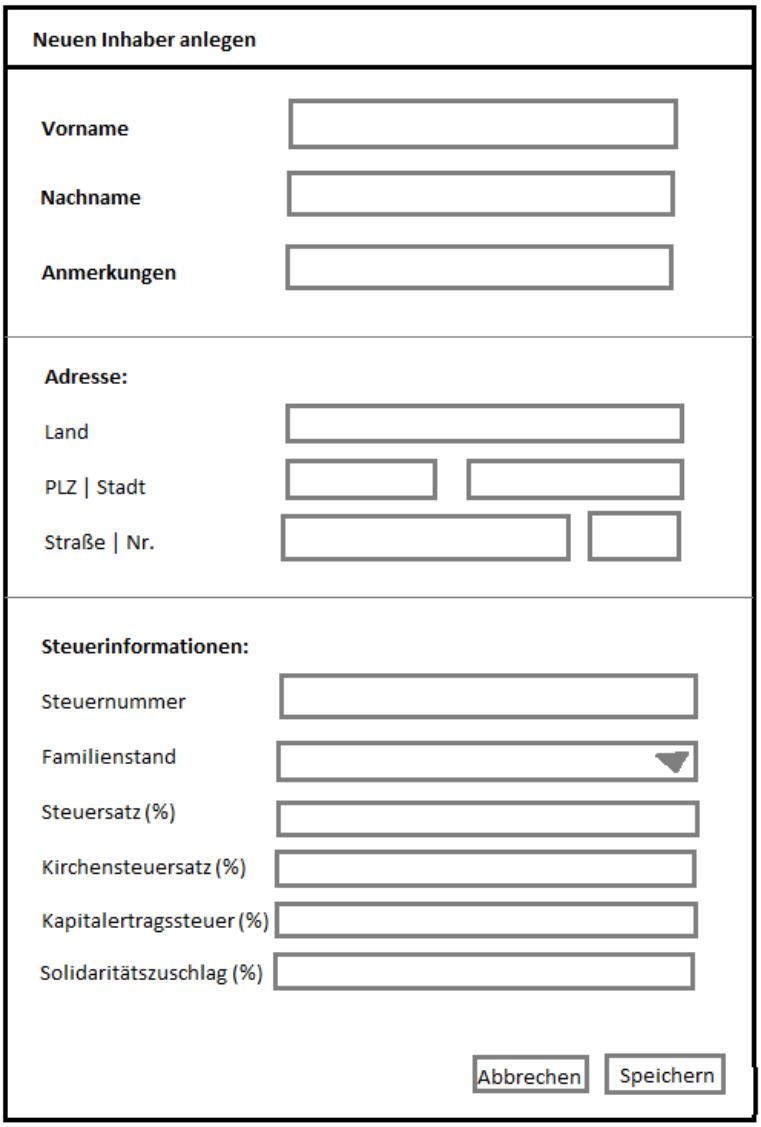

Abbildung 5.6: Mockup für den Dialog zum Erstellen eines neuen Inhabers

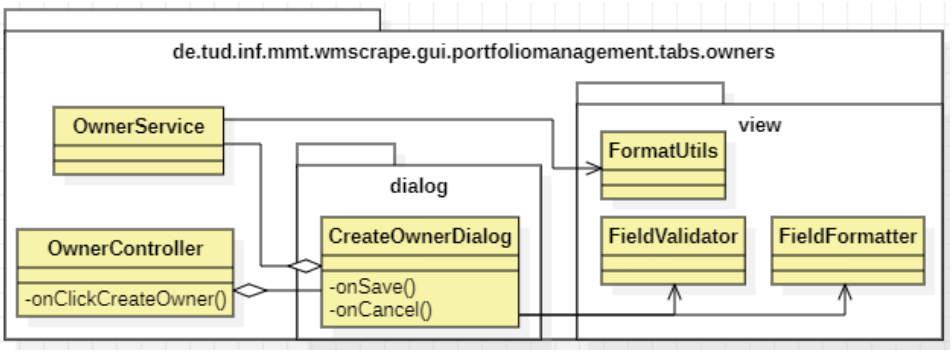

Abbildung 5.7: Konzept zur Realisierung eines Dialogs zum Erstellen von Inhabern

Der *OwnerController* verwaltet den Dialog über die zugehörige Controller-Klasse (*CreateOwnerController*). Beim Laden des Dialogs (siehe Abbildung 5.6) mittels FXML wird der Ansicht dem Controller zugewiesen, sodass er auf die Nutzereingaben im Dialog reagieren kann. Darüber hinaus erfolgt das Laden bzw. die Anzeige des Dialogs in der *onClickCreateOwner*-Methode des Controllers für das Hauptmenü. Diese Methode wird von FXML ausgelöst, wenn ein Klick auf den Button zum Erstellen eines Inhabers registriert wurde.

Der *CreateOwnerController* nutzt die Service-Klasse, um einen neuen Inhaber zu speichern. Die Methoden *onSave* und *onCancel*, die in der Controller-Klasse definiert sind, ermöglichen es, durch Klicken des entsprechenden Buttons entweder einen neuen Inhaber zu erstellen oder den Vorgang abzubrechen und den Dialog zu schließen. Vor dem Speichern wird sichergestellt, dass wesentliche Eingaben wie der Name nicht leer sind. Hierfür stellt die *FieldValidator*-Klasse eine passende Methode bereit. Die Validierung der Steuersatz-Eingaben ist hingegen nicht erforderlich, da die Eingabefelder bereits bei der Initialisierung des Dialogs mithilfe der *FieldFormatter*-Klasse eingeschränkt werden, sodass ausschließlich Dezimalzahlen im Bereich von 0,00 bis 100,00 eingegeben werden können. Mithilfe der *FormatUtils*-Klasse sollen diese Eingaben jedoch so „geparst“ werden, dass Dezimalzahlen im Kommaformat in den primitiven Typ *float* umgewandelt werden. Nach erfolgreichem Speichern wird die *open*-Methode der *Openable*-Schnittstelle aufgerufen, um die Daten im Hauptmenü zu aktualisieren.

Es ist zudem anzumerken, dass für die Eingaben der Steuernummer sowie der Adressdaten keine Einschränkungen vorgesehen sind. Der Grund dafür liegt in der erheblichen Variabilität dieser Daten je nach Land. So besteht die Postleitzahl in Deutschland ausschließlich aus Zahlen, während sie in Lettland auch Buchstaben und Sonderzeichen enthalten kann (z. B. „LV-1234“[6]).

[6] https://de.wikipedia.org/wiki/Liste_der_Postleitsysteme

5.1.3 Inhaber-Übersicht

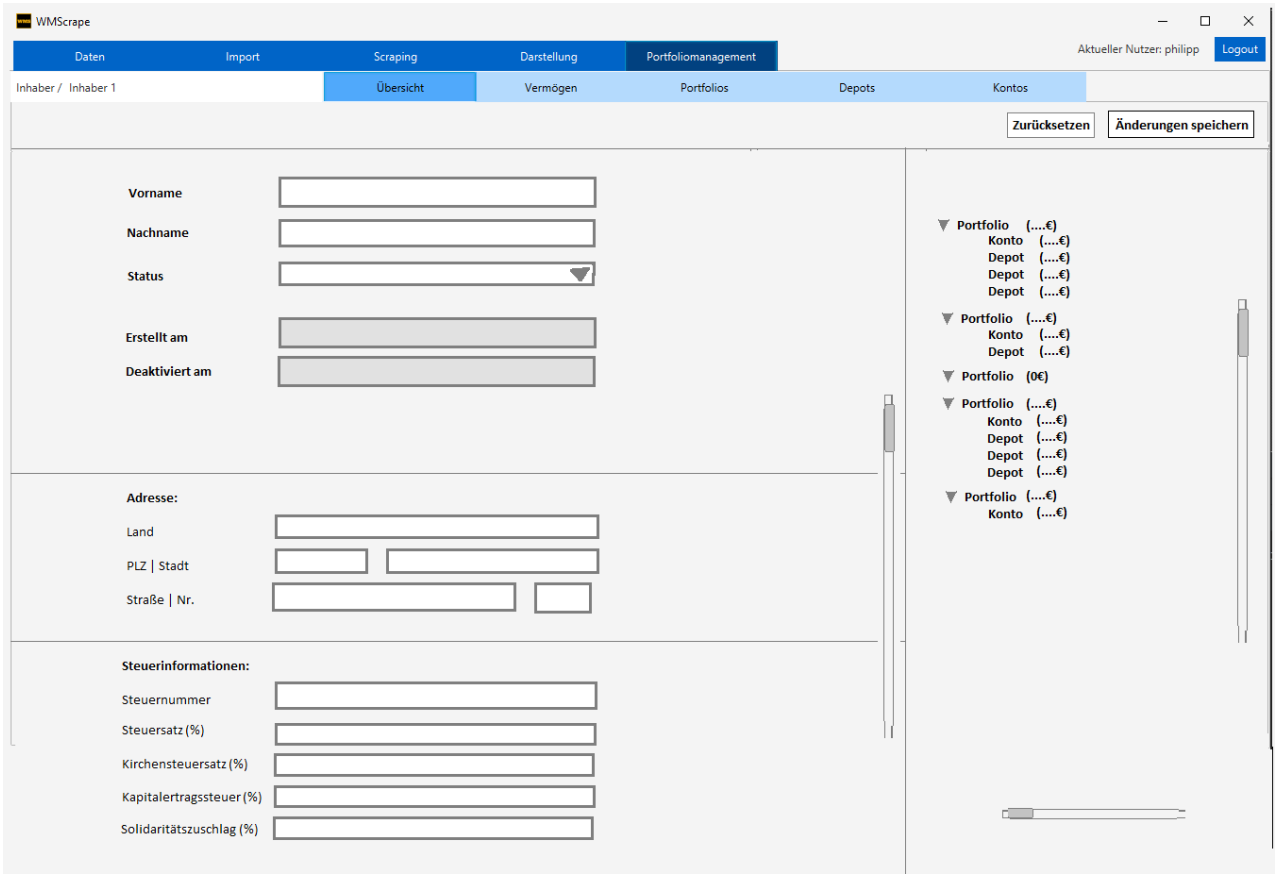

Abbildung 5.8: Mockup für die Übersicht eines Inhabers

In der Übersicht aus Abbildung 5.8 werden alle Parameter eines Inhabers sowie dessen zugehörige Portfolios mit den darin enthaltenen Konten und Depots angezeigt. Zudem hat der Benutzer die Möglichkeit, die Parameter zu bearbeiten. Mit dem Button „Zurücksetzen“ können alle vorgenommenen Änderungen rückgängig gemacht werden.

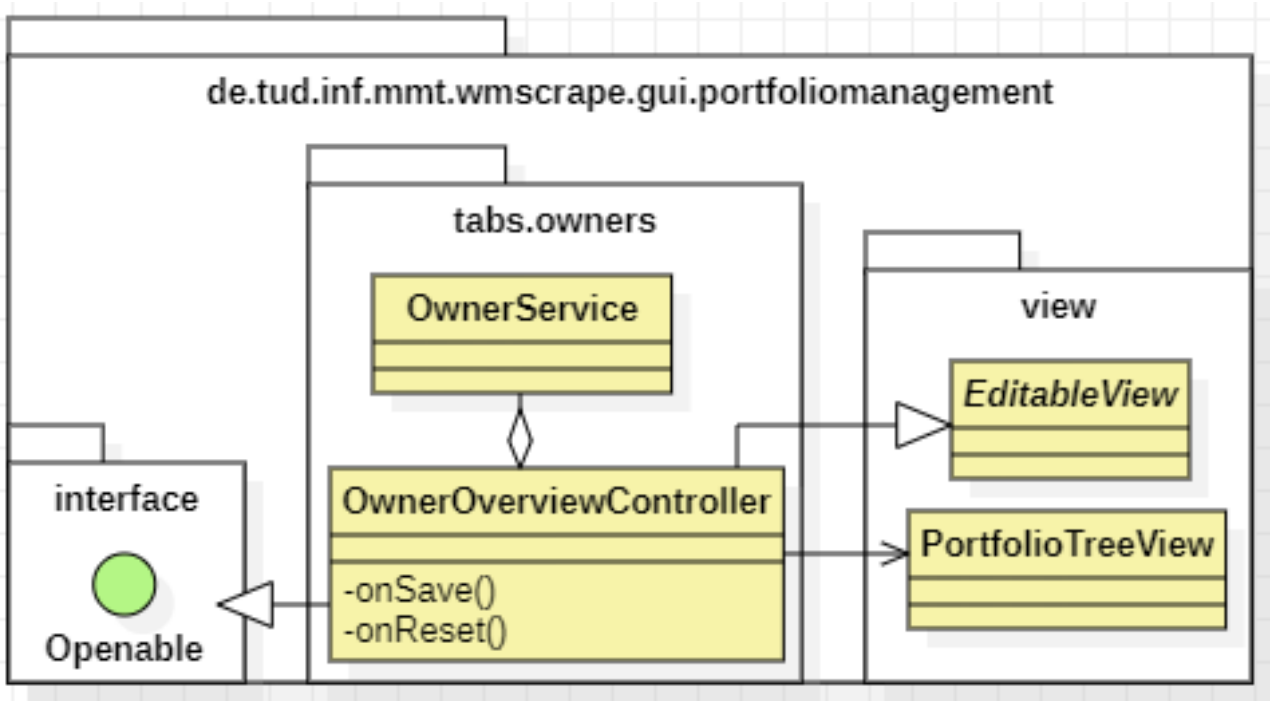

Abbildung 5.9: Konzept zur Darstellung der Inhaber-spezifischen Parameter

Der *OwnerOverviewController* ist für die Darstellung der Daten und die Verarbeitung von Eingaben innerhalb dieser Ansicht verantwortlich. Er implementiert die *Openable*-Schnittstelle, um beim Öffnen des Tabs („Übersicht“) die Daten aus der Tab-Instanz zu holen und mittels der Service-Klasse (*OwnerService*) aktuell zu halten. Zudem initialisiert er die Superklasse *EditableView*, welche für Controller-Klassen gedacht ist, die eine bearbeitbare Ansicht bereitstellen. Dies ermöglicht eine Überwachung aller wesentlichen Navigationsmöglichkeiten innerhalb des Portfoliomanagements, um zu erkennen, wann auf die Navigationsmöglichkeiten zugegriffen (angeklickt) wurde und informiert den Benutzer gegebenfalls über ungespeicherte Änderungen. Die Navigation umfasst sowohl Breadcrumbs als auch die Tableiste mit Tabs wie „Daten“, „Import“ usw., sowie die übergeordnete Tableiste des Portfoliomanagements mit Kategorien wie „Portfolios“ und „Depots“.

Analog zum Hauptmenü nutzt der Controller die *PortfolioTreeView*, um die Portfolios eines Inhabers anzuzeigen.

Die *onSave*-Methode arbeitet ähnlich der gleichnamigen Methode im Dialog (*CreateOwnerDialog*). Jedoch werden nun zusätzlich die Breadcrumbs aktualisiert.

Die *onReset*-Methode hingegen wird durch einen Klick auf den Button „Zurücksetzen“ ausgelöst und setzt nicht gespeicherte Änderungen durch den Aufruf der *restore*-Methode aus der *Changable*-Schnittstelle auf den vorherigen Zustand zurück.

5.1.4 Inhaber-Portfolios

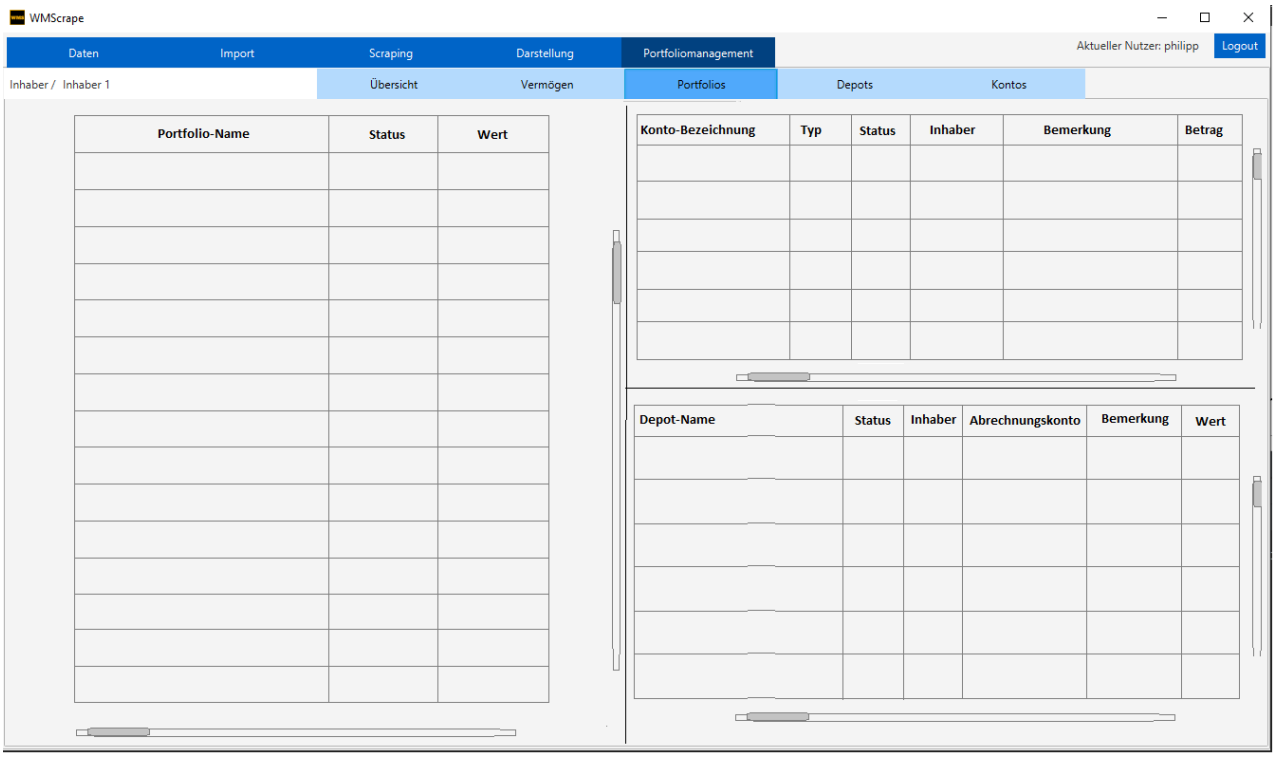

Abbildung 5.10: Mockup für die detaillierte Ansicht der dem Inhaber zugehörigen Portfolios

Beim Klick auf das Menü „Portfolios“ erhält der Benutzer eine detaillierte Übersicht aller dem Inhaber zugeordneten Portfolios (siehe Abbildung 5.10). Wählt der Benutzer ein Portfolio aus, werden die entsprechenden Konten und Depots in einer Tabelle rechts daneben dargestellt. Zudem führt ein Klick auf ein Portfolio, Konto oder Depot zur jeweiligen Übersicht. Über ein Kontextmenü kann zusätzlich die Vermögensübersicht des Portfolios aufgerufen werden – dabei öffnet sich der Menüpunkt *Vermögen* als modales Fenster und zeigt das Vermögen im Kontext des ausgewählten Portfolios an.

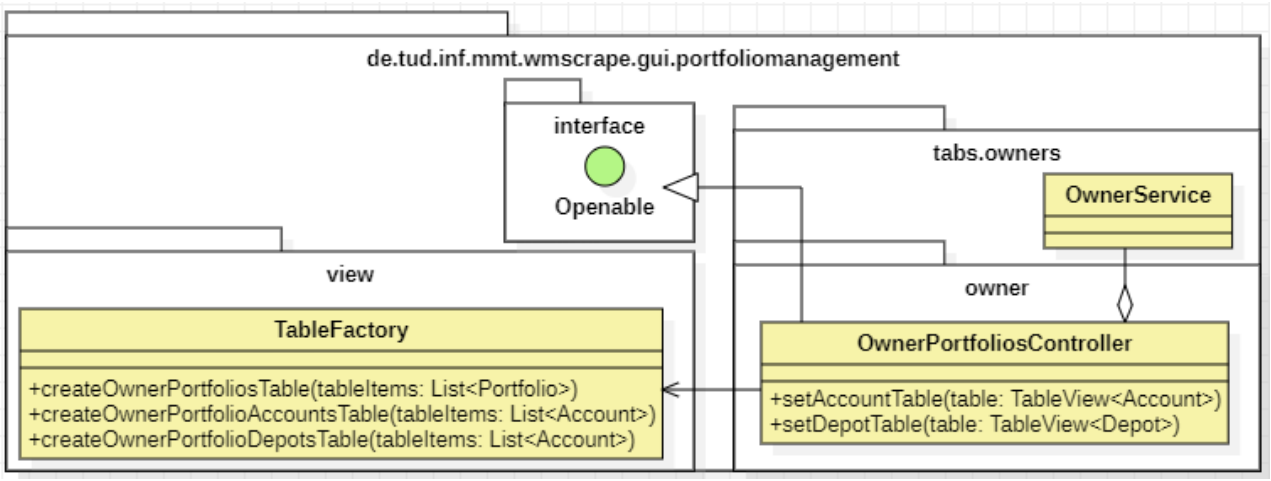

Abbildung 5.11: Konzept zur Darstellung der Inhaber-spezifischen Portfolios

Der *OwnerPortfoliosController* ist für die Darstellung der Daten verantwortlich. Das Laden erfolgt analog zum vorherigen Abschnitt. Er verwendet die *TableFactory*, um mittels der Methode *createOwnerPortfoliosTable* die Tabelle zur Darstellung der dem Inhaber zugeordneten Portfolios zu generieren. Wird ein Portfolio ausgewählt, werden zunächst die Tabellen der zugehörigen Konten und Depots innerhalb dieser Methode (*createOwnerPortfoliosTable*) mittels der entsprechenden Methode *createOwnerPortfolioAccountsTable* bzw. *createOwnerPortfolioDepotsTable* erzeugt und anschließend über die Methode *setAccountTable* bzw. *setDepotTable* angezeigt.

5.1.5 Inhaber-Konten

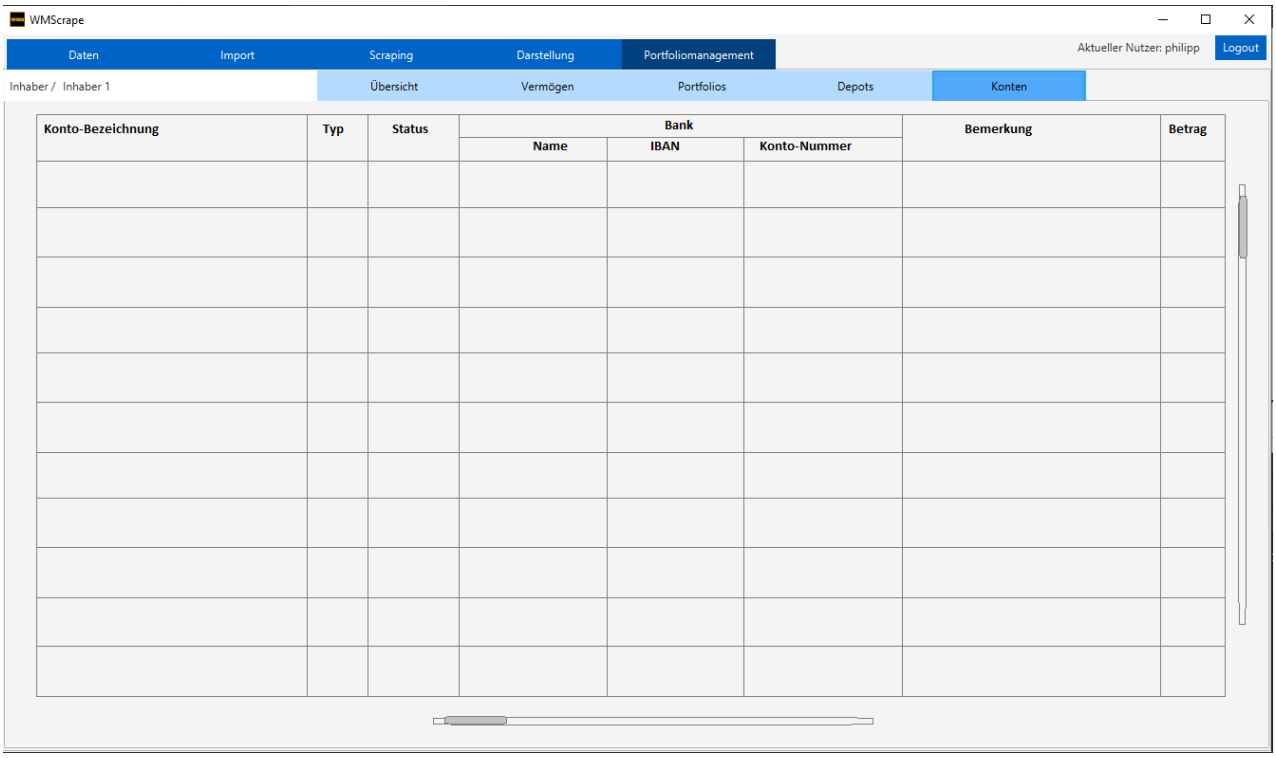

Abbildung 5.12: Mockup für die detaillierte Ansicht der dem Inhaber zugehörigen Konten

In der Konten-Ansicht aus Abbildung 5.12 erhält der Benutzer einen detaillierten Überblick über alle dem Inhaber zugeordneten Konten. Auch hier soll bei einem Doppelklick auf ein Konto zur dessen Übersicht weitergeleitet werden.

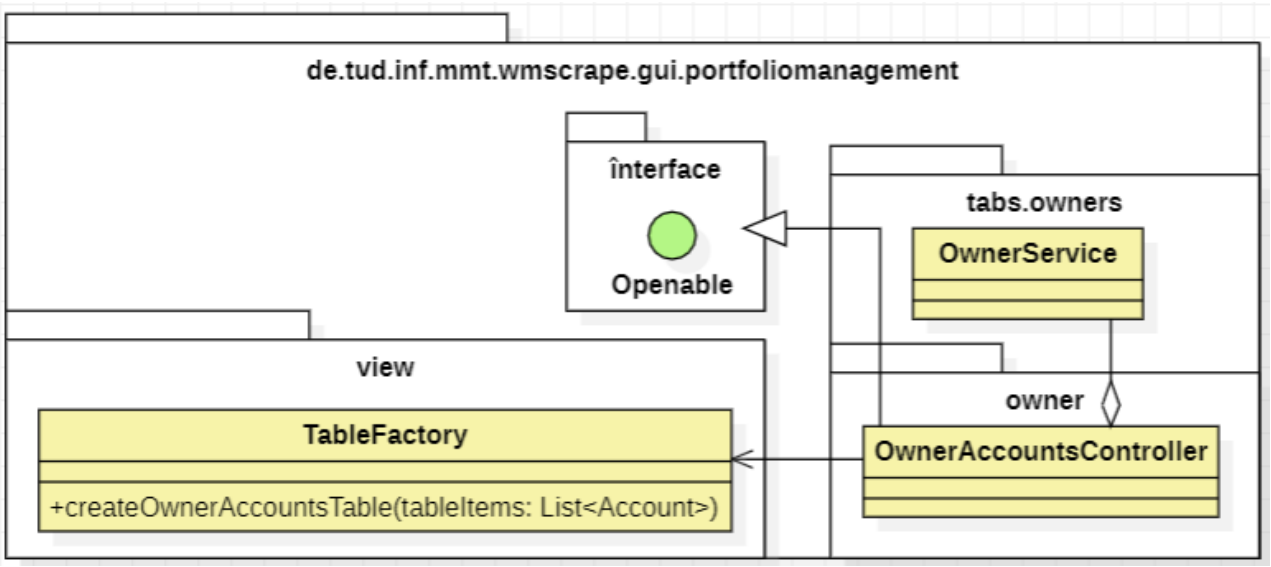

Abbildung 5.13: Konzept zur Darstellung der inhaber-spezifischen Konten

Für die Darstellung dieser Ansicht ist der *OwnerAccountsController* zuständig. Nachdem die Daten geladen wurden, wird mit Hilfe der *createOwnerAccountsTable*-Methode die entsprechende Tabelle erzeugt und anschließend angezeigt. Für die Darstellung der Spalte mit den Unterspalten kommt eine Methode der *TableBuilder*-Klasse namens *addNestedColumn* zum Einsatz.

5.1.6 Inhaber-Depots

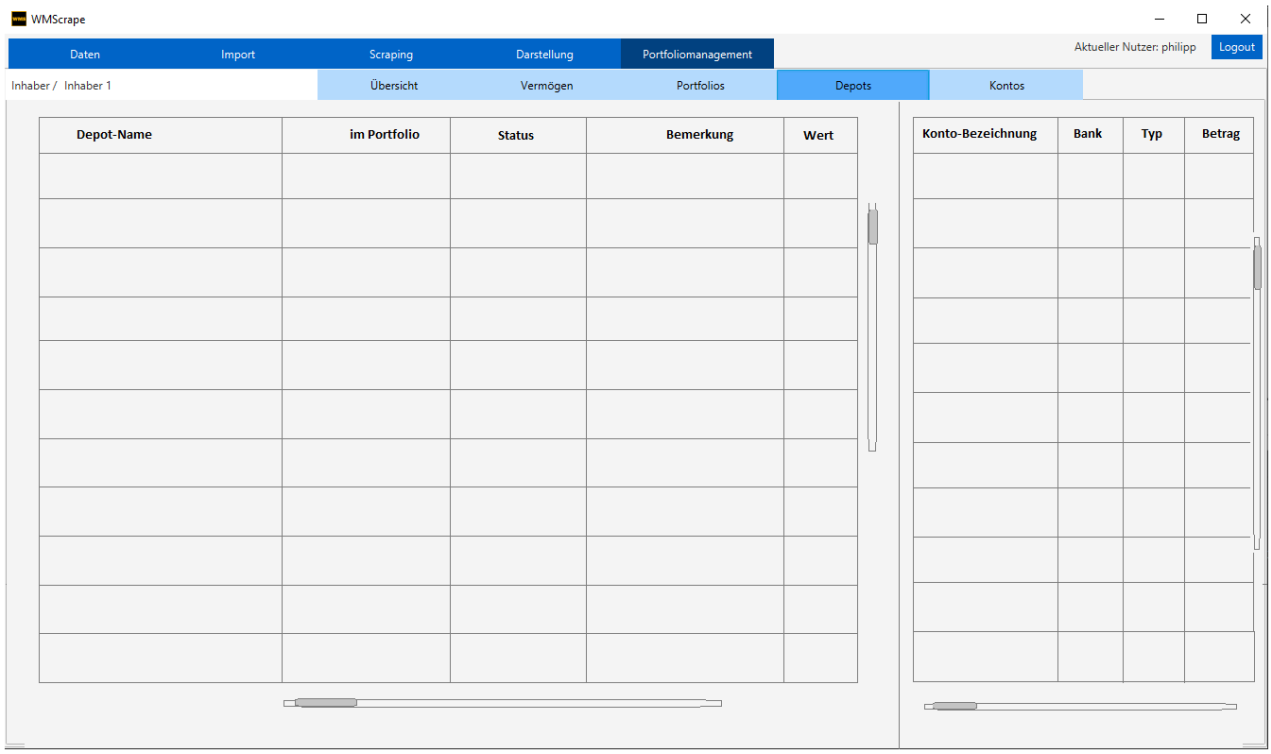

Abbildung 5.14: Mockup für die Ansicht der dem Inhaber zugehörigen Depots

Im Menü „Depots“ aus Abbildung 5.14 werden dem Benutzer alle dem Inhaber zugeordneten Depots angezeigt. Bei Auswahl eines Depots sollen zudem seine zugeordneten Verrechnungskonten dargestellt werden. Wie gehabt soll zudem bei Doppelklick auf ein Depot bzw. Konto zur dessen Übersicht weitergeleitet werden.

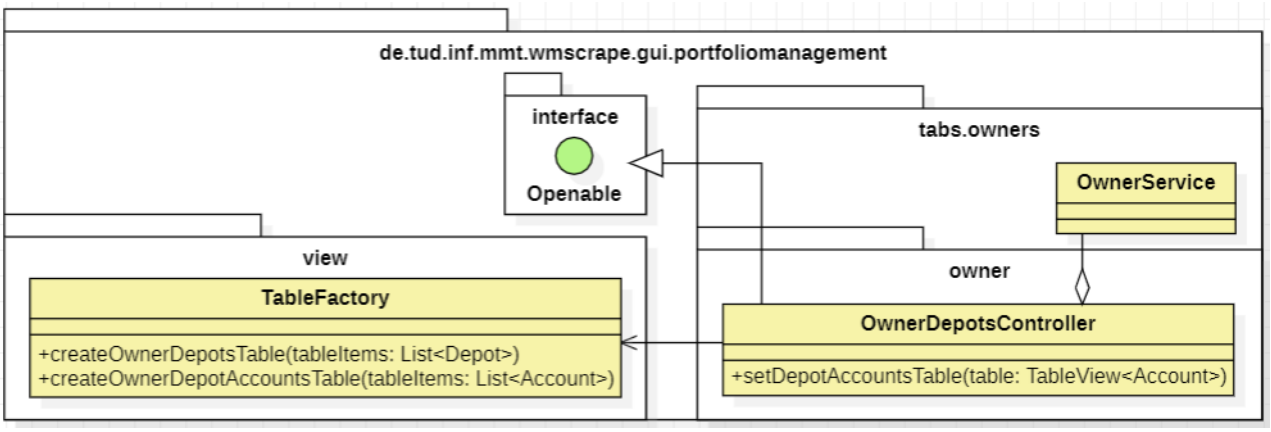

Abbildung 5.15: Konzept zur Darstellung der Inhaber-spezifischen Depots

Zur Darstellung der Daten dient der *OwnerDepotsController*. Die *setDepotAccountsTable*-Methode wird aufgerufen, wenn ein Depot ausgewählt wird, und zeigt die dem Depot zugeordneten Verrechnungskonten an. Zur Generierung der Tabellen wird ebenfalls die *TableFactory* verwendet: Während mit *createOwnerDepotsTable* die Depots tabellarisch aufgelistet werden, wird mittels *createOwnerDepotsAccountTable* die Tabelle erzeugt, welche die Verrechnungskonten des ausgewählten Depots auflistet.

5.2 Portfolio-Verwaltung

Die Portfolio-Verwaltung bietet dem Benutzer die Möglichkeit, neue Portfolios zu erstellen und eine Übersicht der bereits registrierten Portfolios anzuzeigen. Zudem steht beim „Öffnen eines Portfolios“ eine detaillierte Ansicht zur Verfügung, in der dessen Parameter eingesehen werden können.

5.2.1 Hauptmenü

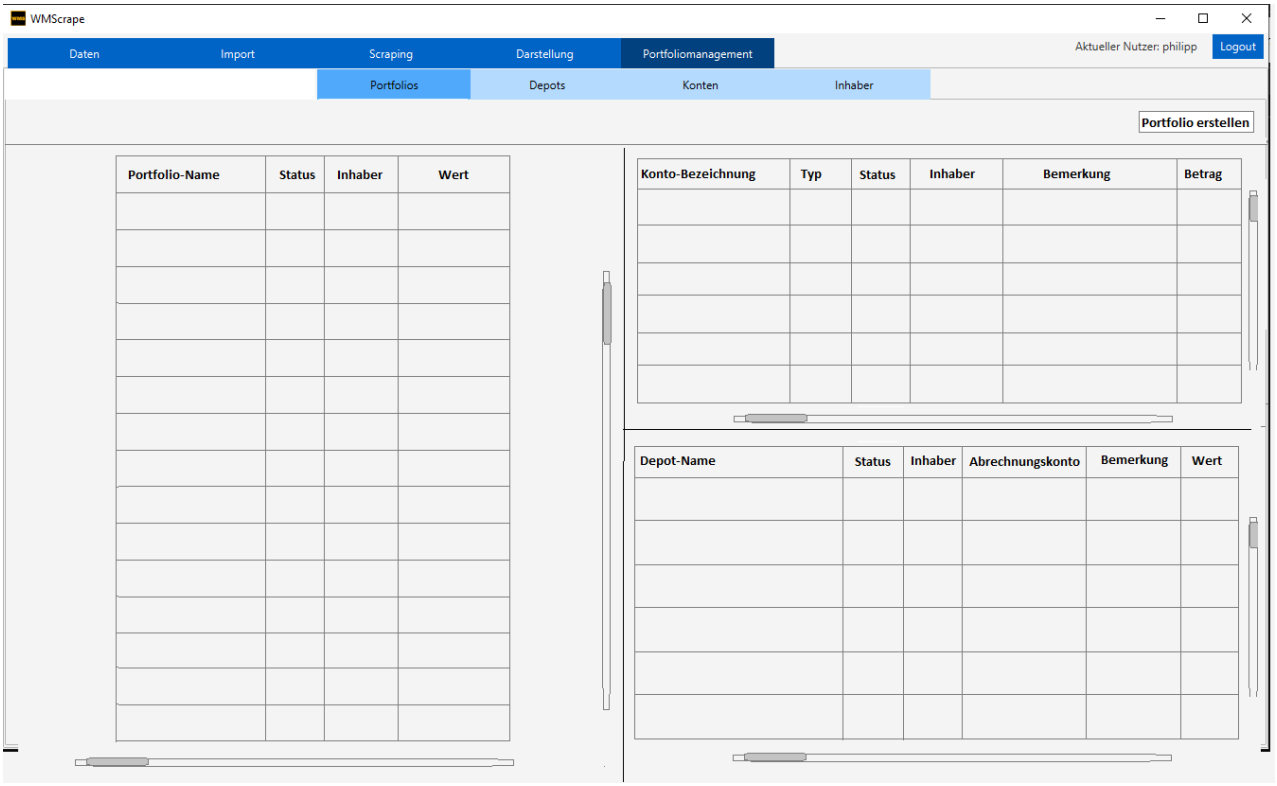

Abbildung 5.16: Mockup des Hauptmenüs der Portfolio-Verwaltung

In der Portfolio-Verwaltung wird, ähnlich wie in der Inhaber-Verwaltung, im Hauptmenü aus Abbildung 5.16 eine Übersicht aller Portfolios angezeigt, einschließlich der enthaltenen Konten und Depots. Zudem besteht die Möglichkeit, neue Portfolios zu erstellen.

Wie in der Inhaber-Verwaltung bietet auch hier ein Kontextmenü die Option, Portfolios direkt zu löschen oder ihren Status zu ändern. Zudem ermöglicht – analog zum Menüpunkt *Portfolios* in der inhaber-spezifischen Ansicht – das Kontextmenü eine direkte Darstellung der Vermögen in einem modalen Fenster.

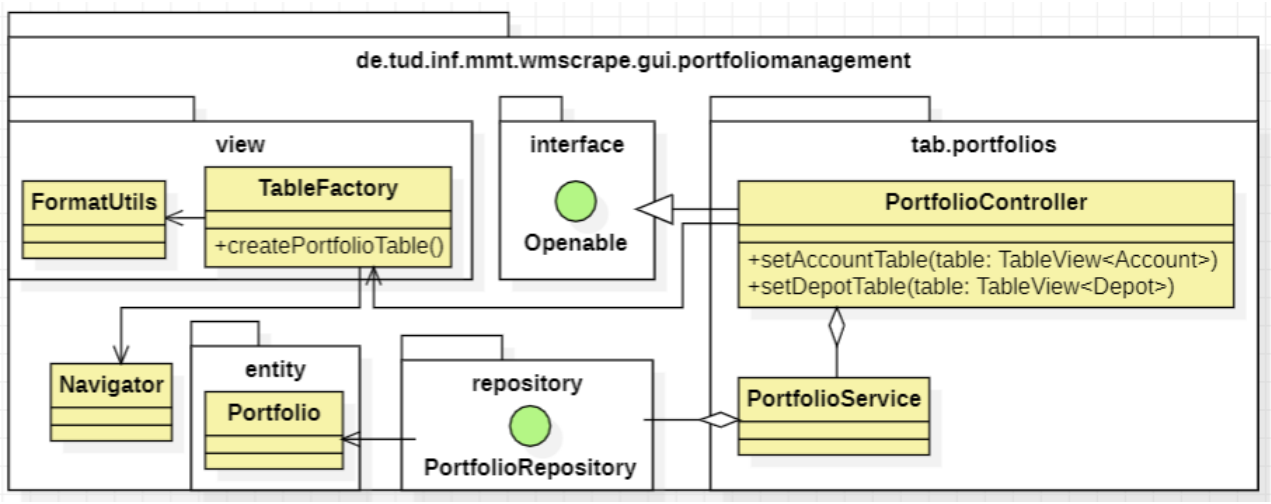

Abbildung 5.17: UML-Diagramm für das Hauptmenü

Die Struktur des Pakets *tab.portfolios* entspricht im Wesentlichen der des bereits vorgestellten Pakets *tab.owners*. Der *PortfolioController* ist für die Darstellung der Daten im Hauptmenü verantwortlich. Zusätzlich implementiert die Controller-Klasse analog zur Inhaber-Verwaltung die *Openable*-Schnittstelle.

Die *createPortfolioTable*-Methode wurde in der *TableFactory* entwickelt und ist für die tabellarische Darstellung der Portfolios im Hauptmenü zuständig. Die Methoden *setAccountTable* und *setDepotTable* stammen aus der Inhaber-Verwaltung und werden aufgerufen, sobald ein Portfolio in der Tabelle ausgewählt wird. Zusätzlich soll die eingeführte *Navigator*-Klasse für die Navigation wiederverwendet werden. Die Methode *createPortfolioTable* soll daher auf *navigateToPortfolio* zurückgreifen, um gezielt zu einem Portfolio zu navigieren.

Für die Kommadarstellung wird auch hier die *FormatUtils*-Klasse verwendet.

5.2.2 Erstellen eines Portfolios

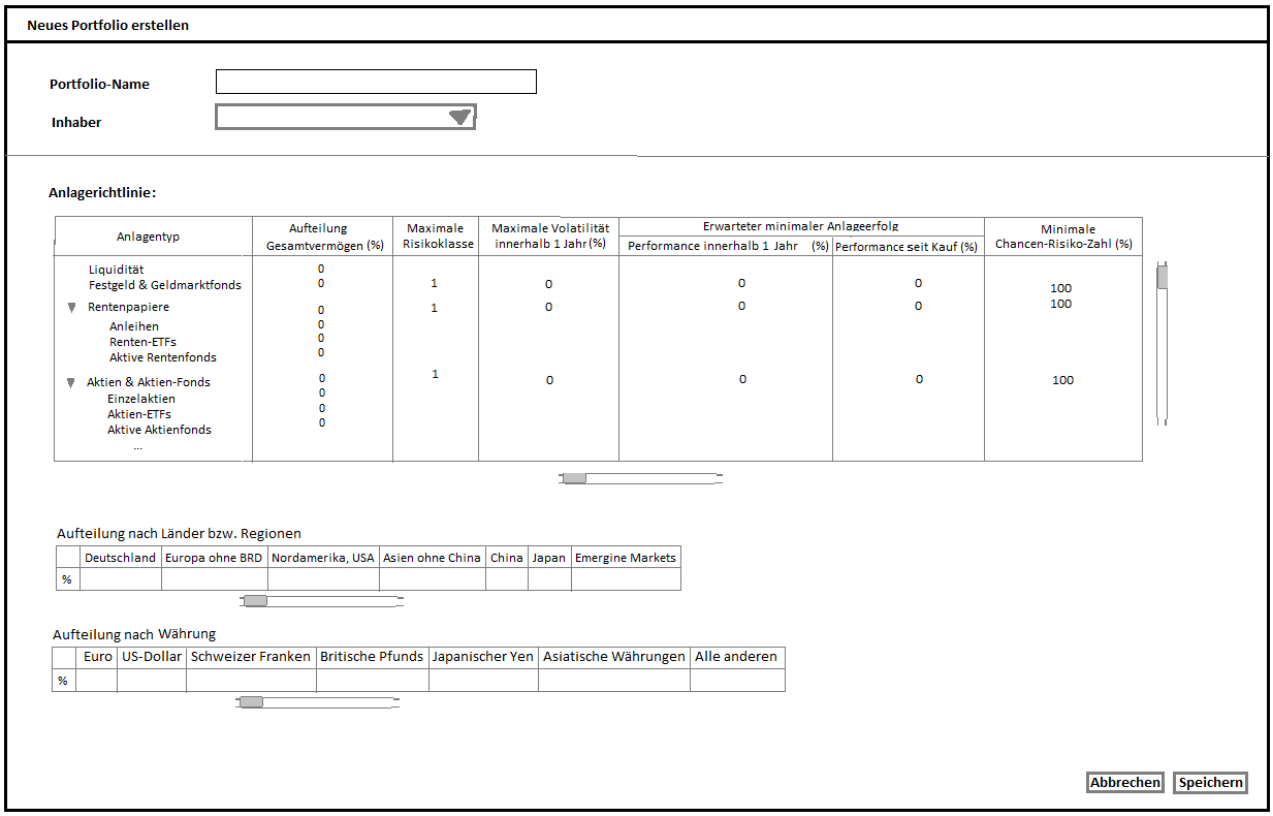

Abbildung 5.18: Mockup für den Dialog zum Erstellen eines Portfolios

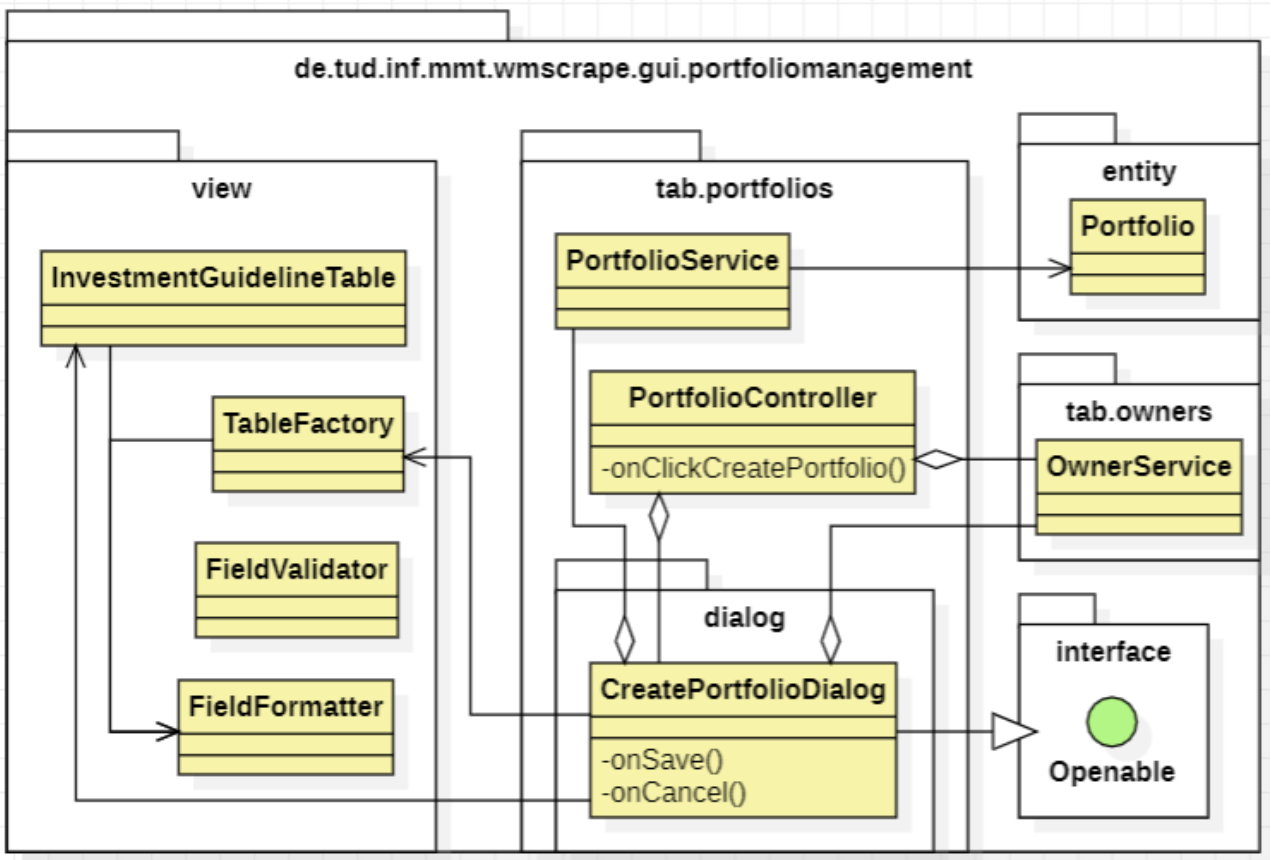

Abbildung 5.19: UML-Diagramm für den Dialog

Das Konzept des Dialogs, wie in Abbildung 5.19 dargestellt, entspricht im Wesentlichen dem der Inhaber-Verwaltung (vgl. Abbildung 5.7).

Da ein Portfolio nur erstellt werden kann, wenn bereits registrierte Inhaber vorhanden sind, nutzt die *onClickCreatePortfolio*-Methode die Service-Klasse der Inhaber-Verwaltung (*OwnerService*), um zu prüfen, ob Inhaber existieren, denen das Portfolio zugeordnet werden kann. Falls keine Inhaber vorhanden sind, ist die Erstellung eines Portfolios nicht möglich, und der Benutzer wird entsprechend informiert. Andernfalls wird der Dialog angezeigt, sodass nun der Controller *CreatePortfolioDialog* die Benutzerinteraktionen behandelt.

Die Implementierung der *Openable*-Schnittstelle ist erforderlich, da der Dialog die aktuellen Inhaber als dynamische Daten bereitstellen muss. Zudem werden hier die Einträge (*Entry*, siehe Abbildung 5.20) der Anlagenrichtlinie initialisiert, die für die entsprechende Tabelle benötigt werden.

Zur Erstellung der Tabellen für die Aufteilung nach Ländern und Währungen kommt die *TableFactory* zum Einsatz. Diese greift auf eine Methode der bestehenden *FieldFormatter*-Klasse zurück, um Eingaben auf einen Wertebereich von 0,00 bis 100,00 zu beschränken. Analog dazu wird die

InvestmentGuidelineTable-Klasse zur Erzeugung der Anlagenrichtlinien-Tabelle verwendet (siehe erste Tabelle in Abbildung 5.18). Auch sie nutzt die *FieldFormatter*-Klasse, um beispielsweise die Eingaben zur Vermögensaufteilung auf einen Bereich zwischen 0 und 100 zu begrenzen.

Die Validierung der Eingaben vor dem Speichern erfolgt über eine Methode der Service-Klasse, die mithilfe der *FieldValidator*-Klasse sowohl auf leere Eingaben prüft als auch sicherstellt, dass die Summen der Aufteilungen korrekt sind – sowohl für die übergeordneten als auch für die untergeordneten Einträge. Falls Unstimmigkeiten auftreten, wird der Benutzer entsprechend benachrichtigt

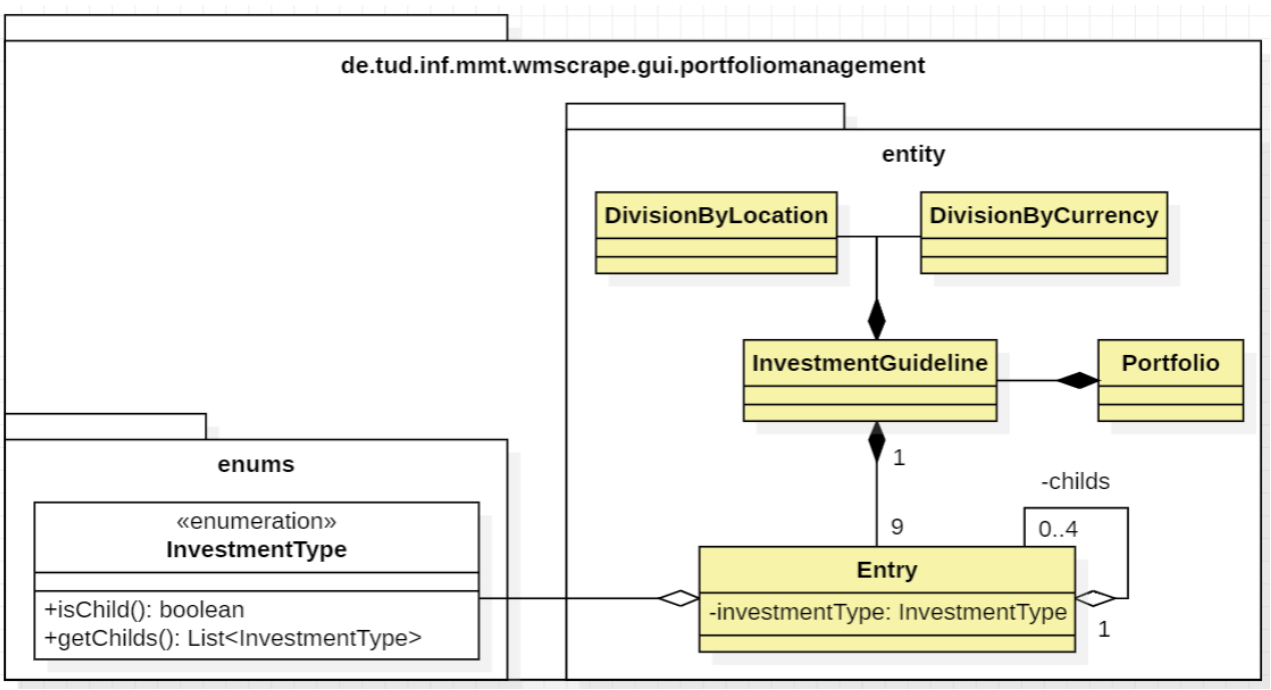

Abbildung 5.20: Detaillierte Ansicht der Portfolio-Klasse

Die *InvestmentGuideline*-Klasse umfasst Datensätze zur Aufteilung des Gesamtvermögens sowohl nach Ländern bzw. Regionen als auch nach Währungen. Daraus ergibt sich eine Aggregation der Klassen *DivisionByLocation* und *DivisionByCurrency*, die diese Entitäten in der Datenbank repräsentieren. Die *InvestmentGuideline*-Klasse besteht zudem aus 9 Einträgen (für „Liquidität“, „Rentenpapiere“, „Aktien & Aktien-Fonds“, usw.), die jeweils durch die Klasse *Entry* dargestellt werden. Diese Einträge repräsentieren Datensätze (Aufteilung des Gesamtvermögens, Chancen-Risiko-Zahl, usw.) eines Anlagetyps (*InvestmentType*) in der Tabelle der Anlagenrichtlinie. Daher besteht eine Aggregationsbeziehung zwischen *Entry* und dem Enum *InvestmentType*. Durch das Attribut *childs* soll zudem die hierarchische Struktur der Einträge, analog zur Tabelle aus Abbildung 5.18, ermöglicht werden, indem übergeordneten Einträgen wie „Aktien & Aktienfonds“ un-

tergeordneten Einträgen wie „Einzelaktien“ hinzugefügt werden. Daher kann sich die Klasse *Entry* selbst mehrfach aggregieren.

Das Enum *InvestmentType* definiert Literale, denen im Konstruktor zwei Parameter übergeben werden sollen. Zum einen ein booleschen Wert, der angibt, ob es sich um ein übergeordnetes oder ein untergeordnetes Literal handelt. Der Wert des Parameters kann über die Methode *isChild* abgerufen werden. Zum anderen sollen bei einem übergeordneten Literal mehrere untergeordnete übergeben werden, die mittels der *getChilds*-Methode abgerufen werden können. Diese Struktur ermöglicht es, nachfolgend die *InvestmentGuideline* mit den erforderlichen Einträgen zu initialisieren.

Nachfolgend werden relevante Daten der Anlagenrichtlinie erläutert:

- **Aufteilung des Gesamtvermögens:** Die Werte folgen einem Top-down-Prinzip. Ein Beispiel: Wird für den Anlagentyp „Rentenpapiere“ ein Anteil von 25% und für „Anleihen“ ein Anteil von 100% angegeben, bedeutet dies, dass 25% des Gesamtvermögens in Rentenpapieren angelegt werden und innerhalb dieser 25% ausschließlich Anleihen enthalten sind.
- **Maximale Risikoklasse:** Gibt die maximal zulässige Risikoklasse der Wertanlagen an und orientiert sich an der Skala von Stiftung Warentest[7], die Werte von 1 bis 12 umfasst. Während 1 für sichere Anlagen steht, kennzeichnet 12 hochriskante Investments.
- **Maximale Volatilität:** Gibt an, wie stark der Wert einer Anlage maximal fallen darf. Der Wertebereich liegt zwischen 0% (kein Wertverlust) und 100% (Totalverlust).
- **Performance-Indikatoren:**
  - **Performance innerhalb eines Jahres:** Der erwartete Mindestwertzuwachs der Anlage pro Jahr.
  - **Performance seit Kauf:** Der erwartete Mindestwertzuwachs seit dem Erwerb der Anlage.
- **Minimale Chancen-Risiko-Zahl:** Beschreibt das Verhältnis zwischen Anlage und Benchmark, d.h. es beschreibt das Mindestmaß, um das eine Anlage besser oder schlechter im Verhältnis zum Benchmark abschneiden soll. Der Standardwert „100%“, drückt aus, dass die Anlage gleich gut wie der Benchmark performen soll.

[7] https://www.test.de/Bewertung-von-Fonds-und-ETF-Risikoklasse-hilft-bei-der-Fondswahl-6139024-0/

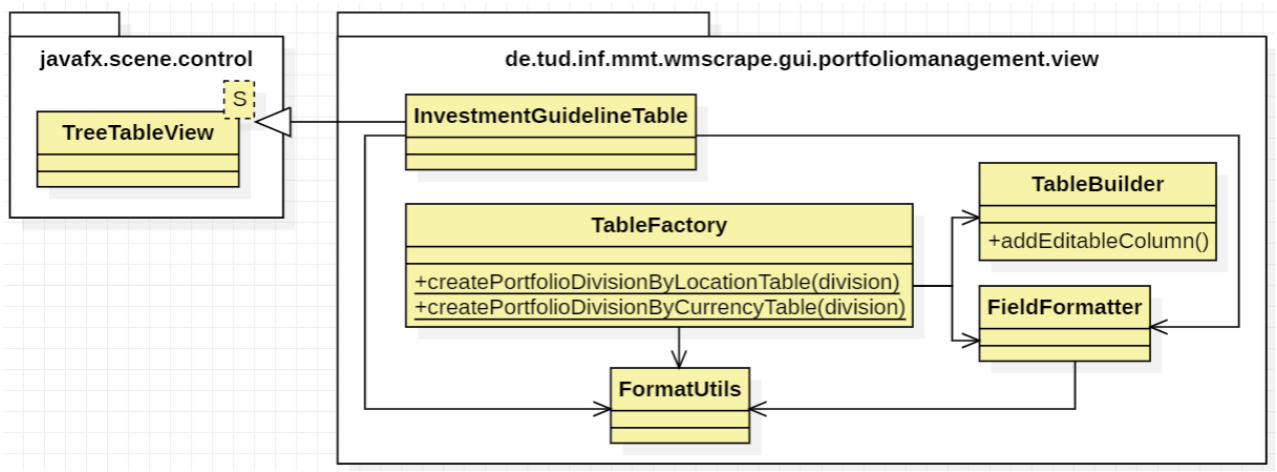

Abbildung 5.21: Detailierte Ansicht der Komponenten im *view*-Paket

Mit Hilfe der *addEditableColumn*-Methode können Spalten zur Tabelle hinzugefügt werden, deren Zellinhalte bearbeitbar sind. Diese Funktion wird genutzt, um die Tabellen für die Aufteilung des Gesamtvermögens nach Region und Währung zu realisieren. Die Erzeugung dieser Tabellen erfolgt über die Methoden *createPortfolioDivisionByLocationTable* und *createPortfolioDivisionByCurrencyTable*. Dem Parameter *division* muss dabei eine Instanz der *DivisionByLocation*- bzw. *DivisionByCurrency*-Klasse übergeben werden, aus denen die Angaben abgerufen bzw. gespeichert werden.

Die *InvestmentGuidelineTable*-Klasse leitet sich von *TreeTableView* ab, die eine hierarchische Tabellenansicht bereitstellt. Sie bindet den generischen Parameter *S* an den Typ *Entry*, sodass jede Zeile der Tabelle eine *Entry*-Instanz repräsentiert. Im Konstruktor werden zunächst die Tabellenspalten definiert, bevor die übergebenen *Entry*-Instanzen aus der Anlagenrichtlinie (*InvestmentGuideline*) zur Tabelle hinzugefügt werden.

Zur Eingabebeschränkung wird der *FieldFormatter* eingesetzt:

- *setInputIntRange*: Begrenzung der Eingabe, z. B. für die Risikoklasse mit einem Wertebereich von 1 bis 12.
- *setInputFloatRange*: Vorgesehen für Eingaben wie die Aufteilung des Gesamtvermögens. Der Parameter *to* kann hier *null* sein, um nur ein Minimum, jedoch kein Maximum festzulegen. Dies ist zum Beispiel für die Eingabe der Performance von Bedeutung.

Die *FormatUtils*-Klasse wird analog zum Hauptmenü genutzt, um Zahlen in Kommadarstellung auszugeben und die Eingabe in diesem Format zu ermöglichen.

5.2.3 Portfolio-Übersicht

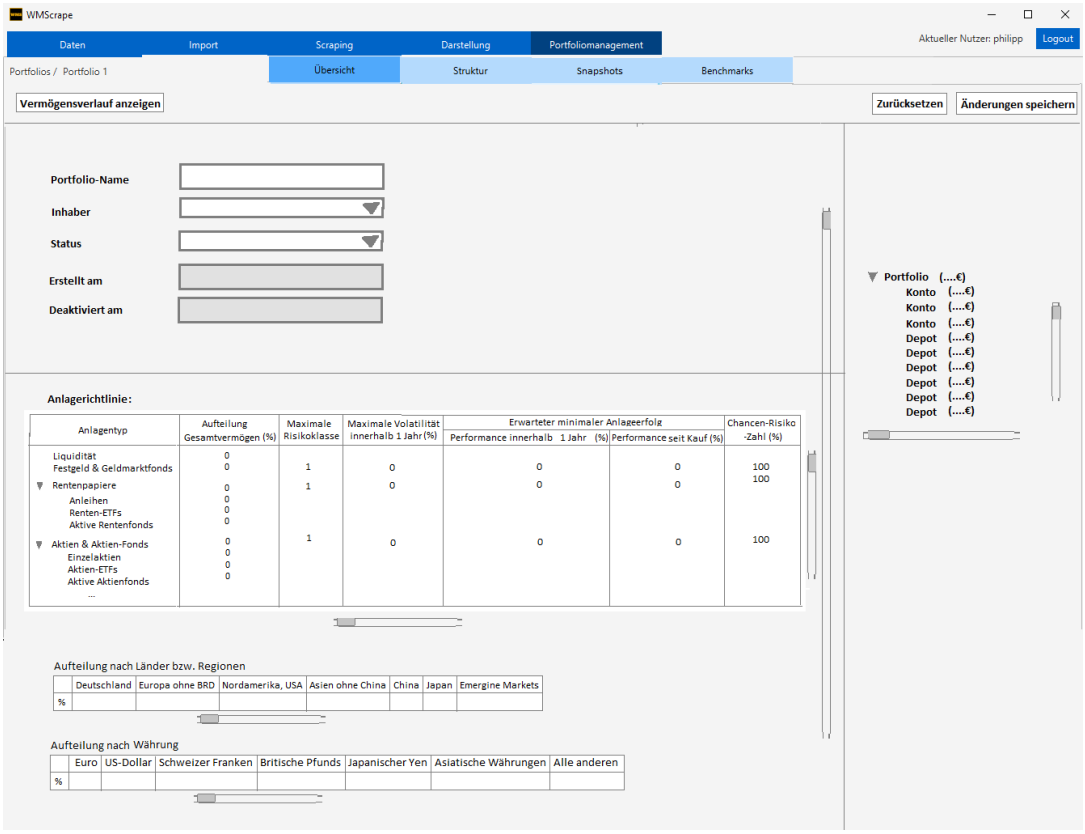

Abbildung 5.22: Mockup für die Übersicht eines Portfolios

Analog zur Übersicht eines Inhabers erhält der Benutzer in der Übersicht aus Abbildung 5.22 eine Darstellung der Parameter des Portfolios sowie der darin enthaltenen Depots und Konten. Dabei können Parameter bei Bedarf geändert werden. Der Button „Vermögensverlauf anzeigen“ soll außerdem den Vermögensverlauf über die Zeit in einem modalen Fenster darstellen.

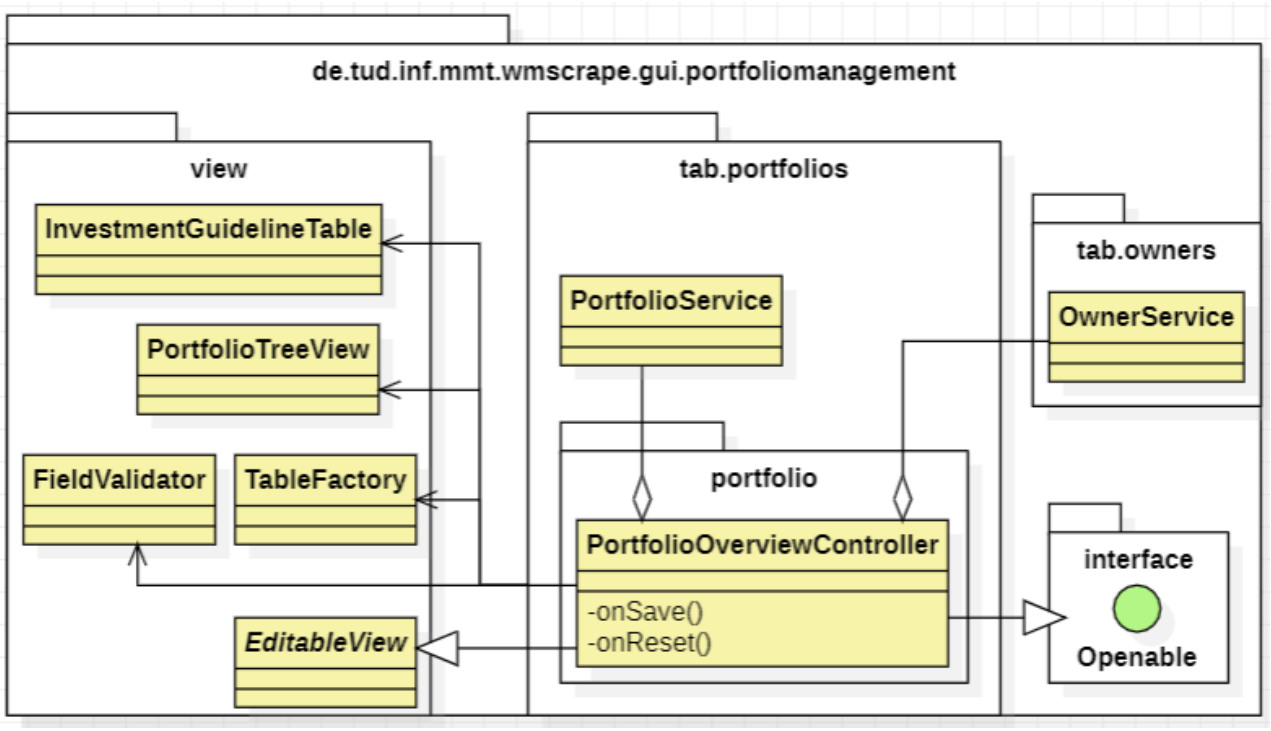

Abbildung 5.23: UML-Diagramm der Portfolio-Übersicht

Für die Darstellung der Daten und die Verarbeitung von Eingaben ist die *PortfolioOverviewController*-Klasse verantwortlich. Seine Methoden sind funktional an die der Inhaber-Verwaltung angelehnt, weshalb hier nicht näher darauf eingegangen wird. Die Beziehungen der Klasse entsprechen denen der *CreatePortfolioDialog*-Klasse. Zusätzlich verwendet der Controller die *PortfolioTreeView*-Klasse, um die dem Portfolio untergeordneten Konten und Depots darzustellen.

5.3 Konto-Verwaltung

Die Konto-Verwaltung bietet dem Benutzer analog zu den bisherigen Verwaltungskomponenten die Möglichkeit, neue Konten zu erstellen und eine Übersicht der bereits registrierten Konten anzuzeigen. Zudem wird beim „Öffnen eines Kontos“ eine detaillierte Ansicht zur Verfügung gestellt, in der dessen Parameter eingesehen werden können.

5.3.1 Hauptmenü

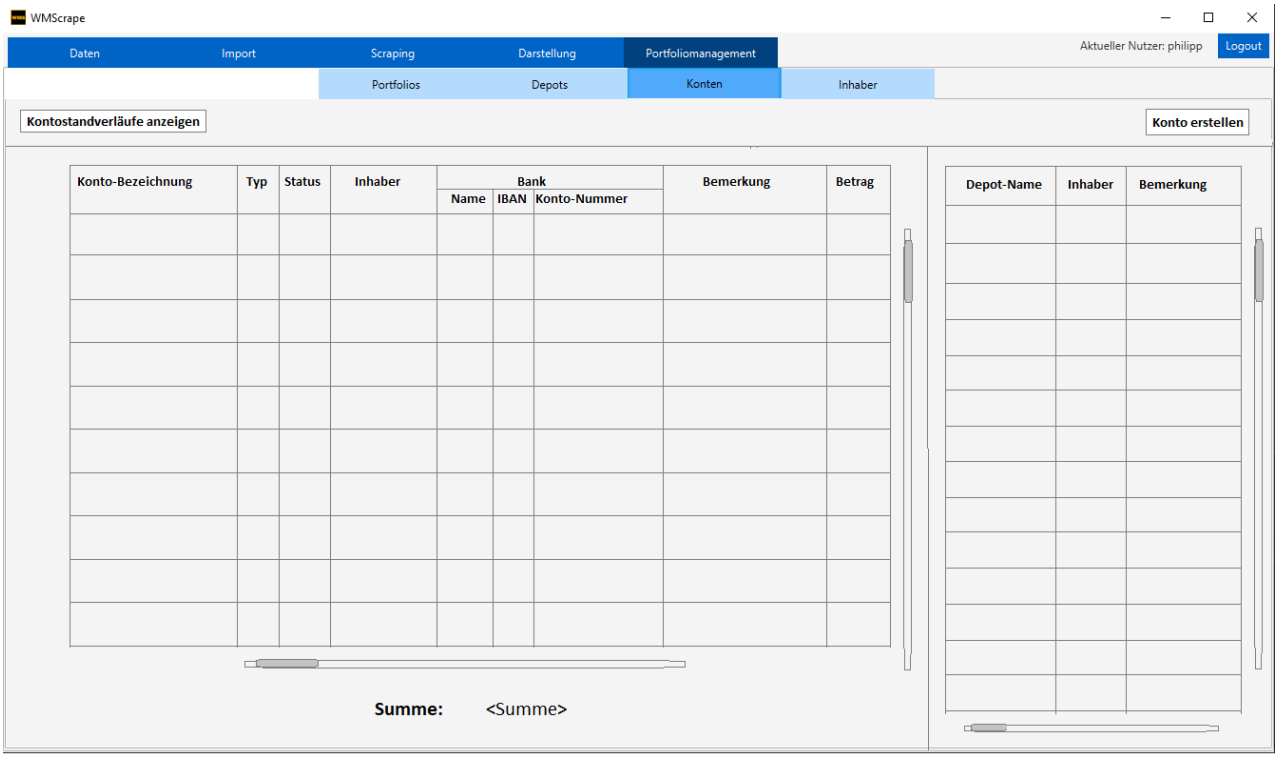

Abbildung 5.24: Mockup des Hauptmenüs der Konto-Verwaltung

Das Hauptmenü der Konto-Verwaltung aus Abbildung 5.24 bietet dem Benutzer eine Übersicht aller registrierten Konten und deren zugehörige Depots (für Verrechnungskonten). Zudem wird der Gesamtbetrag aller Konten ersichtlich. Darüber hinaus ermöglicht es die Erstellung neuer Konten, das Löschen bestehender Konten sowie die Änderung ihres Status.

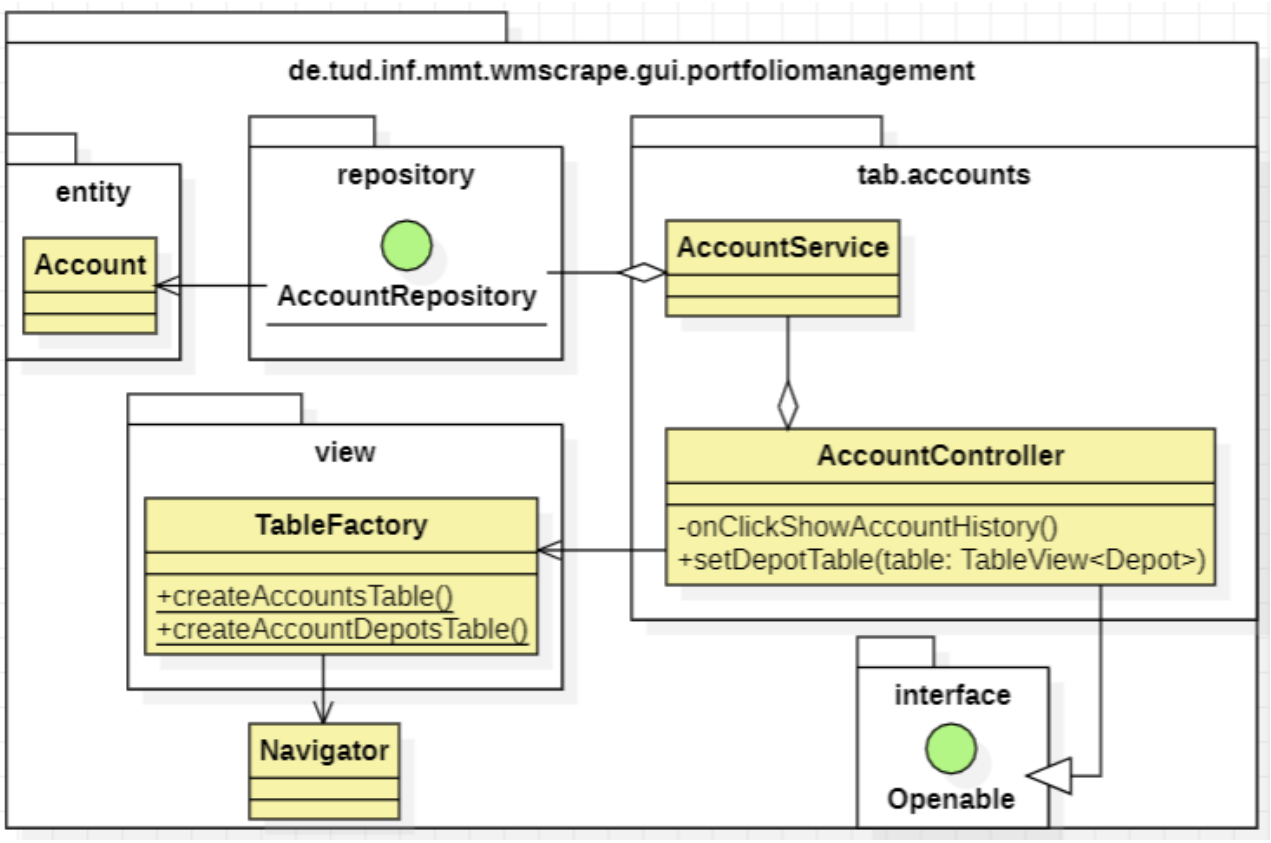

Abbildung 5.25: UML-Diagramm für das Hauptmenü

Die *AccountController*-Klasse aus Abbildung 5.25 übernimmt, analog zu den vorherigen Verwaltungskomponenten, die Steuerung der Kontenverwaltung. Die neu eingeführte *onClickShowAccountHistory*-Methode wird aktiviert, sobald der Button „Kontostandverläufe anzeigen“ betätigt wird. Da diese Funktionalität jedoch nicht im Fokus dieser Arbeit steht, wird sie nicht weiter behandelt.

Beim Öffnen des Tabs „Konto“ wird auch hier die *open*-Methode der *Openable*-Schnittstelle aufgerufen, wodurch zunächst alle Konten über die Service-Klasse (*AccountService*) geladen und anschließend im Controller durch die *createAccountsTable*-Methode tabellarisch dargestellt werden. Bei der Auswahl eines Kontos werden die zugehörigen Depots des Verrechnungskontos mithilfe der Methode *createAccountDepotsTable* angezeigt.

Die Navigation innerhalb der Anwendung wird, wie in den anderen Komponenten, über die *Navigator*-Klasse realisiert.

5.3.2 Erstellen eines Kontos

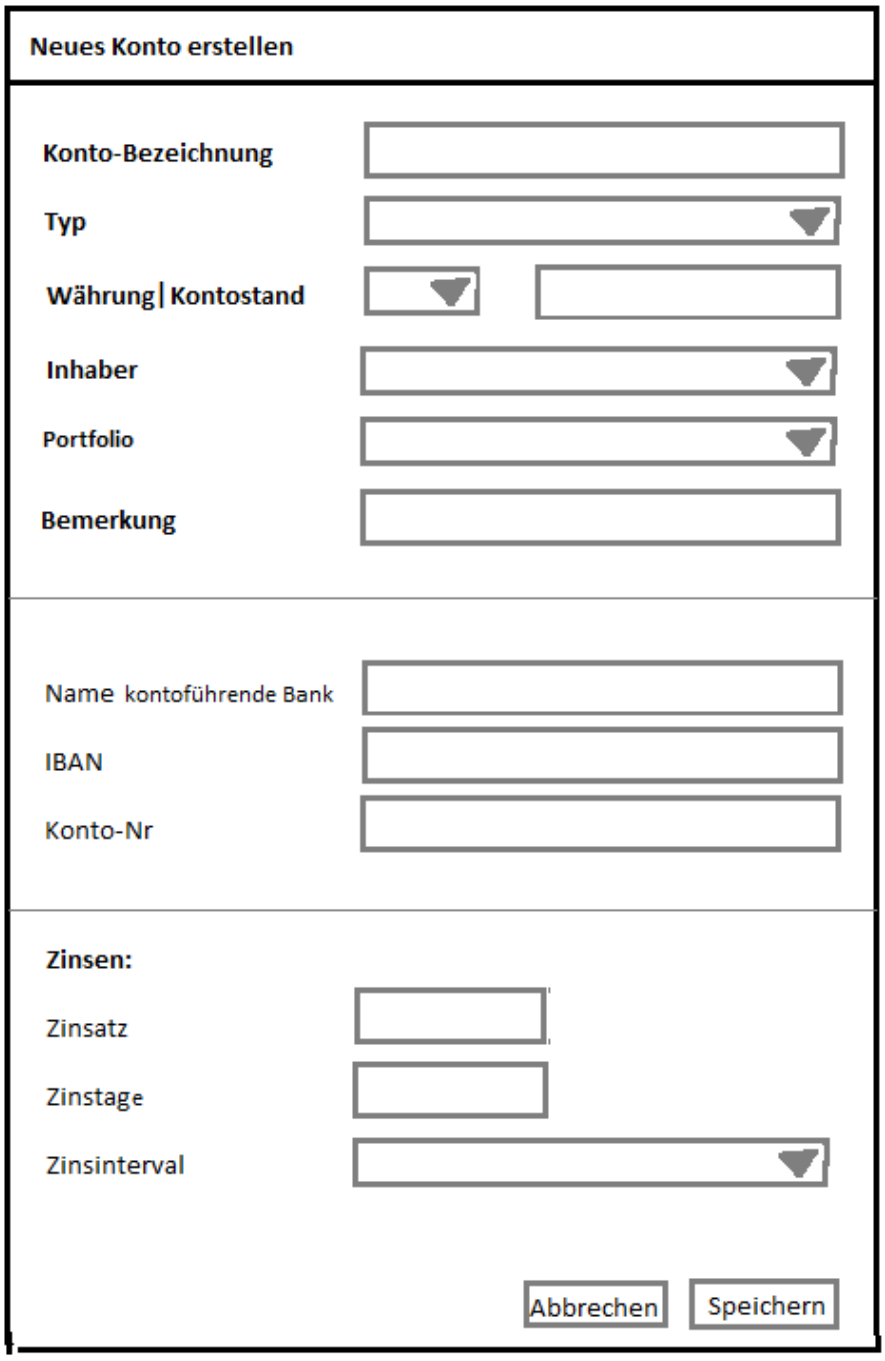

Abbildung 5.26: Mockup für den Dialog zum Erstellen eines neuen Kontos

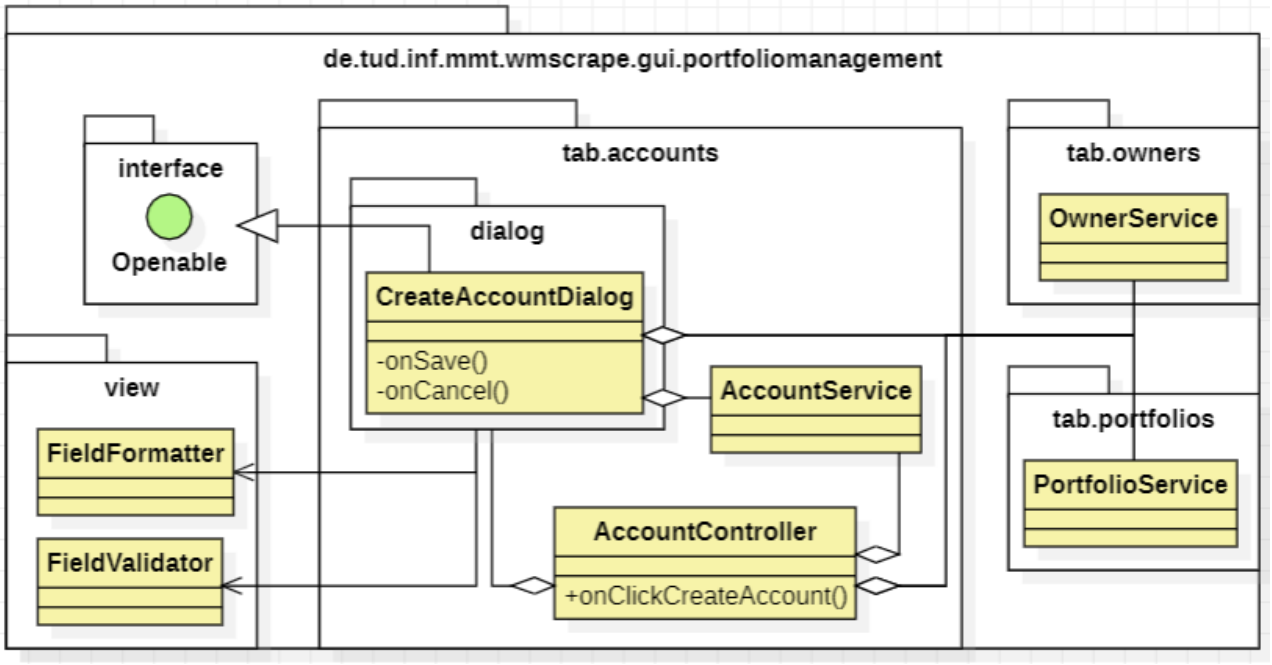

Abbildung 5.27: UML-Diagramm für den Dialog zum Erstellen eines Kontos

Der *CreateAccountDialog* aus Abbildung 5.27 dient zur Verarbeitung der Eingaben aus Abbildung 5.26. Er greift dabei in der *open*-Methode der Schnittstelle *Openable* auf die Service-Klassen der anderen Verwaltungskomponenten zu, um alle vorhandenen Inhaber und Portfolios abzufragen und dem Benutzer zur Auswahl bereitzustellen, damit diese dem neuen Konto zugeordnet werden können.

Da ein Konto nur erstellt werden kann, wenn mindestens ein Inhaber und ein Portfolio vorhanden sind, nutzt der Controller des Hauptmenüs (*AccountController*) die entsprechenden Service-Klassen, um den Dialog nur dann anzuzeigen, wenn diese Voraussetzungen erfüllt sind. Andernfalls wird der Benutzer entsprechend darüber benachrichtigt.

Zudem kommen die Klassen *FieldFormatter* und *FieldValidator* zum Einsatz, um beispielsweise die Eingabe der Anzahl der Zinstage zu beschränken und sicherzustellen, dass alle erforderlichen Felder vor dem Speichern ausgefüllt sind.

5.3.3 Konto-Übersicht

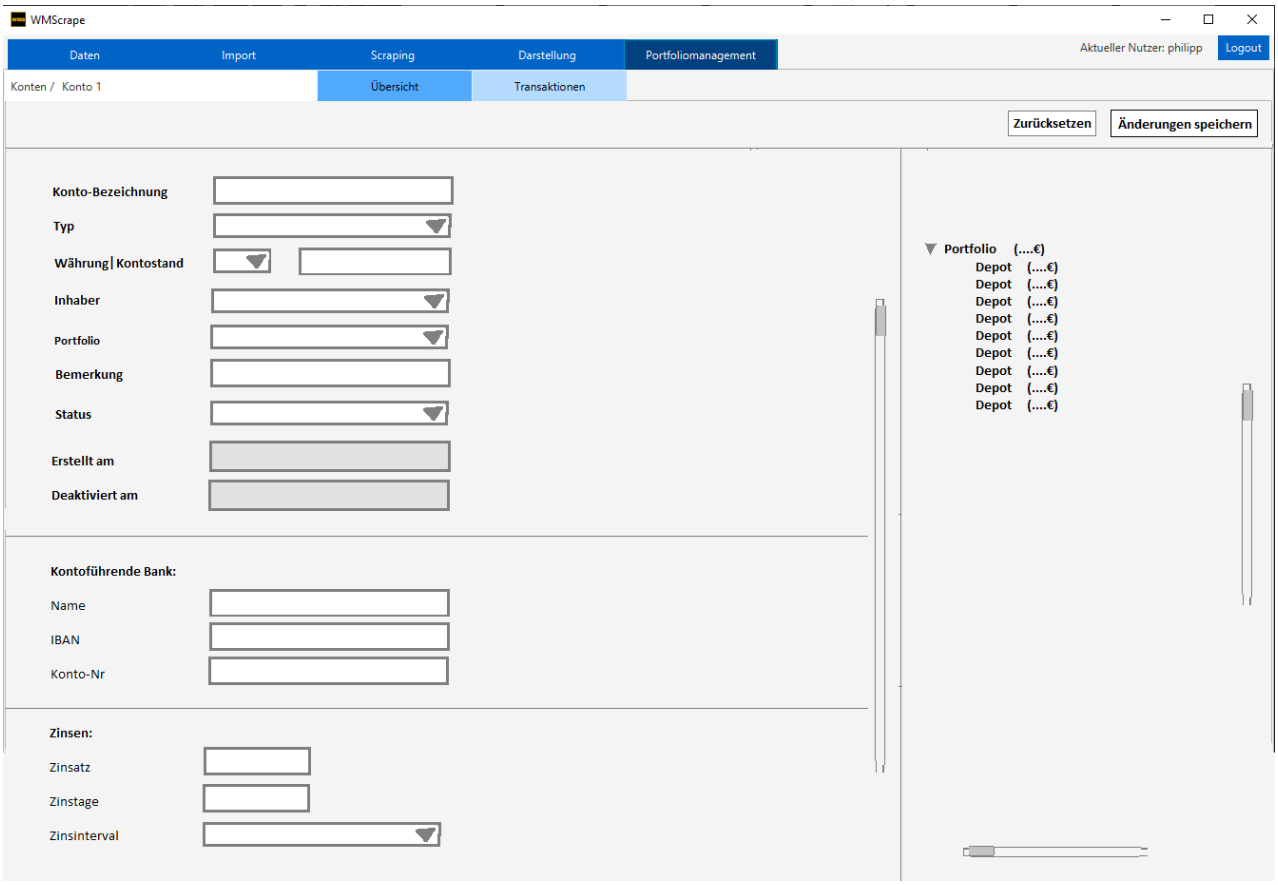

Abbildung 5.28: Mockup für die Übersicht eines Kontos

Wie in den vorherigen Übersichten erhält der Benutzer auch hier (siehe Abbildung 5.28) eine strukturierte Darstellung aller Konto-Parameter sowie die Möglichkeit, diese zu bearbeiten. In der rechten Ansicht sollen zudem die Depots dargestellt werden, denen das Konto (im Fall eines Verrechnungskontos) zugeordnet ist.

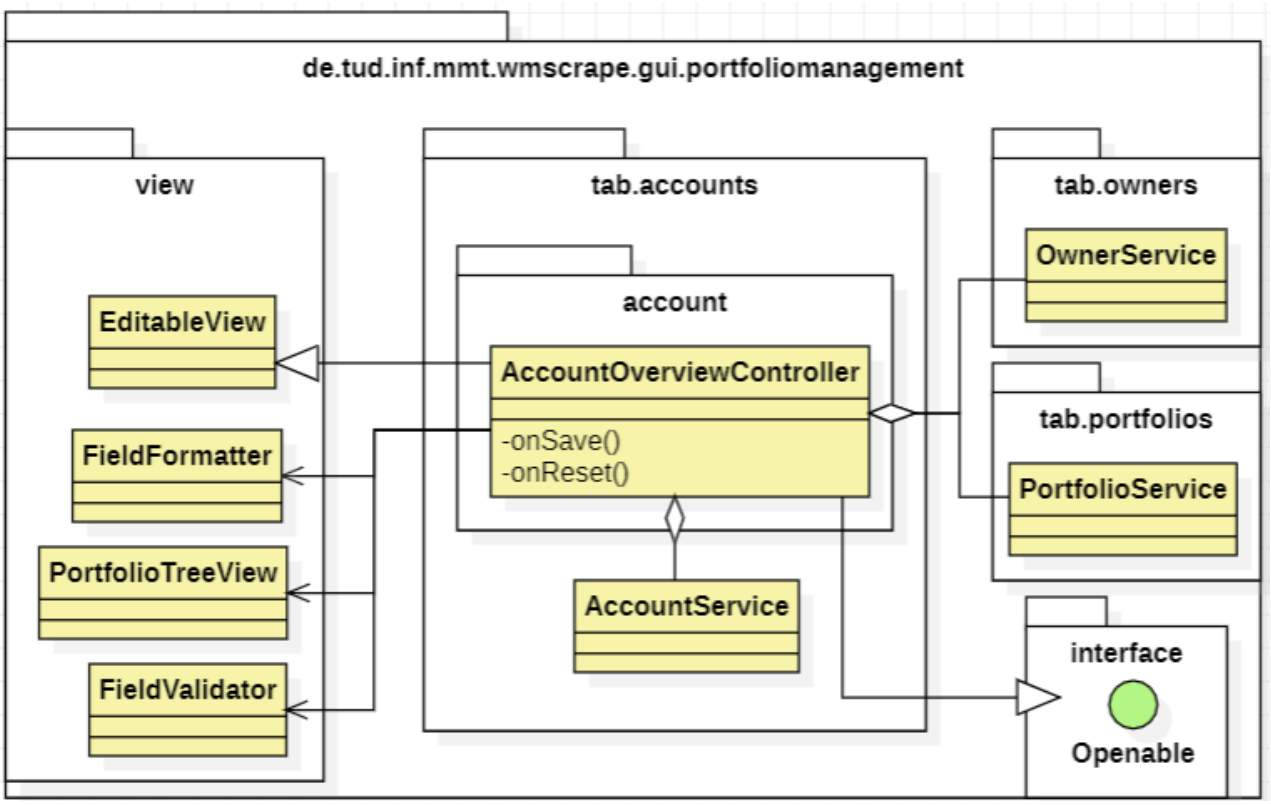

Abbildung 5.29: UML-Diagramm für die Übersicht eines Kontos

Grundsätzlich funktioniert die Übersicht ähnlich wie der Dialog im vorherigen Abschnitt bzw. die Übersicht der vergangenen Verwaltungskomponenten. Daher nutzt auch der Controller aus Abbildung 5.29 die Service-Klassen der anderen Verwaltungskomponenten, um alle aktuellen Portfolios und Inhaber abzurufen und anzuzeigen. Dies ermöglicht es, Inhaber oder Portfolios bei Bedarf neu zuzuordnen. Ebenso kommen hier die Klassen *FieldFormatter* und *FieldValidator* zum Einsatz, um Eingaben entsprechend zu formatieren und zu validieren.

Für die Darstellung der Depots, denen das Konto als Verrechnungskonto zugeordnet wurde, wird die *PortfolioTreeView*-Klasse verwendet.

5.4 Vermeidung von Datenbank-Inkonsistenzen

In der Datenbanktechnik bezeichnet Konsistenz die Integrität der gespeicherten Daten. Inkonsistenzen können schwerwiegende Fehler verursachen, insbesondere wenn die Anwendungsschicht nicht darauf ausgelegt ist, mit solchen Unstimmigkeiten umzugehen. (vgl. [Wik24b])
Die Datenintegrität wird durch folgende Bedingungen gewährleistet (vgl. [Wik24b]):

- **Bereichsintegrität**: Jeder Attributwert muss innerhalb eines vordefinierten Wertebereichs liegen.
- **Entitätsintegrität**: Der Primärschlüssel eines Objekts muss eindeutig sein und darf nicht leer bleiben.
- **Referentielle Integrität**: Ein Fremdschlüssel muss entweder leer sein oder auf ein existierendes Objekt mit dem entsprechenden Schlüssel verweisen.
- **Logische Konsistenz**: Individuell definierte Integritätsregeln, die vom Entwickler festgelegt werden.

Um Datenbank-Inkonsistenzen sowohl auf Datenbank- als auch auf Anwendungsebene zu vermeiden, sind präventive Maßnahmen erforderlich. Dazu gehört die Annotation von Tabellenspalten (z. B. als nicht-nullable) und Beziehungen sowie die Validierung von Benutzereingaben. Neben der Prävention spielt auch die Erkennung bestehender Inkonsistenzen eine wichtige Rolle. Da die Implementierungen nicht als endgültig betrachtet werden, sondern im Zuge zukünftiger Arbeiten angepasst werden könnten, ist eine fortlaufende Überprüfung essenziell. Wird beispielsweise durch eine spätere Änderung ein Fehler eingeführt, der eine Eingabe von 1000% für den Kirchensteuersatz ermöglicht, wäre die Daten-Integrität verletzt – solche Unstimmigkeiten müssen erkannt und behoben werden.

5.4.1 Maßnahmen zur Erkennung und Behandlung

Ziel dieser Maßnahmen ist es, Inkonsistenzen zu erkennen und dem Benutzer daraufhin die Möglichkeit zur Behandlung dieser zu geben. Dadurch soll die Anwendung robuster gegenüber zukünftigen Anpassungen gemacht werden. Die konkreten Maßnahmen ergeben sich aus den hier behandelten Inkonsistenzen:

- Fehlende Datenbanktabellen oder Beziehungen (z. B. Fremdschlüssel-Beziehungen): Diese werden von Hibernate automatisch wiederhergestellt. Dies liegt an der Konfiguration „spring.jpa.hibernate.ddl-auto=update“[8] in der Application-Property-Datei.
- Daten, die eindeutig sein sollten (z. B. die Steuernummer eines Inhabers), sind mehrfach vorhanden.
- Ungültige oder leere Fremdschlüssel
- Ungültige Enum-Werte: Java-Enums können in der Datenbank entweder über ihren ordinalen Wert oder ihren Namen gespeichert werden. In der aktuellen Implementierung erfolgt die Speicherung über den Namen. Beim Laden eines Datensatzes (z. B. eines Inhabers) muss der gespeicherte Name einem existierenden Enum-Literal zugeordnet werden können. Ist dies nicht der Fall, wirft Hibernate einen Fehler und bricht den Ladevorgang ab.

[8] https://hyperskill.org/learn/step/26222

- Pflichtfelder sind leer: Bestimmte Felder (z. B. IBAN) sind als nicht-leer gekennzeichnet, enthalten aber dennoch keine Werte.
- Fehlende oder doppelte Einträge in der Anlagenrichtlinie: Beispielsweise tritt der Eintrag für den Anlagentyp „Liquidität“ mehrfach auf.
- Fehlerhafte Datenstruktur der Anlagenrichtlinien-Einträge: Dies gilt speziell für die korrekte Zuordnung von untergeordneten zu den richtigen übergeordneten Einträgen. Eine falsche Datenstruktur wäre beispielsweise, wenn der Eintrag zum Anlagentyp „Aktien & Aktienfonds“ einen untergeordneten Eintrag des Typs „Liquidität“ enthält.
- Fehlerhafte Summenberechnung in der Anlagenrichtlinie: Die Summe aller übergeordneten Einträgen muss exakt 100% betragen. Analog dazu müssen die untergeordneten Einträge 0% bzw. 100% ergeben.
- Fehlerhafte Summenberechnung bei Länder- oder Währungsaufteilungen: Die Summe der Anteile muss jeweils genau 100% betragen.
- Werte außerhalb ihres definierten Bereichs: Beispielsweise darf der Zinssatz nicht kleiner als 0% oder größer als 100% sein.
- Inkonsistenter Status: Eine Entität ist als „Deaktiviert“ markiert, jedoch fehlt das zugehörige Deaktivierungsdatum.
- Das Erstell-Datum liegt in der Zukunft bzw. das Deaktiviert-Datum liegt vor dem Erstell-Datum bzw. in der Zukunft
- Zu der Währung eines Kontos existiert (sofern die Währung nicht Euro ist) kein Wechselkurs

5.4.2 Ablauf

Bei jedem Lade-, Lösch- sowie Speichervorgang erfolgt eine Überprüfung auf Inkonsistenzen. Da Inhaber, Portfolios und Konten in Beziehung stehen, muss jedoch stets nach Inkonsistenzen in allen dieser Entitäten geprüft werden.

Der allgemeine Ablauf der Überprüfung und Behandlung von Inkonsistenzen ist in Abbildung 5.30 dargestellt bzw. in der *PortfolioManagementTabController*-Klasse vorzufinden. Grundsätzlich erfolgt der Prozess für alle Entitäten gleich. Im Fall eines Inhabers läuft die Prüfung wie folgt ab (vgl. Abbildung 5.30):

1. Abfrage der IDs inkonsistenter Inhaber: Falls keine inkonsistenten Einträge gefunden werden, wird der Prozess beendet.

2. Rekonstruktion der betroffenen Inhaber anhand ihrer IDs: Dies erfolgt unabhängig von Hibernate, da Hibernate sonst zunächst versuchen würde die Inhaber-Daten in den sogenannten Persistence Context (auch als First-Level-Cache bekannt) zu laden[9], welcher sich alle Entitätsinstanzen merkt (vgl. [Dal23]). Da die Entität jedoch inkonsistent ist, kann dies zu Fehlern führen, sodass der Ladevorgang fehlschlägt.

3. Benachrichtigung des Benutzers: Der allgemeine Dialog informiert den Benutzer über gefundene Inkonsistenzen. Nach einem Klick auf „Inkonsistenzen anzeigen“ öffnet sich der Dialog zur Einsicht der inkonsistenten Inhaber-Daten.

4. Korrektur oder Löschung durch den Benutzer: Der Benutzer kann die betroffenen Parameter bearbeiten oder den inkonsistenten Inhaber vollständig löschen.

5. Erneute Überprüfung der Daten: Der Vorgang beginnt erneut bei Schritt 1. Dies ist notwendig, da die Validierung der korrigierten Daten auf der Geschäftslogik basiert. Falls eine fehlerhafte Implementierung weiterhin ungültige Daten speichert, kann die Entität nur noch gelöscht werden.

[9] https://chatgpt.com/share/6764a71b-b128-8001-a7b5-f84a06c97d97

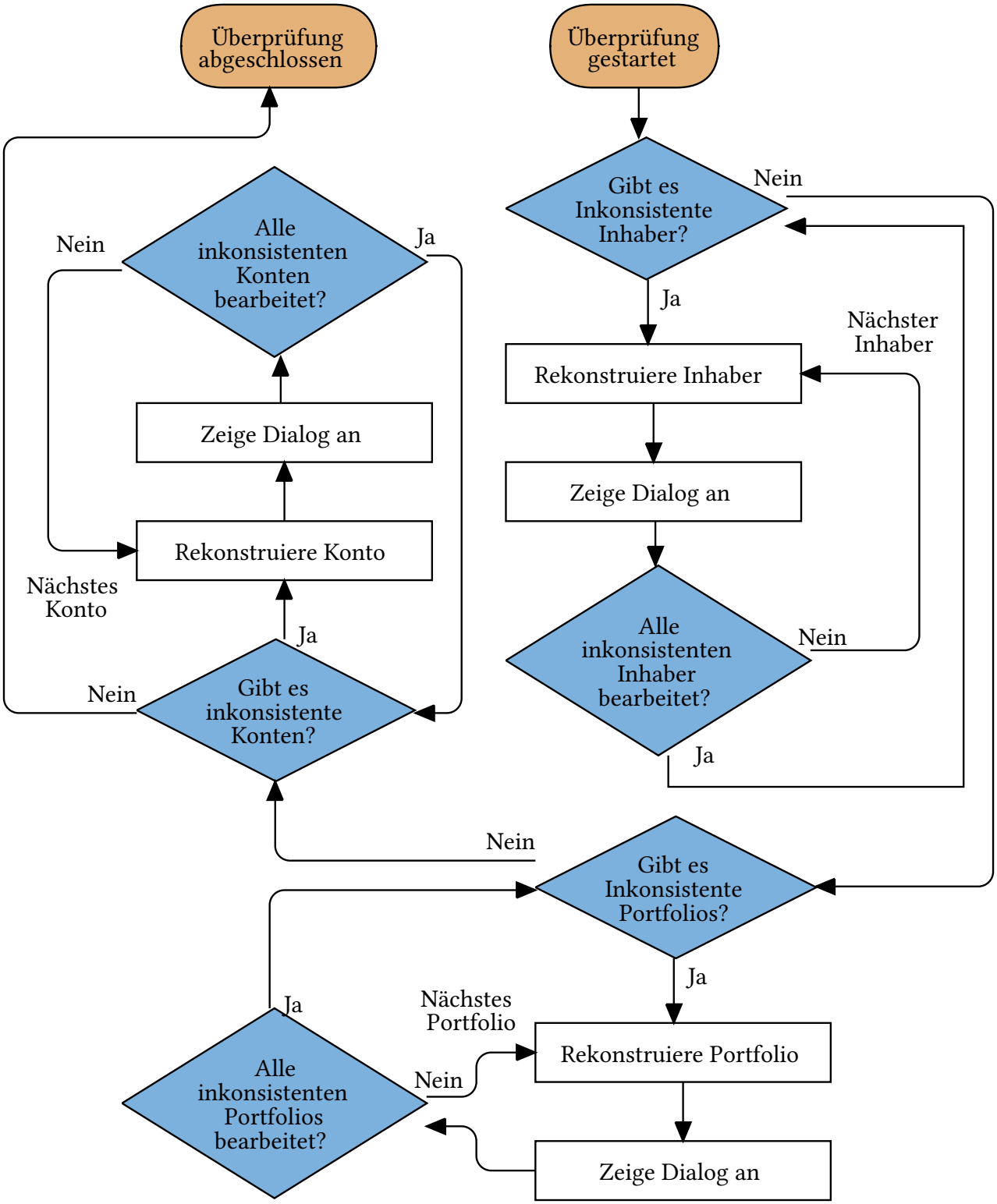

Abbildung 5.30: Ablauf der Überprüfung und Behandlung von Inkonsistenzen

5.5 Breadcrumb-Funktion

Abbildung 5.31: Beispielhafte Darstellung der Breadcrumb-Bar

Die Breadcrumb-Funktion (Brotkrumen-Navigation) aus Abbildung 5.31 ist ein Navigationselement, das dem Benutzer eine visuelle Orientierung innerhalb einer hierarchischen Struktur gibt. Sie zeigt den aktuellen Standort innerhalb der Anwendung und ermöglicht es, schnell zu übergeordneten Ebenen zurückzukehren.

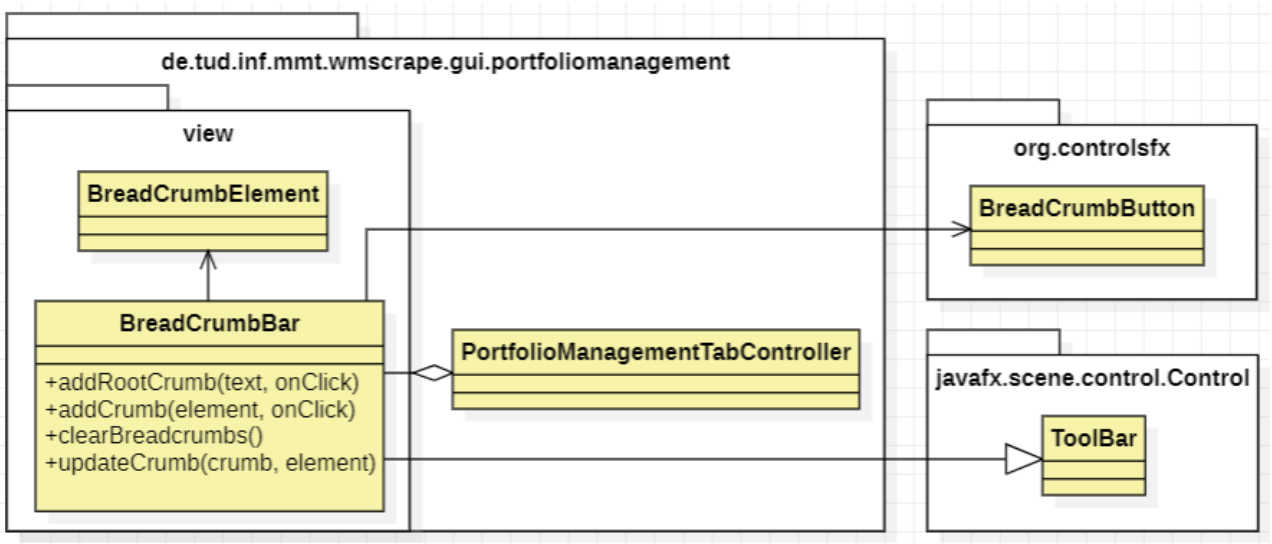

Abbildung 5.32: UML-Diagramm für die Breadcrumb-Funktion

Im Wesentlichen repräsentiert die *BreadCrumbBar*-Klasse aus Abbildung 5.32 die Breadcrumb-Bar in Abbildung 5.31. Sie erbt von der *ToolBar*-Klasse, um das Verhalten zu übernehmen, bei Platzmangel für die Darstellung der Breadcrumbs die entsprechenden Breadcrumbs in einem Overflow-Menü wie in Abbildung 5.33 darzustellen. Die Breadcrumbs selbst werden über Buttons des Typs *BreadCrumbButton* dargestellt.

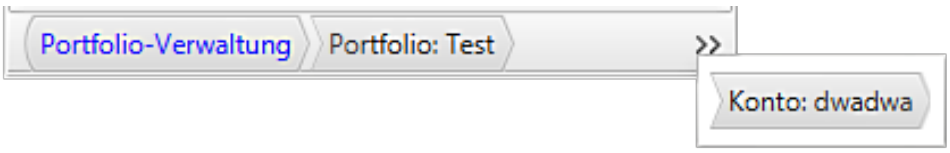

Abbildung 5.33: Overflowmenü der Breadcrumb-Bar

Die Controller-Klasse für das Portfoliomanagement (*PortfolioManagementTabController*), welche die Steuerung der Tableiste im Portfoliomanagement übernimmt, ergänzt beim Öffnen einer Ansicht (z. B. der Inhaber-Übersicht) mithilfe der Methoden *addRootCrumb* bzw. *addCrumb* einen entsprechenden Breadcrumb in der Navigationsleiste.

Die Methode *addRootCrumb* wird ausschließlich zur Erstellung des initialen Breadcrumbs verwendet – etwa „Portfolio-Verwaltung“ in Abbildung 5.31. Ihr wird lediglich der anzuzeigende Text übergeben. Im Gegensatz dazu erhält *addCrumb* ein Objekt vom Typ *BreadCrumbElement*, welches unter anderem das zugehörige Datenobjekt (z. B. einen Inhaber) enthält. Dadurch ergibt sich die Beschriftung des Breadcrumbs aus dem übergebenen Objekt.

Der initiale Breadcrumb wird visuell hervorgehoben: Er erscheint in blauer Schrift und hebt sich in seiner Gestaltung von den nachfolgenden Breadcrumbs ab, um den Startpunkt der Navigation – die Verwaltungskomponente – eindeutig zu kennzeichnen.

Um einen bestehenden Breadcrumb zu aktualisieren, wird die Methode *updateCrumb* aufgerufen. Ihr wird das zu ändernde Breadcrumb sowie ein neues Objekt übergeben, das den aktualisierten Inhalt repräsentiert. Dies ermöglicht es, die Anzeige des Breadcrumbs entsprechend zu überschreiben.

Beim Wechsel zum Hauptmenü einer Verwaltungskomponente werden alle Breadcrumbs durch den Aufruf von *clearBreadcrumbs* entfernt.

6 Umsetzung

Für die Verwaltungskomponenten wurde ein iterativer Entwicklungsansatz gewählt, um Fortschritte frühzeitig sichtbar zu machen und direkt testen zu können. Dabei wurden Erweiterungen nicht in einem einzigen Durchgang vollständig entwickelt, sondern in mehreren Zyklen (Iterationen) schrittweise verbessert und erweitert. Einzelne Funktionalitäten wurden nach und nach implementiert, getestet und bei Bedarf angepasst.

Ein Beispiel hierfür ist das Hauptmenü der Inhaber-Verwaltung: Zunächst wurde die Tabelle umgesetzt, die alle registrierten Inhaber anzeigt. Nach einer ersten Validierung wurde anschließend die Baumansicht zur Darstellung der dem Inhaber zugeordneten Portfolios implementiert und ebenfalls überprüft, bevor die Entwicklung weitergeführt wurde. Dieses Vorgehen ermöglichte es, die Funktionalität frühzeitig zu validieren und potenzielle Probleme zeitnah zu erkennen.

Um ein besseres Verständnis des Quellcodes zu ermöglichen, werden die Namen von wichtigen Klassen, Methoden und Variablen in diesem Kapitel kursiv dargestellt. Zudem wird folgende Abkürzung eingeführt: *KlasseA.methodeA* - damit wird verkürzt ausgedrückt, dass die Methode *methodeA* aus der Klasse *KlasseA* gemeint ist.

6.1 Technologien, Frameworks und Bibliotheken

Im Rahmen dieser Arbeit wurde die Bibliothek *org.controlsfx*[1] neu integriert, um die Breadcrumb-Funktion benutzerfreundlicher zu gestalten. Die Bibliothek unterliegt der 3-Klausel-BSD-Lizenz[2], was bedeutet, dass sie frei verwendet werden darf, solange der Copyright-Vermerk des ursprünglichen Programms erhalten bleibt (vgl. [Lin]).

6.2 Inhaber-Verwaltung

6.2.1 Hauptmenü

Das Hauptmenü der Inhaber-Verwaltung aus Abbildung 5.1 wird durch den *OwnerController* aus Abbildung 5.2 gesteuert, welcher die Tabelle und Baumansicht lädt. Zudem reagiert er auf Benutzeraktionen wie das Klicken auf den Button „Inhaber erstellen“ mittels der Methode *onClickCreateOwner* (vgl. Abbildung 5.7). Die Daten werden dynamisch aktualisiert, indem die Methode *open* der

[1] https://central.sonatype.com/artifact/org.controlsfx/controlsfx/overview
[2] https://de.wiki.bluespice.com/wiki/BSD_3-Clause

Schnittstelle *Openable* - wie in Abbildung 5.2 dargestellt - implementiert wird. Diese Methode wird beispielsweise beim Öffnen des Tabs „Inhaber“ aufgerufen. Der Name der Schnittstelle ergibt sich aus folgender Problematik: die Ansicht eines Tabs wird durch FXML initialisiert – dieser Vorgang findet bereits beim Programmstart statt. Dadurch wird die Methode *initialize*[3] im Controller einmalig ausgeführt. Allerdings müssen die Daten nach der Initialisierung des Controllers bei jedem erneuten Aufruf bzw. Öffnen des Tabs neu geladen werden. Rückblickend wäre eine Bezeichnung wie *Refreshable* passender gewesen.

Zunächst werden die Daten für die Ansicht über die *getAll*-Methode der Service-Klasse (*OwnerService*) aus Abbildung 5.3 abgerufen, welche die Inhaberdaten aus der Datenbank lädt. Hierbei wird die *findAll*-Methode des Repositories *OwnerRepository* aufgerufen. Diese wird durch JPA automatisch implementiert[4].

Nachdem die Daten der Inhaber geladen wurden, wird die Inhaber-Tabelle durch den Aufruf der *TableFactory.createOwnersTable*-Methode aus Abbildung 5.5 erstellt. Dabei werden folgende Parameter übergeben:

- Ein Objekt vom Typ *Parent*, das beispielsweise eine *Pane*-Instanz[5] sein kann, zu der die Tabelle hinzugefügt wird. Dieses Objekt soll nach der Erzeugung ausschließlich die Tabelle enthalten und dient dazu, die Größe der Tabelle an die der Pane anzupassen.
- Eine Liste der darzustellenden Inhaber.
- Die Controller-Klasse des Hauptmenüs, um beim Auswählen eines Inhabers dessen Portfolios über den Aufruf der Methode *OwnerController.setTreeView* (vgl. Abbildung 5.2) anzuzeigen.
- Die Service-Klasse, um unter anderem das Löschen von Inhabern zu ermöglichen.

Zur Erstellung wird u.a. die Methode *TableBuilder.addColumn* verwendet, um Spalten wie „Vorname“ hinzuzufügen. Hierbei wird der Spaltenkopftitel, die prozentuale Breite der Spalte sowie eine Rückruffunktion (über den Parameter *cellValueFactory*) zur Darstellung des Zellinhalts übergeben. (vgl. Abbildung 5.5)

Die *cellValueFactory* ist eine funktionale Schnittstelle vom Typ *Callback<P, R>*, wobei hier der generische Parameter *P* an den generischen Typ *TableColumn.CellDataFeatures<S, T>* und analog *R* an *ObservableValue<T>* gebunden wird. Zusammengefasst ist der Parameter daher vom Typ *Callback<TableColumn.CellDataFeatures<S, T>, ObservableValue<T>*. Diese Schnittstelle bietet eine Funktion an (eine sogenannte Rückruffunktion) und wird beim Rendern einer Tabellenzelle aufgerufen, um den darzustellenden Zellinhalt abzurufen. Dabei wird die zu rendernde Zelle (*TableColumn.CellDataFeatures<S, T>*) übergeben und ein Objekt vom Typ *ObservableValue<T>* erwartet. Der Aufruf erfolgt entsprechend aus der Implementierung der *TreeView*-Klasse. Der Typ *T* stellt

[3] https://docs.oracle.com/javase/8/javafx/api/javafx/fxml/doc-files/introduction_to_fxml.html#controllers

[4] https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/jpa.repositories.html

[5] Eine Pane in JavaFX ist ein grundlegendes Container-Element, das als Layout-Manager dient.

zudem den Typ dar, den die Zellen innerhalb der Spalte repräsentieren (z.B. *String* für Texte wie dem Inhaber-Namen). Der generische Typ *ObservableValue<T>* stellt hingegen eine Klasse dar, die ein Wert vom Typ T hält, für den Listeners registriert werden können, um über Änderungen dieses Wertes informiert zu werden. Zusammengefasst lässt sich dieser Parameter mathematisch wie folgt darstellen:

$$f : \text{TableColumn.CellDataFeatures}\langle S, T\rangle \rightarrow \text{ObservableValue}\langle T\rangle$$

Ein anschauliches Beispiel hierfür ist in Abbildung 6.1 dargestellt, in der die *cellValueFactory* für die Spalte „Vorname“ implementiert wird. Dabei repräsentiert *Owner* das zugehörige Objekt einer Tabellenzeile, während *String* den Zellinhaltstyp angibt. Im Kern besteht die Aufgabe darin, aus der zu rendernden Zelle die Instanz des jeweiligen Inhabers abzuleiten, die zur entsprechenden Zeile gehört, um daraus die relevanten Daten – in diesem Fall den Vornamen – bereitzustellen.

```
1 Callback<TableColumn.CellDataFeatures<Owner, String>, ObservableValue<String>>
      cellValueFactory;
2 cellValueFactory = new Callback<>() {
3     @Override
4     public ObservableValue<String> call(TableColumn.CellDataFeatures<Owner,
          String> cell) {
5         return new SimpleStringProperty(cell.getValue().getForename());
6     }
7 };
```

Listing 6.1: Beispiel für die Implementierung des Parameters cellValueFactory

Die Aktionen bei der Auswahl eines Inhabers, wie das „Öffnen“, werden hingegen durch den Aufruf der Methoden *setActionOnSingleClickRow* sowie *setActionOnDoubleClickRow* aus Abbildung 5.5 konfiguriert. Durch eine entsprechende Implementierung der Parameter der ersten Methode wird bei einem einfachen Klick auf eine Tabellenzeile die *OwnerController.setOwnerTreeView*-Methode aufgerufen, die die Baumansicht mit den Portfolios des ausgewählten Inhabers anzeigt. Ein doppelter Klick auf eine Zeile navigiert den Benutzer zur Übersicht des Inhabers (siehe Abbildung 5.8) mit Hilfe der *Navigator*-Klasse aus Abbildung 5.5.

Die Navigation zu einem Inhaber erfolgt über die Methode *Navigator.navigateToOwner*, die wiederum *PortfolioManagementTabManager.showInhaberTabs* aufruft, die die inhaber-spezifischen Tabs anzeigt und den Benutzer zur Übersicht weiterleitet. Der Parameter *addBreadCrumb* in *navigateToOwner* steuert dabei, ob ein Breadcrumb hinzugefügt wird (*true*) oder nicht (*false*). Detaillierte Informationen zur Navigation werden hier nicht weiter ausgeführt; stattdessen wird auf Abschnitt 6.2.3 verwiesen. Ein Breadcrumb wird nicht gesetzt, wenn eine Navigation aus einer Übersicht heraus erfolgt, aber ungespeicherte Änderungen vorliegen, sodass der Benutzer den Vorgang abbrechen und in der Übersicht verbleiben muss – dazu später mehr.

Die Portfolios werden innerhalb der *TableFactory.createOwnersTable*-Methode dargestellt, indem *OwnerController.setOwnerTreeView* aufgerufen wird. Die *setOwnerTreeView*-Methode fügt die zuvor erstellte Baumansicht beispielsweise einer *Pane* hinzu und macht sie dadurch sichtbar. Die

Erzeugung der Baumansicht erfolgt durch den Konstruktoraufruf der *PortfolioTreeView*-Klasse aus Abbildung 5.5, wobei folgende Parameter übergeben werden:

- Analog zur Tabelle ein Objekt von Typ *Parent*
- Die Portfolios des Inhabers, die man über die *Owner.getPortfolios*-Methode erhält. Die Portfolios werden von Hibernate beim Aufruf der bereits genannten *getAll*-Methode den Inhabern zugeordnet. Dies resultiert aus der definierten Beziehung innerhalb der Inhaber-Klasse:

```
1    @OneToMany(mappedBy = "owner")
2    private Set<Portfolio> portfolios = Collections.emptySet();
```

- Eine Instanz der *PortfolioManagementTabManager*-Klasse, um auf die Service-Klasse der Konto-Verwaltung zugreifen zu können, damit der entsprechende Wechselkurs abgefragt werden kann, um so die Umrechnung aller Beträge in Euro zu ermöglichen.

Im Konstruktor wird zunächst die Wurzel des Baumes mit dem statischen Text „Portfolio“ oder „Portfolios“ erstellt, abhängig von der Anzahl der übergebenen Portfolios. Anschließend werden die Portfolios als untergeordnete Elemente hinzugefügt. Die hierarchische Struktur wird mit der Methode *setItems* aufgebaut, die durch die übergebenen Portfolios iteriert und dabei deren zugehörige Konten und Depots in der Baumansicht unterordnet.

Die Einträge in der Baumansicht werden durch die generische Klasse *TreeItem<T>* aus dem gleichen Paket wie *TreeView* dargestellt. Dabei ist der generische Typ *T* an die Klasse *PortfolioTreeView.Item* gebunden. (vgl. Abbildung 5.5) Diese Klasse enthält ein Attribut vom Typ *FinancialAsset*, einer abstrakten Klasse, die von *Portfolio*, *Account* und *Depot* geerbt wird. Die *TreeItem*-Klasse nutzt die *toString*-Methode der jeweiligen *Item*-Instanz zur Darstellung der Einträge. Die genaue Darstellung variiert je nach Typ der Finanzanlage:

- **Portfolio:** „*Portfolio-Name* (*Portfolio-Wert* €)“, z. B.: „Sandbox-Portfolio (391,02 €)“
- **Konto:** „Konto: *IBAN* (*Kontostand* €)“, z. B.: „Konto: DE02500105170137075030 (391,02 €)“
- **Depot:** „Depot: *Depot-Name* (*Depot-Wert* €)“, z. B.: „Depot: Comdirect-Depot (0,00 €)“

Das Abrufen des Wertes beispielsweise eines Portfolios erfolgt mittels der *Valuable.getValue*-Methode. Eine genaue Erläuterung zur Funktionsweise der Umrechnung soll allerdings der Konto-Verwaltung vorbehalten sein, da die Funktion erst in dieser Verwaltung relevant wird. Entsprechend des Finanzanlagentyps gibt die Methode außerdem folgende Werte (umgerechnet) zurück:

- **Portfolio**: Summe der Werte aller enthaltenen Konten und Depots
- **Konto**: den Kontostand
- **Depot**: Summe der aktuellen Werte aller enthaltenen Wertpapier
  (*Hinweiß:* die Implementierung ist für zukünftige Arbeiten vorbehalten.)

In der Konzeption wurde außerdem davon ausgegangen, dass Klickaktionen auf die Einträge (*TreeItem*) registriert werden können. Da sich bei der Implementierung herausstellte, dass dies nicht möglich ist, musste entsprechend eine neue Lösung gefunden werden, sodass stattdessen die Auswahl eines Eintrags über ein Listener der *selectedItemProperty* des SelectionModels[6] der *TreeTableView*-Klasse überwacht werden kann.

Die Aktion „Inhaber erstellen“ wird durch den Button ausgelöst, welcher in der FXML-Datei mit der Methode *onClickCreateOwner* verknüpft ist. Der Dialog dazu (vgl. Abbildung 5.6) wird durch die statische Methode *loadFxml* der Klasse *PrimaryTabManager* geladen und angezeigt.

Abschließend soll noch kurz auf ein zentrales Problem im Umgang mit Beziehungen zwischen Entitätsklassen eingegangen werden, das auch für zukünftige Entwicklungen von Bedeutung ist. Dies ist insofern von Bedeutung, da die bisherige Implementierung vergangener Arbeiten die Übersichtlichkeit und Lesbarkeit des Quellcodes beeinträchtigte. Konkret geht es um die Fehlermeldung „cannot simultaneously fetch multiple bags“, die auftritt, wenn mehrere Beziehungen vom Typ *List* geladen werden und der *fetch*-Parameter der *@OneToMany*-Annotation auf *FetchType.EAGER* gesetzt ist. Die *@OneToMany*-Annotation definiert eine Beziehung zwischen Entitäten, wobei *FetchType.EAGER* bewirkt, dass die referenzierten Entitäten unmittelbar mit der Hauptentität geladen werden. Ein detailliertes Verständnis dieses Problems erfordert eine tiefere Auseinandersetzung mit Hibernate, was hier jedoch den Rahmen sprengen würde. Daher soll eine kurze Erklärung genügen: Die Ursache des Fehlers liegt darin, dass beim gleichzeitigen Laden mehrerer Beziehungen vom Typ *List* ein kartesisches Produkt entsteht. Dies kann zu Duplikaten führen, die Hibernate nicht korrekt auflösen kann. Im Gegensatz dazu interpretiert Hibernate Beziehungen vom Typ *Set* als eindeutige Sammlungen, sodass das gleichzeitige Laden mehrerer solcher Relationen problemlos möglich ist.[7]

Eine Lösung für dieses Problem besteht darin, anstelle einer *List* eine *Set*-Struktur zu verwenden:

```
1 /* empfohlen */
2 @OneToMany(mappedBy = "owner", fetch = FetchType.EAGER)
3 private Set<Portfolio> portfolios = Collections.emptySet();
4 /* problematisch */
5 @OneToMany(mappedBy = "owner", fetch = FetchType.EAGER)
6 private List<Portfolio> portfolios = new ArrayList();
```

6.2.2 Dialog zum Erstellen eines Inhabers

Wird ein Klick auf den Button „Inhaber erstellen“ ausgelöst, wird die Methode *onClickCreateOwner* im Controller des Hauptmenüs ausgelöst, was zur Folge hat, dass mittels der statischen Methode *PrimaryTabManager.loadFxml* der Dialog aus Abbildung 5.6 zum Erstellen eines Inhabers geladen und angezeigt wird.

[6] https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/TreeTableView.html#selectionModelProperty

[7] https://chatgpt.com/share/67b89c48-0a64-8001-bbf4-a4f93d753b46

Dabei wird im Controller des Dialogs (*CreateOwnerDialog*, vgl. Abbildung 5.7) die Methode *initialize* ausgelöst, wodurch die Eingabefelder initialisiert werden. Bei der Initialisierung werden beispielsweise dem Auswahlfeld für „Familienstand“ die verfügbaren Werte übergeben. Darüber hinaus werden auch die Beschränkungen der Eingabefelder wie „Steuersatz“ gesetzt. So wird mittels der *FieldFormatter.setInputFloatRange*-Methode (vgl. Abbildung 5.7) die Eingabe auf einen Wert von 0,00 bis 100,00 eingeschränkt. Da diese Methode eine zentrale Rolle spielt und mit hoher Wahrscheinlichkeit auch in zukünftigen Arbeiten benötigt wird, folgt an dieser Stelle ein Beispiel. Dieses soll veranschaulichen, wie die Methode angewendet wird:

```
1  FieldFormatter.setInputFloatRange(
2      inputTaxRate,
3      0f, 100f,
4      change -> {
5          try {
6              return ownerService.testTaxRatesOrShowError(
7                      FormatUtils.parseFloat(change.getControlNewText()),
8                      FormatUtils.parseFloat(inputChurchTaxRate.getText())
9                              + FormatUtils.parseFloat(inputCapitalGainsTaxRate.
                                   getText())
10                             + FormatUtils.parseFloat(
                                   inputSolidaritySurchargeTaxRate.getText())
11             );
12         } catch (ParseException e) {
13             return false;
14         }
15     }
16 );
```

Der erste Parameter gibt das Textfeld an, auf das die Einschränkung angewendet wird. Anschließend werden die minimal (0,00) und maximal (100,00) zulässigen Werte für die Dezimalzahl-Eingabe festgelegt, wobei das Maximum auch *null* sein kann, sodass nur eine untere Grenze gilt. Der letzte, optionale Parameter ist eine Implementierung der *Predicate<TextFormatter.Change>*-Schnittstelle, mit dem zusätzliche Bedingungen in Form eines Prädikats für die Eingabe definiert werden können. Beispielsweise lässt sich so sicherstellen, dass die Summe ausgewählter Felder 100,00 nicht überschreitet. Die Implementierung der funktionalen Schnittstelle überprüft die Eingabe und erlaubt sie nur, wenn das Prädikat als zusätzliche Bedingung erfüllt ist. Andernfalls wird die Eingabe abgelehnt. Das Prädikat wiederum ist erfüllt, wenn die Eingabe (Zeile 7) nicht größer als die restlich verbleibenden Eingaben ist. Die restlich verbleibende Eingabe resultiert aus der Differenz zwischen der Eingabe und den bereits getätigten Eingaben (Zeile 8-10). Ein Beispiel: angenommen die Eingaben für Kirchen-, Kapitalertrags- und Solidaritätssteuer ergeben in Summe 75,00. Dann wird bei einer Eingabe von 0,00 bis 25,00 diese angenommen. Andernfalls wird der Benutzer über den Konflikt informiert und die Eingabe wird ignoriert.

Mittels *FormatUtils.parseFloat* (vgl. Abbildung 5.7) werden dabei die Eingaben als Dezimaleingaben „geparsed“ und anschließend der Methode *testTaxRatesOrShowError* übergeben. Zum Übersetzen der Benutzereingaben (Zahlen im String-Format) nutzt die Methode die *DecimalFormat*-Klasse

aus dem Paket *java.text.DecimalFormat*.

Möchte der Benutzer den Erstell-Vorgang abbrechen, kann dies über den Button „Abbrechen“ getan werden. Daraufhin wird die Methode *onCancel* ausgelöst. Um das Fenster des Dialogs schließen zu können, kann über ein Element aus der Ansicht (z.B. Eingabefeld für den Vornamen) die Instanz des Fensters durch den Aufruf von getParent().getScene().getWindow() abgerufen werden, die die *hide*-Methode anbietet, mit der der Dialog geschlossen werden kann.

Das Speichern hingegen erfolgt über den Button „Speichern“. Daraufhin werden die Eingaben mittels *FieldValidator.isInputEmpty* (vgl. Abbildung 5.7) überprüft. Diese Klasse dient - analog zur *FieldFormatter*-Klasse - für die zentrale Bereitstellung von Validierungs-Methoden, für eine möglichst hohe Wiederverwendbarkeit. Die Methode nimmt eine beliebige Anzahl an Eingabefeldern und gibt den Wert *true* zurück, wenn eines der Eingabefelder leer (also gar keine Eingabe oder nur Leerzeichen enthalten) ist. Zudem ermöglicht die Methode *decorateField* aus der *PrimaryTabManager*-Klasse einer vergangenen Arbeit die visuelle Hervorhebung fehlerhafter Felder. Hat die Überprüfung ergeben, dass alle Eingabefelder valide sind, werden die Eingaben mittels *OwnerService.writeInput* aus den Eingabefeldern in eine neue *Owner*-Instanz geschrieben, welche anschließend mittels *OwnerService.save* gespeichert wird. Dabei wird der Aufruf an das Repository weitergeleitet und gibt den Wert *true* zurück, wenn die Persistierung erfolgreich war. Wird eine Instanz mit beispielsweise der selben Steuernummer hinterlegt, wird der Benutzer darüber informiert, dass eine Persistierung nicht möglich ist, da jedem Inhaber eine eineindeutige Steuernummer zugeordnet sein muss. Gleiches gilt bei einem unerwarteten Fehler, z. B. wenn die Verbindung zur Datenbank fehlgeschlagen ist. War die Persistierung erfolgreich, wird die *onCancel*-Methode aufgerufen, um den Dialog zu schließen und das Hauptmenü zu aktualisieren. Dabei wird außerdem die *open*-Methode des Hauptmenü-Controllers aufgerufen, um die Daten zu aktualisieren. Zudem erfolgt eine Benachrichtigung über das erfolgreiche Erstellen des Inhabers.

6.2.3 Inhaber-Übersicht

Klickt der Benutzer auf einen bestimmten Inhaber in der Tabelle des Hauptmenüs, wird die Übersicht im selben Fenster angezeigt, welche in Abbildung 5.8 zu sehen ist. Bereits im Kapitel zur Implementierung des Hauptmenüs wurde die dabei verwendete Methode *Navigator.navigateToOwner* eingeführt, jedoch nicht im Detail erläutert. Da das Navigieren im Allgemeinen essenziell in dieser, aber auch zukünftigen, Arbeiten ist, soll daher der Vorgang an dieser Stelle näher erläutert werden. Beim Aufruf dieser Methode wird zunächst *PortfolioManagementTabController.showInhaberTabs* aufgerufen, welche alle Tabs der spezifischen Inhaber-Ansicht anzeigt und die Inhaber-Übersicht öffnet. Dabei wird die Instanz des zu „öffnenden“ Inhabers übergeben, welche den Tabs als Property hinzugefügt wird. Diese Properties erlauben das Hinzufügen (vgl. Listing 6.2) von Einträgen mit einem Schlüssel und seinem Wert - beide Parameter sind vom Typ *Object*, sodass jeder Typ als Schlüssel bzw. Wert gespeichert werden kann.:

```
1 if (!tab.getProperties().containsKey(TAB_PROPERTY_ENTITY)) {
2     // Wenn noch kein Inhaber hinterlegt wurde
3     tab.getProperties().put(TAB_PROPERTY_ENTITY, owner);
```

```
} else {
    // Wenn bereits ein Inhaber in der Property existiert
    tab.getProperties().replace(TAB_PROPERTY_ENTITY, owner);
}
```

Listing 6.2: Übergabe einer Inhaber-Instanz an die inhaber-spezifischen Tabs

Zur Vereinheitlichung wird als Schlüssel der Wert des Attributs *PortfolioManagementTabController.TAB_PROPERTY_ENTITY* (ein *String*) verwendet, über den dann auch die Entität wieder abgefragt werden kann (vgl. Listing 6.3). Anschließend werden die Tabs des spezifischen Inhabers angezeigt und die Übersicht geöffnet.

Während der Initialisierung der Übersicht werden zum Einen die Eingabewerte analog zum Dialog beschränkt und zum Anderen erhält hier jedes Eingabefeld einen Listener, der bei einer Änderung (z. B. ändern des Vornamens „Sven“ zu „Svenja“) die entsprechende Setter-Methode des Inhabers aufruft. Dadurch wird die Änderung sofort an die Inhaber-Klasse übergeben - im Vergleich zum Dialog geschieht dies erst beim Speichervorgang.

Das Laden des in der Property hinterlegten Inhabers geschieht in der *open*-Methode wie folgt:

```
Owner owner = (Owner) portfolioManagementManager // Service-Klasse des
    Portfoliomangements
    .getPortfolioController() // Controller-Klasse des Portfoliomangements
    .getInhaberÜbersichtTab()
    .getProperties()
    .get(PortfolioManagementTabController.TAB_PROPERTY_ENTITY);
owner = ownerService.getOwnerById(owner.getId());
```

Listing 6.3: Abrufen der dem Tab übergebene Inhaber-Instanz

Zunächst wird die Tab-Instanz der Übersicht über den Controller des Portfoliomanagements referenziert, aus dem die hinterlegte Inhaber-Entität über die Properties abgerufen werden kann. Anschließend wird der Inhaber aus der Datenbank neu geladen, um sicherzustellen, dass beim Öffnen stets die aktuellen Daten angezeigt werden. Andernfalls könnten die Daten wie die Portfolios des Inhabers inkonsistent werden, wenn zum Beispiel aus der Übersicht zu einem der dem Inhaber zugeordneten Portfolios navigiert, anschließend bearbeitet (z. B. der Inhaber geändert wird) sowie gespeichert wird und über die Breadcrumbs zurück zum Inhaber navigiert wird.

Die Generierung der Portfolios-Ansicht erfolgt auch in dieser Ansicht mit der *PortfolioTreeView*-Klasse. Die notwendigen Portfolios werden dabei aus der Inhaber-Instanz abgerufen und dem Konstruktor übergeben.

In der *open*-Methode wird außerdem die zu erbende Klasse *EditableView* aus Abbildung 5.9 initialisiert. Dies erfolgt durch den Aufruf der Methode *EditableView.initialize*. Die dabei übergebenen Parameter dienen zum Einen dazu, die Aktionen der Buttons „Ja“ (Änderungen werden gespeichert) sowie „Nein“ (Änderungen werden nicht gespeichert) des Dialogs zu implementieren, und zum Anderen, um auf die jeweiligen Elemente (Navigationsmöglichkeiten, z. B. Breadcrumbbar) zugreifen zu können, welche standardmäßig überwacht werden sollen, um zu erkennen, wann auf die Elemente zugegriffen (angeklickt) wurde. Dazu werden bei den Elementen Listeners registriert.

Um weitere Navigationsmöglichkeiten in die Überwachung einzubeziehen, lassen sich die Methoden *setListeners* bzw. *clearListeners* der Superklasse (*EditableView*) überschreiben. Wichtig ist dabei immer die entsprechende Super-Methode (im gegebenen Beispiel: *super.setListeners*) aufzurufen, um die Standardimplementierung weiterhin zu verwenden. Beim Überschreiben sollen den zusätzlichen Elementen einen Listener übergeben werden, der die *showUnsavedChangesDialog*-Methode der *EditableView*-Klasse aufruft. Wird der Listener ausgelöst, wird dadurch der Dialog angezeigt, sofern es eine Änderung gab. Darüber hinaus wird der *initialize*-Methode eine Entität vom Typ *Changable* (vgl. Abbildung 5.4) übergeben. Mittels der darin definierten Methode *isChanged* können die Listeners abfragen, ob die Inhaberdaten geändert wurden.

Das Entfernen der Listeners geschieht, wenn über eines der zu überwachenden Elemente zu einer anderen Ansicht navigiert wird. Dieser Vorgang ist wichtig, da andernfalls ungewollte Nebeneffekte auftreten können.

An dieser Stelle soll auf eine Problematik im Zusammenhang mit den Listeners hingewiesen werden. Es ist wichtig zu beachten, dass ein Element mehrere Listeners haben kann. Wie bereits aus der Implementierung des Hauptmenüs bekannt, wird bei der Initialisierung der *PortfolioTreeView*-Klasse ein Listener registriert, der beispielsweise den Benutzer zum ausgewählten Portfolio navigiert. Beim Aufruf der *setListeners*-Methode wird ein zweiter Listener hinzugefügt, der bei einem Abbruch des Dialogs (bezüglich einer offenen Änderung) die Ansicht beibehalten soll - also beispielsweise das Navigieren zum ausgewählten Portfolio umgehen soll. Daher muss dieser zweite Listener sicherstellen, dass die Navigation zur Übersicht zurückgeführt wird. Darüber hinaus sollte die Erweiterung dieser Methode mit Bedacht erfolgen. Ein Problem, das beispielsweise auftrat, war das doppelte Anzeigen des Dialogs. Dies wurde durch die Erweiterung der Methode um die Baumansicht verursacht. Beim ersten Aufruf löste der Listener der Baumansicht den Dialog aus. Wird dann zur Portfolio-Ansicht navigiert, wird der Listener der Tableiste des Portfoliomanagements ausgelöst, was dazu führt, dass der Dialog ein zweites Mal angezeigt wird. Um dies zu vermeiden, sollte die Baumansicht nicht zusätzlich eingebunden werden, wodurch sich die Nutzung der *EditableView*-Klasse weiter vereinfacht. Dennoch bleibt die Navigation ein komplexes Thema.

Damit die Eingaben mit den ursprünglichen Werten in *isChanged* verglichen werden können, muss für die jeweiligen Attribute in der Entitäts-Klasse eine Property implementiert werden. Diese Property speichert die Benutzereingaben. Nach dem Laden (*Changable.onPostLoadEntity*) der Entität müssen ihr die aus der Datenbank geladenen Werte zugewiesen werden. Die Getter- und Setter-Methoden sollen mit der Property arbeiten, während die Attribute unverändert bleiben, welche von Hibernate beim Laden der Entität mit Daten gefüllt wurden. Vor dem Speichern oder Aktualisieren der Entität müssen die Werte aus den Properties in die entsprechenden Attribute übernommen werden (*Changable.onPrePersistOrUpdateEntity*). Die Attribute dienen bis dahin zur Abfrage der Originalwerte. Die Aufgabe der *isChanged*-Methode besteht darin, den ursprünglichen Wert des Attributs mit dem aktuellen Wert der zugehörigen Property zu vergleichen. Dadurch ist es wiederum auch möglich das Restoren der Daten zu realisieren, ohne zusätzliche Datenbankabfragen durchführen zu müssen.

Beim Zurücksetzen der Änderungen wird in der Methode *onReset* lediglich die *Changable.restore*-Methode aufgerufen. Sie setzt die Werte der Properties auf ihren ursprünglichen Stand zurück,

indem sie diese mit den Werten aus den Attributen lädt.

Der Löschvorgang (*onRemove*) ruft die *delete*-Methode der Service-Klasse auf. Tritt ein Fehler beim Vorgang auf, wird der Benutzer darüber benachrichtigt und der Vorgang wird beendet. War der Vorgang jedoch erfolgreich, wird die *PortfoliomanagementTabController.navigateBackAfterDeletion*-Methode aufgerufen, die je nach gelöschtem Objekt zum Hauptmenü der entsprechenden Verwaltungskomponente navigiert. Beispielsweise führt das Löschen eines Portfolios zurück zum Hauptmenü der Portfolio-Verwaltung, während das Entfernen eines Inhabers zum Hauptmenü der Inhaber-Verwaltung leitet.

Der Speichervorgang erfolgt vom Ablauf analog zu dem aus dem Dialog: zuerst wird überprüft, dass keine Eingabe leer ist, danach wird der Inhaber in der Datenbank aktualisiert und anschließend erscheint die Meldung, dass das Speichern erfolgreich war.

6.2.4 Inhaber-Portfolios

Für die Darstellung der dem Inhaber zugeordneten Portfolios aus Abbildung 5.10 wird wie gehabt in der *open*-Methode des *OwnerPortfoliosController* aus Abbildung 5.11 zunächst der Inhaber aus der Property des Tabs geladen und anhand seiner Id aus der Datenbank neu geladen. Anschließend erfolgt die Generierung der Portfolios-Tabelle mittels *TableFactory.createOwnerPortfoliosTable* sowie der Konten- und Depots-Tabelle mittels *TableFactory.createOwnerPortfolioAccountTable* bzw. *TableFactory.createOwnerPortfolioDepotTable* (vgl. Abbildung 5.11). Die Parameter diese Methoden sind ähnlich zu denen, die bereits näher in einem vergangenen Abschnitt erläutert wurden. Aus diesem Grund soll hier und in den folgenden Abschnitten auf eine detaillierte Erklärung verzichtet werden.

Den letzten beiden Methoden wird zunächst eine leere Liste übergeben, sodass die Tabellen standardmäßig bereits beim Öffnen sichtbar sind. Erst durch einen Klick auf ein Portfolio werden sie mit den entsprechenden Konten und Depots befüllt. Auch hier wurden der Tabelle entsprechende Aktionen für einfache und doppelte Klicks zugewiesen, analog zum Hauptmenü. Die benötigten Konten und Depots lassen sich direkt aus der Instanz des ausgewählten Portfolios abrufen

6.2.5 Inhaber-Konten

Zur Darstellung der dem Inhaber zugeordneten Konten (vgl. Abbildung 5.12) wird analog in der *open*-Methode der Inhaber aus der Tab-Property geladen. Anschließend erfolgt die Generierung der Konten-Tabelle über die Methode *TableFactory.createOwnerAccountsTable* aus Abbildung 5.13.

Die Erstellung der Tabelle erfolgt im Allgemeinen analog zu den bisherigen Tabellen. Neu ist jedoch die Verwendung der Methode *TableBuilder.addNestedColumn*, um die verschachtelten Spalten, zur Darstellung der Bankdaten des Kontos, der Tabelle zu ermöglichen. Zur einfacheren Handhabung dieser Methode wurde auf eine individuelle Breite der Unterspalten verzichtet. Stattdessen wird die Breite der Unterspalte ins Verhältnis zur Anzahl der Unterspalten gesetzt, sodass alle Unterspalten gleich groß sind. Die Übergabe der zu erstellenden Unterspalten erfolgt durch eine Liste

mit *Map.Entry*-Instanzen. Diese ist durch einen Schlüssel und einen Wert für den Schlüssel gekennzeichnet. Als Schlüssel wird jeweils der Name der Unterspalte übergeben. Als Wert wird hingegen - analog zur bereits bekannten Methode *addColumn* - eine *cellValueFactory* übergeben, mit der der Inhalt der Tabellenzelle bestimmt wird.

6.2.6 Inhaber-Depots

Da das Prinzip hier (vgl. Abbildung 5.14) der Implementierung der Portfolios-Ansicht aus dem Abschnitt 6.2.4 ähnlich ist, soll nicht weiter auf die Implementierung eingegangen werden.

6.2.7 Embold Ausgabe nach finaler Implementierung

Nach der vollständigen Implementierung wurde Embold herangezogen, um den Quellcode bewerten zu lassen, der in Zusammenhang mit der Inhaber-Verwaltung steht. Im Wesentlichen ergab die Untersuchung keine schwerwiegenden Probleme:

Tabelle 6.1: Nennenswerte Ausgaben von Embold im Kontext der Inhaber-Verwaltung

Datei	Tag	Fehler
OwnerRepository.java	Maintainability	„Reports import statements that can be removed. They are either unused, duplicated, or the members they import are already implicitly in scope, because they are in java.lang, or the current package.“
Owner.java	Security	„If there is a method which is declared public, but it returns a reference to a private data structure, which can be modified and create Vulnerability.“
FinancialAsset.java	Understandability	„If an abstract class does not provides any methods, it may be acting as a simple data container that is not meant to be instantiated. In this case, it is probably better to use a private or protected constructor in order to prevent instantiation than make the class misleadingly abstract.“
OwnerOverviewController	Robustness	„A field declared inside the spring component should be thread safe. By default, beans in spring are singleton. Multiple threads accessing the same component may produce inconsistent results, if they access fields declared globally.“

Die einzige relevante Meldung bezieht sich auf die Rückgabe privater Objekte über öffentliche Methoden. Konkret betrifft es den Zugriff auf die Adresse innerhalb der Inhaber-Klasse. Würde das Adressattribut öffentlich zugänglich gemacht werden, bestünde die Gefahr, dass es mit *null* überschrieben wird. Dies würde jedoch eine logische Inkonsistenz erzeugen, da ein Inhaber stets eine Adresse haben muss.

6.3 Portfolio-Verwaltung

6.3.1 Hauptmenü

Das Hauptmenü der Portfolio-Verwaltung aus Abbildung 5.16 folgt dem gleichen Prinzip wie das Menü „Portfolios“ in der Inhaber-Verwaltung (siehe Kapitel 6.2.4).

Beim Öffnen des Tabs „Portfolios“ im Portfoliomanagement werden die Daten über die Methode *PortfolioService.getAll* abgerufen und anschließend in der Methode *open* dargestellt. Die Tabellen werden entsprechend der in Kapitel 6.2.4 beschriebenen Vorgehensweise erstellt und angezeigt.

Die Portfolio-Tabelle wird mit der Methode *TableFactory.createPortfolioTable* generiert, die sich nur geringfügig von der Portfolio-Tabelle der Inhaber-Verwaltung unterscheidet. Eine Neuerung ist hier die Spalte „Inhaber“ sowie die Möglichkeit, Portfolios direkt in der Tabelle zu löschen bzw. zu aktivieren oder zu deaktivieren. Die Konten- und Depot-Tabellen werden durch die Methoden *createOwnerPortfolioAccountTable* und *createOwnerPortfolioDepotTable* erstellt, die ebenfalls aus der Inhaber-Verwaltung übernommen wurden (vgl. Abbildung 5.11).

6.3.2 Dialog zum Erstellen eines Portfolios

Ähnlich wie bei der Inhaber-Verwaltung wird der Dialog aus Abbildung 5.18 über die Methode *onClickCreatePortfolio* im *PortfolioController* aus Abbildung 5.19 angezeigt. Zunächst wird geprüft, ob registrierte Inhaber vorhanden sind, die dem neuen Portfolio zugeordnet werden können. Sind keine Inhaber vorhanden, ist die Erstellung eines Portfolios nicht möglich, und der Benutzer wird entsprechend durch eine Mitteilung informiert. Im Unterschied zur Inhaber-Verwaltung muss hier jedoch zum Öffnen der Dialogansicht zusätzlich die Methode *open* (aus der Schnittstelle *Openable*) aufgerufen werden. Dies erfolgt aus dem Grund, dass dieser Dialog zur Behebung von Datenbank-Inkonsistenzen wiederverwendet wird. Beide Vorgänge (Erstellen von Portfolios bzw. beheben von Portfolio-Inkonsistenzen) müssen die Inhaber-Daten allerdings auf unterschiedliche Weise laden. Daher muss der Initialisierungsvorgang für beide Vorgänge allgemeingültig gehalten sein. Andernfalls hätte, wie in der Inhaber-Verwaltung, das Abrufen der Inhaber während der Initialisierung (*initialize*) erfolgen können.

Bei der Portfolio-Verwaltung gibt es eine Besonderheit hinsichtlich des Dialogs. Wie bereits erläutert, wird erst beim Speichern beispielsweise eines Inhabers eine neue Inhaber-Instanz erstellt, der dann die Eingaben aus den Eingabefeldern übergeben wird. Allerdings benötigt die Tabelle der Anlagenrichtlinie bereits beim Öffnen des Dialogs eine Portfolio-Instanz, da diese die Grundlage für die Einträge (*Entry*, siehe Abbildung 5.20) der Anlagenrichtlinie bildet. Daher wird bereits beim Aufruf der *open*-Methode im Dialog eine neue Portfolio-Instanz erstellt und anschließend die Einträge der Anlagenrichtlinie durch den Aufruf von *InvestmentGuideline.initializeEntries* initialisiert. Damit nun die Darstellung der Tabellen erfolgen kann, wird anschließend die *initialize*-Methode des Dialogs aufgerufen, die speziell für diesen Fall die Generierung der Tabellen übernimmt. Dieser Ansatz mag ungewöhnlich erscheinen, da die *initialize*-Methode der Initialisierung beim Laden des Dialogs vorbehalten ist. Dieser Workaround ist jedoch notwendig, um den Dialog für die Behandlung von Inkonsistenzen wiederverwenden zu können.

Beim Initialisieren der Anlagenrichtlinie über die *initializeEntries*-Methode werden die Einträge (*Entry*) entsprechend einer hierarchischen Baumstruktur erzeugt. Dazu wird für jedes übergeordnete Literal des Enums *InvestmentType* aus Abbildung 5.20 ein Eintrag erzeugt. Wenn ein Literal untergeordnete Literale besitzt, werden die untergeordneten Einträge erzeugt und dem übergeordneten zugeordnet.

Zur Realisierung der hierarchischen Struktur innerhalb des Enums (*InvestmentType*) übernimmt der Konstruktor des Enums drei Parameter: einen String, der die Textdarstellung des Literals definiert, einen booleschen Wert, der angibt, ob es sich um ein übergeordnetes Element handelt, und eine optionale Liste von *InvestmentType*-Literalen, die die untergeordneten Elemente repräsentieren. Das Listing 6.4 zeigt ein Beispiel für die hierarchische Struktur des Typs „Aktien- & Aktienfonds“. Dort wird ersichtlich, dass das Literal *SINGLE_STOCK* ein untergeordnetes Literal ist, da der zugehörige boolesche Wert *false* ist. Im Gegensatz dazu markiert der Wert *true* beim Literal *EQUITY_AND_EQUITYFUNDS* dieses als übergeordnetes. Zudem enthält dieses Literal eine Liste weiterer Literale, die die untergeordneten Literale darstellen.

```
1 SINGLE_STOCK("Einzelaktien", false),
2 EQUITY_ETF("Aktien-ETF", false),
3 ACTIVE_EQUITYFUNDS("Aktive Aktienfonds", false),
4 EQUITY_AND_EQUITYFUNDS("Aktien- & Aktienfonds", true, SINGLE_STOCK, EQUITY_ETF,
      ACTIVE_EQUITYFUNDS),
```

Listing 6.4: Erzeugung der hierarischen Struktur im Enum *InvestmentType* für „Aktien- Aktienfonds“

Ein weiterer Punkt ist die Unterstützung durch die Auto-Complete-Funktion von GitHub Copilot, die die Implementierung der Literale erheblich beschleunigte. Sie schlug größtenteils die korrekten Literale einschließlich der passenden Parameter vor und erkannte, welche Literale einem Literal untergeordneten werden müssen. So erkannte Copilot beispielsweise, dass *SINGLE_STOCK*, *EQUITY_ETF* und *ACTIVE_EQUITYFUNDS* dem Literal *EQUITY_AND_EQUITYFUNDS* zugeordnet werden müssen.

Die Erzeugung der Anlagenrichtlinien-Tabelle erfolgt durch den Konstruktor-Aufruf der *InvestmentGuidelineTable*-Klasse aus Abbildung 5.21. Im Konstruktor werden die Tabellenspalten erzeugt. Dabei wird die Methode *createStaticColumn* beispielsweise für die Spalte „Anlagetyp“ und *createDynamicColumn* für die Spalte „Aufteilung Gesamtvermögen“ verwendet.

Die Methode *createStaticColumn* dient der Erstellung von Spalten mit statischem Zellinhalt. Dazu werden der Methode der Spaltenname sowie eine *cellValueFactory* übergeben, die den Zellinhalt analog zur *TableBuilder*-Klasse definiert.

Mit der Methode *createDynamicColumn* lassen sich hingegen bearbeitbare Spalten erstellen. Diese Methode ist in zwei Varianten verfügbar, ähnlich wie die *FieldFormatter.setInputFloatRange*-Methode. Beide Varianten nutzen die folgenden Parameter:

- Spaltenname und -beschreibung sowie eine *cellValueFactory*: Definieren die Spalte und ihren Inhalt.
- *onCommit*: Legt eine Aktion fest, die beim Bestätigen einer Eingabe ausgeführt wird, z. B. den Aufruf von *Entry.setMaxRiskclass*. Dadurch wird die Eingabe in den entsprechenden Eintrag weitergeleitet.
- *inputFormatter*: Gibt das Eingabeformat vor, z. B. durch Verwendung von *FieldFormatter.setInputFloatRange*.

- Boolescher Wert *showOnlyParent*: Dieser bestimmt, ob nur für übergeordnete oder auch untergeordnete Einträge der Wert innerhalb der Spalte angezeigt und bearbeitet werden kann. Dadurch konnte beispielsweise realisiert werden, dass in der Spalte der maximalen Risikoklasse nur Eingaben für den Anlagentyp „Aktien & Aktienfonds“, aber nicht für „Einzelaktien“ möglich sind (vgl. Abbildung 5.18).

Der Unterschied zwischen den beiden Varianten besteht in dem zusätzlichen Parameter *showOnlyIfTrue*, vom bereits bekannten Typ *Predicate*. Dieser erlaubt es, individuell festzulegen, für welche Literale aus dem Enum *InvestmentType* die Zellinhalte innerhalb der Spalte dargestellt werden sollen. Dadurch wird beispielsweise verhindert, dass ein Wert in der Spalte „Maximale Risikoklasse“ für den Anlagetyp „Liquidität“ angezeigt bzw. bearbeitet werden kann.

Zudem sei hier auf die Implementierung der mit dem *showOnlyParent*-Parameter verknüpften Funktionalität näher einzugehen. Ursprünglich wurde in Erwägung gezogen, die *cellFactory*[8] so zu gestalten, dass bei einem Parameter-Wert von *true* keine Zellen für untergeordnete Einträge erstellt werden. Dies hätte jedoch erfordert, dass der Anlagentyp der jeweiligen Zeile, in der die die Zelle bereitgestellt wird, abgefragt werden kann. Daher wurde stattdessen beschlossen, die Zellwerte lediglich auszublenden und den Bearbeitungsmodus für diese Einträge zu deaktivieren. Im Folgenden wird erläutert wie dies realisiert wurde.

- **Verdecken der Werte:** Die zu erstellende Spalte erhält eine Standardimplementierung der *cellValueFactory* (vgl. Listing 6.5). Wenn es sich bei dem Zellobjekt (*Entry*) um einen untergeordneten Eintrag handelt oder der Parameter *showOnlyIfTrue false* zurückgibt, bleibt die Zelle ohne Inhalt. Andernfalls wird die Anfrage an den *cellValueFactory*-Parameter weitergeleitet, der den eigentlichen Zellinhalt definiert.

```
1  newColumn.setCellValueFactory(cell -> {
2          InvestmentGuideline.Entry entry;
3          // Wenn showOnlyParent==true: untergeordnete Einträge werden
               als leere Zelle gerendert.
4          if (showOnlyParent && entry.getType().isChild()) return null;
5          // Wenn das in der Predicate-Instanz definierte Prädikat
               unwahr ist, bleibt der Zellinhalt leer.
6          if (showOnlyIfTrue != null && !showOnlyIfTrue.test(entry.
               getType())) return null;
7          // Rufe die als Parameter übergebene cellValueFactory auf, um
               den Zellinhalt abzufragen.
8          return cellValueFactory.call(entry);
9      });
```

Listing 6.5: Standardimplementierung der *cellValueFactory* der Anlagenrichtlinien-Tabelle

- **Ignorieren von Eingaben:** Beim Übergang in den Bearbeitungsmodus der Zelle werden die gleichen Bedingungen wie in der *cellValueFactory* geprüft. Sind diese nicht erfüllt, wird der Bearbeitungsmodus direkt verlassen, wodurch ein Bearbeitungsversuch verhindert wird.

[8] Ist eine Rückruffunktion, die zur Erzeugung der Tabellenzellen innerhalb einer Spalte genutzt wird.

Nachdem die Spalten erstellt wurden, erfolgt im Konstruktor der *InvestmentGuidelineTable*-Klasse die Initialisierung der Tabelleneinträge durch den Aufruf der Methode *initializeItems*. Diese Methode funktioniert ähnlich wie *InvestmentGuideline.initializeEntries*, konzentriert sich jedoch ausschließlich auf die Initialisierung der Tabelleneinträge, basierend auf den Einträgen einer Anlagenrichtlinie.

Mittels der *TableBuilder.addEditableColumn*-Methode können außerdem Spalten mit bearbeitbaren Zellen zu einer Tabelle hinzugefügt werden. Sie wird unter anderem genutzt, um die Tabellen zur Aufteilung des Gesamtvermögens nach Land bzw. Währung zu erzeugen, welche über die Methoden *createPortfolioDivisionByLocationTable* bzw. *createPortfolioDivisionByCurrencyTable* aus der Klasse *TableFactory* generiert werden. (vgl. Abbildung 5.21)

Nachdem die Funktionsweise der Tabellen erläutert wurde, soll abschließend kurz auf das Speichern und Abbrechen eingegangen werden. Die entsprechenden Methoden arbeiten grundsätzlich ähnlich zu denen in der Inhaber-Verwaltung. Eine Ausnahme bildet jedoch die Validierung beim Speichern. Anstelle der Klasse *FieldFormatter* wird hierfür die *PortfolioService.isInputInvalid*-Methode verwendet. Diese Methode überprüft nicht nur, ob leere Eingaben vorliegen, sondern stellt auch sicher, dass:

- die Aufteilung des Gesamtvermögens bei über- und untergeordneten Einträgen jeweils eine Summe von 100 ergibt, und
- die Tabellen zur Aufteilung nach Ländern bzw. Währungen ebenfalls eine Gesamtsumme von 100 aufweisen.

Falls eine dieser Bedingungen nicht erfüllt ist, wird dem Benutzer eine entsprechende Mitteilung angezeigt, die ihn darauf hinweist, und der Speicher-Vorgang wird abgebrochen.

6.3.3 Portfolio-Übersicht

Analog zur Übersicht eines Inhabers wird auch hier (siehe Abbildung 5.22) das dem Tab übergebene Portfolio (analog zu Listing 6.3) beim Öffnen geladen und dessen Daten dargestellt und anschließend die Superklasse *EditableView* initialisiert. Das Löschen und Zurücksetzen erfolgt ähnlich wie in der Übersicht der Inhaber-Verwaltung. Der Speichervorgang erfolgt wie im Dialog beschrieben.

Wie in der Inhaber-Übersicht mussten auch hier Listeners für die Eingabefelder implementiert werden, um den Benutzer über ungespeicherte Änderungen zu benachrichtigen, wenn er zu einer anderen Ansicht wechselt. Dabei trat jedoch ein Problem auf: Nach erneutem „Öffnen eines Portfolios“ wurde der Inhaber nicht mehr korrekt angezeigt. Jedes Mal, wenn die Übersicht geöffnet wird, werden die Inhaber aus der Datenbank geladen und dem Benutzer zur Auswahl bereitgestellt. Beim ersten Öffnen der Übersicht wird dem Auswahlfeld für den Inhaber die initialen Werte übergeben. Bei jedem weiteren Laden werden die ursprünglichen Werte entfernt und durch die neu geladenen Inhaber ersetzt. Dadurch wird jedoch der Listener des Auswahlfelds ausgelöst, da das Entfernen der alten Werte dazu führte, dass die Auswahl kurzzeitig auf *null* gesetzt wird. Daraufhin ruft der Listener die Setter-Methode auf, wodurch der Inhaber des Portfolios auf *null* gesetzt wird. Beim

anschließenden Laden der Inhaber-Daten wurde der Inhaber aus dem Portfolio abgerufen, um ihn im Auswahlfeld vorauszuwählen – da dieser jedoch auf *null* gesetzt wurde, blieb das Auswahlfeld ebenfalls auf *null*. Die Lösung bestand darin, eine Bedingung im Listener zu ergänzen, die sicherstellt, dass der Inhaber des Portfolios nur dann geändert wird, wenn der neue Wert nicht *null* ist.

Zuletzt sei auf die potenzielle Notwendigkeit der *equals*-Methode in Entitäts-Klassen hingewiesen. Ein konkretes Problem trat beim Zurücksetzen eines Portfolios auf, da auch hierbei kein Inhaber mehr im Auswahlfeld ausgewählt wurde. In Listing 6.6 ist der Code-Ausschnitt dargestellt, mit dem das Auswahlfeld für die Inhaber initialisiert wird. Dabei wird deutlich, warum der Code ohne Anpassung nicht wie gewünscht funktioniert: Woher soll das *SelectionModel* des Auswahlfeldes beim Aufruf von *select* wissen, welche Instanz aus der Liste `ownerService.getAll()` der Instanz `portfolio.getOwner()` entspricht?

```
1 inputOwner.getItems().setAll(ownerService.getAll());
2 inputOwner.getSelectionModel().select(portfolio.getOwner());
```

Listing 6.6: Initialisierung der Auswahlbax mit Inhabern

Laut der Dokumentation zur *SelectionModel*-Klasse, die von Auswahlfeldern verwendet wird, wird beim *select*-Aufruf wie folgt verfahren: Es wird über die zugrunde liegenden Daten iteriert, um zu prüfen, ob sich ein Element mit dem übergebenen Element gleicht. Diese Prüfung erfolgt mit Hilfe der *equals*-Methode. Ohne eine benutzerdefinierte Implementierung verwendet die Standardimplementierung von *equals* die Objektidentität, d.h., sie prüft, ob beide Referenzen auf dasselbe Objekt im Speicher zeigen (vgl. [bae]). Die Lösung bestand daher darin, die *equals*-Methode in der Inhaber-Klasse zu überschreiben, sodass die Instanzen zweier Inhaber als gleich gelten, wenn der Primärschlüssel beider Instanzen identisch ist.

6.3.4 Embold Ausgabe nach finaler Implementierung

Nach der vollständigen Implementierung wurde auch hier Embold wieder herangezogen, um den Quellcode bewerten zu lassen, der in Zusammenhang mit der Portfolio-Verwaltung steht.

Insgesamt konnten keine gravierenden Probleme entdeckt werden (vgl. Tabelle 6.2). So wurde in der *InvestmentGuidelineTable*-Klasse fälschlicherweise eine private Methode (*createDynamicColumn*) als ungenutzt gemeldet, obwohl sie mehrfach im Konstruktor der Klasse aufgerufen wird. Zudem kritisiert Embold die direkte Verwendung des Literals „100“ in der Bedingung `if (sum != 100)`. Die Einführung einer zusätzlichen Variablen für das Literal ist jedoch aus logischer Sicht nicht notwendig.

Tabelle 6.2: Nennenswerte Ausgaben von Embold im Kontext der Portfolio-Verwaltung

Datei	Tag	Fehler
InvestmentGuideline.java	Understandability	„Identifies private felds whose values never change once they are initialized either in the declaration of the field or by a constructor. This helps in converting existing classes to becoming immutable ones.“
InvestmentGuidelineTable.java	Understandability	„Unused Private Method detects when a private method is declared but is unused.“
PortfolioService.java	Understandability	„Avoid using hard-coded literals in conditional statements. By declaring them as static variables or private members with descriptive names maintainability is enhanced. By default, the literals -1 and 0 are ignored.“

6.4 Konto-Verwaltung

6.4.1 Hauptmenü

Die Implementierung des Hauptmenüs aus Abbildung 5.24 orientiert sich strukturell an die Hauptmenüs der vergangenen Verwaltungskomponenten.

Zum Abrufen der mit einem Konto verknüpften Depots (bei einem Verrechnungskonto) existiert eine bidirektionale Beziehung zwischen der Konto- und Depot-Entität (vgl. Listing 6.7). In der Depot-Klasse wird dazu eine ManyToMany-Beziehung zwischen der Depot- und Konto-Entität definiert. Die *JoinColumn*-Annotation legt dabei den Namen der Fremdschlüsselspalte in der Depot-Tabelle fest, die auf die Konto-Tabelle verweist. In der Konto-Klasse wird ebenfalls die Beziehung hinterlegt, jedoch mit dem *mappedBy*-Parameter. Dieser gibt den Namen des Attributs (*billingAccounts*) aus der Depot-Klasse an, dem die Beziehung gehört. Beim Abrufen eines Kontos kann Hibernate so die zugehörigen Depots über die Beziehung aus der Depot-Tabelle laden.

```
1 // Attribut in der 'Konto'-Klasse
2 @ManyToMany(mappedBy = "billingAccounts")
3 private Set<Depot> mappedDepots;
4
5 // Attribut in der 'Depot'-Klasse
6 @ManyToMany
7 @JoinColumn(name = "billing_account_id", nullable = false)
8 private List<Account> billingAccounts;
```

Listing 6.7: Abrufen aller Depots, die ein bestimmtes Konto als Verrechnungskonto führen

Ein weiterer wichtiger Aspekt ist die Umrechnung beliebiger Währungen in Euro, die insbesondere für die Implementierung der Methode *getValue* (aus der Schnittstelle *Valuable*) zur Umrechnung des Kontostands relevant ist. Hierfür wird die Methode *AccountService.getLatestExchangeCourse* verwendet. Sie erhält eine Währungsinstanz (*Currency*) als Eingabe und gibt den neuesten Wechselkurseintrag (unter „Daten“>„Wechselkurs“ in SmartWM) für diese Währung in Euro zurück. Dabei wird nur der aktuellste nicht-leere Eintrag berücksichtigt. Dieser kann leer sein, wenn mehrere Wechselkurs-Spalten existieren und im letzten Eintrag nicht alle Wechselkurse eingetragen wurden.

Ein Beispiel: Falls Wechselkurs-Spalten für „EUR_USD“ und „EUR_GBP“ existieren und ein neuer Kurseintrag nur für „EUR_USD“ erstellt wird, bleibt der Eintrag für „EUR_GBP“ leer.

Ist die übergebene Währung bereits Euro, gibt die Methode den Wechselkurs „1“ zurück. Zur Berechnung des Konto-Werts in Euro wird der Kontostand durch den ermittelten Wechselkurs geteilt.

Ursprünglich war vorgesehen, diese Methode im Repository (*AccountRepository*) statt in der Service-Klasse (*AccountService*) zu implementieren. Allerdings unterstützt SQL keine dynamischen Platzhalter für Spaltennamen. Um dennoch eine direkte Datenbankabfrage aus der Service-Klasse heraus durchführen zu können, wurde von ChatGPT die Verwendung der *JdbcTemplate*-Klasse vorgeschlagen.[9]

6.4.2 Dialog zum Erstellen von Konten

Der Dialog in Abbildung 5.26 funktioniert ähnlich wie der aus der Portfolio-Verwaltung: Er wird über die *onClickCreateAccount*-Methode im Controller des Hauptmenüs aus Abbildung 5.27 aufgerufen. Vor der Anzeige erfolgt eine Überprüfung, ob bereits Inhaber und Portfolios vorhanden sind, denen das neue Konto zugeordnet werden kann. Falls keine vorhanden sind, wird die Konto-Erstellung verhindert und der Benutzer erhält eine entsprechende Benachrichtigung. Zudem implementiert der Dialog analog die *Openable*-Schnittstelle.

Bei der Initialisierung des Dialogs werden die Eingabefelder mit Hilfe der *FieldFormatter*-Klasse eingeschränkt:

- *setInputOnlyDecimalNumbers*: beschränkt beispielsweise den Kontostand auf beliebige Dezimalzahlen
- *setInputFloatRange*: begrenzt den Zinssatz auf gültige Werte (0,00 bis 100,00) und
- *setInputIntRange*: legt den zulässigen Bereich für die Zinstage (0 bis 366) fest.

Anschließend werden Auswahlfelder, wie etwa für die Währung, initialisiert.

Für die Währungsoptionen kommt die Methode *getAvailableCurrencies* aus der Klasse *Currency* (*java.util*) zum Einsatz, die eine Liste aller verfügbaren Währungen zurückgibt (siehe Listing 6.8). Die Sortierung erfolgt alphabetisch mit Hilfe von Comparator.comparing(Currency::getCurrencyCode), sodass die Währungen anhand ihres Währungscodes in aufsteigender Reihenfolge sortiert werden.

```
1 Currency.getAvailableCurrencies().stream()
2     .sorted(Comparator.comparing(Currency::getCurrencyCode))
3     .toList()
```

Listing 6.8: Abruf und alphabetische Sortierung aller verfügbaren Währungen

Beim anschließenden Aufruf der *open*-Methode werden alle aktuellen Inhaber und Portfolios aus der Datenbank geladen und dem Nutzer zur Auswahl angeboten.

Die Aktionen für das Abbrechen (*onCancel*) und Speichern (*onSave*) entsprechen denen aus dem Dialog zur Inhaber-Erstellung, sodass hier keine weiteren Details erläutert werden. Es sei jedoch

[9] https://chatgpt.com/share/67586d00-d838-8001-8c7d-ae27400cc619

erwähnt, dass beim Speichern geprüft wird, ob für die gewählte Währung mindestens ein Wechselkurseintrag existiert. Falls kein Eintrag vorhanden ist, kann das Konto nicht erstellt werden, und der Nutzer wird entsprechend darauf hingewiesen.

6.4.3 Konto-Übersicht

Viele grundlegende Funktionen dieser Übersicht (vgl. Abbildung 5.28) – wie das Laden der Daten, das Zurücksetzen und das Speichern – entsprechen der Implementierung der zuvor beschriebenen Verwaltungsfunktionen.

Für die Darstellung der Depots, bei denen das Konto als Verrechnungskonto hinterlegt ist, war jedoch ein spezieller Ansatz erforderlich. Hierfür wird die Klasse *PortfolioTreeView* wiederverwendet. Damit die Anzeige korrekt funktioniert, musste die Portfolio-Instanz des Kontos so angepasst werden, dass alle Konten innerhalb des Portfolios ausgeblendet werden. Gleichzeitig werden die gespeicherten Depots so gefiltert, dass nur jene angezeigt werden, die dieses Konto als Verrechnungskonto referenzieren. Dementsprechend wurde im Controller (*AccountOverviewController*, siehe Abbildung 5.29) folgende Hilfsmethode implementiert:

```
1 private void prepareTableData() {
2     Portfolio portfolio = account.getPortfolio();
3     // Entfernt alle Konten aus dem Portfolio
4     portfolio.setAccounts(List.of());
5     // Zeigt nur die Depots an, die mit dem Konto verknüpft sind
6     portfolio.setDepots(account.getMappedDepots());
7 }
```

Listing 6.9: Anpassung der Portfolio-Daten für die Darstellung der zugeordneten Depots

Das hätte allerdings zur Folge, dass der Wert der Portfolios nicht mehr korrekt ermittelt werden könnte. Aus diesem Grund wurde der *PortfolioTreeView*-Klasse ein zweiter Konstruktor implementiert, der den zusätzlichen Parameter *disableShowPortfolioSum* ermöglicht. Dieser ermöglicht es die Darstellung des Werts für Portfolios zu deaktivieren, sodass dieser beispielsweise nur für die einzelnen Depots angezeigt wird.

6.4.4 Embold Ausgabe nach finaler Implementierung

Nach der Implementierung der Konto-Verwaltung konnten mittels Embold keine neuen, unbekannten bzw. gravierende Probleme aufgedeckt werden, sodass hier auf Details verzichtet werden soll.

6.5 Vermeidung von Datenbank-Inkonsistenzen

6.5.1 Vorbeugende Maßnahmen

Wie bereits im Konzept erwähnt, umfasst die Prävention von Inkonsistenzen sowohl die Validierung und Beschränkung von Benutzereingaben als auch die korrekte Konfiguration von Spalten

und Beziehungen der Entitäten.

Da die Validierung bereits in verschiedenen Implementierungen und Konzeptionsphasen behandelt wurde, liegt der Fokus in den folgenden Abschnitten auf der Konfiguration der Spalten und Beziehungen.

Sicherstellen, dass Spalteninhalte nicht leer sind

Bestimmte Felder, wie der Vorname eines Inhabers, dürfen nicht leer sein. Diese Anforderung lässt sich mit dem Parameter *nullable* realisieren (vgl. Listing 6.10). Wird versucht, einen Inhaber mit einem *null*-Wert im Vornamen zu speichern, wirft Hibernate einen Fehler aus und bricht den Speichervorgang ab.

```
1 @Column(name = "forename", nullable = false)
2 private String forename;
```

Listing 6.10: Konfiguration einer Datenbankspalte als nicht leer

Der Parameter *nullable* kann auch in der Annotation *JoinColumn* verwendet werden, um Fremdschlüssel als nicht optional zu deklarieren.

Zusätzlich gibt es den Parameter *optional* für Beziehungs-Annotationen (z. B. *OneToOne*), der eine ähnliche Funktion erfüllt. Während *nullable* direkt die Datenbankspalte beeinflusst, wird *optional* bereits auf Anwendungsebene von Hibernate durchgesetzt. Dadurch können Fehler frühzeitig erkannt werden, noch bevor ein Datenbankzugriff erfolgt.

Kaskadierung von Datenbank-Operationen

Bei der Nutzung von Beziehungen ist eine korrekte Kaskadierung essenziell, um Inkonsistenzen zu vermeiden. Andernfalls könnte beispielsweise ein Inhaber gelöscht werden, während seine zugehörigen Portfolios weiterhin in der Datenbank verbleiben, was zu fehlerhaften Zuständen führt. Das Listing 6.11 zeigt eine Konfiguration, bei der sämtliche Datenbankoperationen eines Inhabers automatisch auf dessen Adress-Entität übertragen werden.

```
1 @OneToOne(cascade = CascadeType.ALL, orphanRemoval = true)
2 @JoinColumn(name = "address_id")
3 private Address address;
```

Listing 6.11: Beispiel für die Kaskadierung von Datenbank-Operationen

Zusätzlich sollte der Parameter *orphanRemoval* in Betracht gezogen werden. Dieser sorgt dafür, dass verwaiste Entitäten automatisch gelöscht werden. Eine Entität gilt als verwaist, wenn ihre Referenz durch eine neue ersetzt wird und der Inhaber anschließend mit der aktualisierten Adresse gespeichert wird.

Wahrung von Einzigartigkeiten

Um sicherzustellen, dass eine Spalte keine Duplikate enthält, muss der Parameter *unique* in der *Column*-Annotation auf *true* gesetzt werden. Dies ist beispielsweise bei Portfolios relevant, da der Name eines Portfolios eindeutig ist und nicht mehreren Portfolios zugeordnet werden darf. Zwar könnte die Einzigartigkeit auch über einen Primärschlüssel erzwungen werden, jedoch wäre dieser nicht änderbar. Dadurch wäre es jedoch nicht möglich, den Namen eines Portfolios nachträglich zu bearbeiten. Die Verwendung des *unique*-Parameters bietet daher eine flexible Lösung.

6.5.2 Ablauf

Konkret für Inhaber

Das Überprüfen auf Inkonsistenzen erfolgt über die *OwnerRepository.inconsistentOwnerExists*-Methode. Wurden Inkonsistenzen entdeckt, wird aus dem Repository die *getInconsistentOwnerIds*-Methode aufgerufen, um die IDs der inkonsistenten Inhaber zu sammeln.

Im Detail ruft die *getInconsistentOwnerIds*-Methode verschiedene andere Methoden des Repositories auf, die für die eigentliche Prüfung auf Inkonsistenzen zuständig sind. Beispielsweise wird die Methode *findAllByForenameIsNullOrAfternameIsNullOrCreatedAtIsNull* ausgeführt, welche die IDs der Inhaber zurückgibt, deren Vorname oder Nachname leer ist, da diese Spalten in der Entität mit *nullable=false* als nicht-leer gekennzeichnet sind.

Allgemein wird empfohlen, bei der Implementierung zur Identifikation und Erfassung der IDs inkonsistenter Entitäten (z. B. Transaktionen für zukünftige Erweiterungen) der Struktur in Listing 6.12 zu folgen. Diese sieht vor, Methoden innerhalb der Repository-Schnittstelle zu definieren, die spezifische Inkonsistenz-Typen abfragen (z.B. das Prüfen auf leere Felder, die als nicht-null annotiert sind). Diese Methoden sollen dann von einer zentralen Methode verwendet werden, um inkonsistente Inhaber zu ermitteln, wodurch die Lesbarkeit und Wartbarkeit der Implementierung zur Ermittlung aller inkonsistenten Entitäten erheblich verbessert wird.

```
1 default Set getInconsistentOwnerIds() {
2 Set inconsistentOwnerIds = new HashSet<>();
3 inconsistentOwnerIds.addAll(findAllByAddressOrTaxInformationIsInvalid());
4 inconsistentOwnerIds.addAll(
      findAllByForenameIsNullOrAfternameIsNullOrCreatedAtIsNull());
5 /* ... */
6 return inconsistentOwnerIds;
7 }
```

Listing 6.12: Vereinfachte Darstellung der Methode zur Identifikation inkonsistenter Inhaber

Das Rekonstruieren von Inhabern erfolgt über die Methode *reconstructOwner* des Repositories. Wie in Listing 6.13 dargestellt, wird zunächst eine neue Inhaber-Instanz erzeugt. Anschließend werden die Daten des Inhabers durch gezielte Abfragen vervollständigt. Beispielsweise erfolgt eine Abfrage zur Ermittlung des Vor- und Nachnamens. Sollte für die ID des Inhabers kein Vorname gefunden werden (weil kein passender Datensatz existiert), wird der Wert auf *null* gesetzt.

```
default Owner reconstructOwner(Long id) {
    Owner reconstructedOwner = new Owner();
    reconstructedOwner.setId(id);
    reconstructedOwner.setForename(findForenameById(id).orElse(null));
    reconstructedOwner.setAftername(findAfternameById(id).orElse(null));

    Optional<Long> addressId = findAddressIdByOwnerId(id);
    if (addressId.isPresent()) {
        Owner.Address address = reconstructedOwner.getAddress();
        address.setId(addressId.get());
        address.setCountry(findCountryByAddressId(addressId.get()).orElse(null)
            );
    }
    return reconstructedOwner;
```

Listing 6.13: Abstrakter Überblick zum Ablauf des Rekonstruierens eines Inhabers anhand seiner Id

Nach der Rekonstruktion des Inhabers können seine Parameter über den Dialog (*FixOwnerInconsistenciesDialog*) angezeigt werden. Dieser leitet sich aus dem Dialog zum Erstellen von Inhabern *CreateOwnerDialog* ab, da die Dialogstruktur zur Behebung von Inkonsistenzen weitgehend derjenigen zur Erstellung eines Inhabers entspricht. Durch diese Vererbung wird verhindert, dass die gleichen Attribute und Eingabefelder erneut definiert werden müssen. Zudem kann die Initialisierung der Eingabefelder (z.B. Einschränkungen der Eingaben) übernommen werden. Allerdings musste die Initialisierung zum Beispiel für die Status-Eingabe erweitert werden. Um fehlerhafte Daten zu kennzeichnen, wird die bereits bekannte Methode *FieldValidator.isInputEmpty* genutzt, sowie *ComboBoxValidator.areComboboxInputsValid*. Letztere überprüft, ob in allen Auswahlfeldern eine Auswahl getroffen wurde – falls nicht, wird das entsprechende Feld rot markiert.

Auch der Speichervorgang musste überschrieben werden, um sicherzustellen, dass in allen Auswahlfeldern eine gültige Auswahl vorhanden ist. Falls beispielsweise beim Rekonstruieren der Status des Inhabers nicht geladen werden konnte, wird im Auswahlfeld der Wert *null* ausgewählt, was einer fehlenden Auswahl entspricht.
Die Persistierung erfolgt konkret durch den Aufruf von *OwnerService.updateOwnerNatively*. Dabei werden die Datensätze des Inhabers gezielt über native SQL-Befehle aktualisiert oder neu gespeichert, beispielsweise wenn der Fremdschlüssel zur Inhaber-Adresse ungültig ist. Eine alternative Vorgehensweise wäre, den Inhaber zunächst vollständig zu löschen und anschließend mit Hibernate erneut zu speichern, was zwar einfacher wäre, aber unter Umständen das Speichern der korrigierten Daten verhindern könnte. Dies könnte etwa der Fall sein, wenn ein inkonsistentes Portfolio P1 existiert, das einem Inhaber zugeordnet ist, welcher wiederum mit Konten aus anderen Portfolios, wie Portfolio P2, verknüpft ist. Da Hibernate beim Speichern zunächst versuchen würde die Inhaberdaten aus der Datenbank in den Persistence Context zu laden (einschließlich der Beziehungen, die eager gefetcht werden), um die zu speichernden Daten mit gegebenfalls bestehenden Daten „zu mergen“, kann es zu Problemen kommen. Falls beispielsweise das inkonsistente Portfolio P1

gespeichert werden soll, lädt Hibernate auch den zugehörigen Inhaber und damit das verbundene Portfolio P2. Ist jedoch auch Portfolio P2 inkonsistent, kann das Laden und damit das Persistieren gegebenfalls fehlschlagen.

Konkret für Portfolios

Die Erkennung von Inkonsistenzen erfolgt im Wesentlichen analog zu Inhabern.

Zur Rekonstruktion eines Portfolios werden sogenannte Fake-Inhaber erzeugt. Dabei handelt es sich um unvollständige Inhaber-Instanzen, die nur die ID, den Vor- und Nachname sowie die Steuernummer enthalten. Diese minimalen Informationen genügen, um alle registrierten Inhaber darzustellen und den dem Portfolio zugeordneten Inhaber im Dialog (*FixPortfolioInconsistencies-Dialog*) anzuzeigen. Der Einsatz dieser Fake-Inhaber ist erforderlich, da das Laden über Hibernate andernfalls fehlschlagen könnte. Beim Laden eines Inhabers über Hibernate werden beispielsweise auch dessen Portfolios geladen. Falls eines dieser Portfolios ebenfalls inkonsistent ist, könnte das Laden der Inhaber unter Umständen zu einem Fehler führen. Sollten die für den Fake-Inhaber benötigten Daten fehlen – beispielsweise, wenn der zugehörige Inhaber nicht mehr existiert – wird dem Portfolio kein Inhaber zugewiesen. In diesem Fall muss ein neuer Inhaber manuell ausgewählt werden.

Die Umsetzung des Dialogs (*FixPortfolioInconsistenciesDialog*) ähnelt stark der des vorherigen Abschnitts, weshalb hier nicht weiter darauf eingegangen wird. Für die Auswahl eines Inhabers wird die Methode *OwnerRepository.findAllAsFake* aufgerufen, die analog zum vorherigen Abschnitt alle in SmartWM registrierten Inhaber als Fake-Inhaber zurückgibt.

Ein besonderer Aspekt ist die Persistierung der Anlagerichtlinien-Daten für die verschiedenen Anlagetypen. Hier wurde entschieden, die bestehenden Daten zunächst vollständig zu löschen und anschließend mithilfe nativer SQL-Befehle neu anzulegen. Der Vorteil dieser Vorgehensweise liegt darin, dass fehlerhafte Einträge nicht mühsam identifiziert, korrigiert oder manuell neu erstellt werden müssen.

Konkret für Konten

Auch die Erkennung und Behandlung von Inkonsistenzen in Konten folgt grundsätzlich dem gleichen Prinzip wie bei Inhabern und Portfolios.

Zusätzlich zum bereits erwähnten Fake-Inhaber werden hier jedoch auch Fake-Portfolios benötigt. Dieses besteht lediglich aus der Id und dem Namen des Portfolios und folgt dem gleichen Prinzip wie der Fake-Inhaber.

Auch der hierfür verwendete Dialog (*FixAccountInconcistenciesDialog*) arbeitet im Wesentlichen ähnlich zu denen der vergangenen Abschnitte.

6.6 Breadcrumb-Funktion

Das zentrale Element für die Breadcrumb-Navigation ist das *breadCrumbBar*-Attribut (*BreadCrumbBar*) in der *PortfolioManagementTabController*-Klasse. Sie stellt das zentrale Objekt dar, in der die Breadcrumbs verwaltet werden. Ursprünglich war gedacht, die bereits vorhandene Breadcrumb-Funktionalität aus der vorherigen Arbeit (vgl. [Gör23]) zu übernehmen. Allerdings erwies sich diese als unzureichend, da die Darstellung zu klein und unübersichtlich war, was die Navigation erschwerte.

Es bestand außerdem die Idee, direkt die *BreadCrumbBar*-Klasse aus dem *org.controlsfx*-Paket zu nutzen, um den Implementierungsaufwand zu reduzieren. Allerdings stellte sich heraus, dass diese Lösung ungeeignet war, da Breadcrumbs, die nicht in den verfügbaren Platz passten, nicht korrekt angezeigt wurden.

Das Hinzufügen eines neuen Breadcrumbs erfolgt über die Methode *PortfolioManagementTabController.createBreadcrumbInstance*. Falls noch kein Breadcrumb vorhanden ist, wird automatisch ein initialer Eintrag (z. B. „Portfolio-Verwaltung“, vgl. Abbildung 5.33) durch die Verwendung der *BreadCrumbBar.addRootCrumb*-Methode erstellt. Diese Methode nimmt ein Label (*String*), das den anzuzeigenden Text des initialen Breadcrumbs definiert, sowie eine Aktion (*Runnable*), die beim Klick auf den Breadcrumb ausgeführt wird. Standardmäßig sollte diese Aktion die Weiterleitung zum Hauptmenü beinhalten.

Der initiale Breadcrumb unterscheidet sich in der Form von den nachfolgenden Breadcrumbs, um den Startpunkt – also die Verwaltungskomponente – klar zu kennzeichnen. Dies wird durch die CSS-Klasse „first“ aus *org.controlsfx* realisiert. Um herauszufinden, welche CSS-Klasse in der *BreadcrumbBar*-Klasse des Pakets verwendet wird, war eine Analyse des Quellcodes erforderlich.

Alle weiteren Breadcrumbs (z. B. „Portfolio: Test“) werden über die Methode *addCrumb* hinzugefügt. Diese Methode erhält ein *BreadcrumbElement*-Objekt, das sowohl den Typ (*BreadcrumbElementType*) als auch das zugehörige Element (z. B. ein Inhaber) enthält. Der Text des Breadcrumbs wird durch die Implementierung der *toString*-Methode des übergebenen Elements bestimmt. Abhängig vom Typ wird dem Text ein entsprechendes Präfix vorangestellt, sodass beispielsweise bei einem Portfolio-Breadcrumb der Text „Portfolio:“ vorangestellt wird. Um zu verhindern, dass zu lange Namen die Leiste überladen, wurde die maximale Zeichenlänge eines Breadcrumbs auf 50 Zeichen beschränkt. Überschreitet der Text diese Grenze, wird er nach 47 Zeichen mit „…“ gekürzt. Zusätzlich wird der Methode eine Aktion übergeben, die beim Anklicken des Breadcrumbs ausgeführt wird. Bei der Implementierung dieser Aktion ist zu beachten, dass die *BreadCrumbBar*-Klasse standardmäßig alle nachfolgenden Breadcrumbs entfernt, sobald ein bestimmter Breadcrumb ausgewählt wird. Dies muss daher in den definierten Aktionen nicht gesondert berücksichtigt werden.

Ein wichtiger Aspekt ist auch das Aktualisieren der Breadcrumbs. Wurde beispielsweise ein Inhaber geöffnet, darüber zu eines seiner Konten navigiert und von da aus weiter zum Inhaber des Kontos und wird dieser dann bearbeitet, dann muss sich auch der Text im Breadcrumb entsprechend anpassen. Dazu wird aus jeder Übersicht heraus die *PortfolioManagementTabController.refreshCrumbs*-Methode beim Speichervorgang aufgerufen. Dadurch wird über alle Breadcrumbs iteriert und deren Text aktualisiert.

6.7 Abschluss

Ein weiterer inhaltlicher Aspekt dieser Arbeit war die Anwendung ausgewählter KI-gestützter Tools, basierend auf der Arbeit von Herrn Sömisch.

Dabei erwies sich Embold als wenig hilfreich. Es ist äußerst rechenintensiv und überprüft kontinuierlich eine große Anzahl an Dateien. Insbesondere nach einem Upgrade auf die IntelliJ-Version 2024.3 führte dies zu erheblichen Leistungseinbußen, bis hin zum vollständigen Einfrieren der IDE. Daher wurde Embold nach dem Upgrade nur noch am Ende eines Implementierungszyklus verwendet und während der restlichen Entwicklungszeit deaktiviert.

Im Gegensatz dazu stellte sich Github-Copilot als nützliches Werkzeug heraus, insbesondere für die schnelle Implementierung wiederkehrender oder mühsam per Hand zu schreibender Code-Strukturen, etwa die Literale des Enums InvestmentType . Allerdings zeigte auch dieses Tool Schwächen, insbesondere durch falsche Vorschläge wie die Verwendung nicht-existierender Methoden.

Besonders hervorzuheben ist die vielseitige Unterstützung durch ChatGPT, das sich als wertvolle Hilfe für verschiedene Entwicklungsaufgaben erwies. Neben der Identifikation von Fehlerursachen lieferte es alternative Lösungsansätze und unterstützte so maßgeblich den Entwicklungsprozess.

7 Zusammenfassung

In dieser Arbeit wurden die Konzepte und wesentlichen Funktionen für die Verwaltung von Inhabern, Portfolios sowie Konten entwickelt und implementiert. Ein besonderer Fokus lag auf der Erkennung und Behebung von Inkonsistenzen. Zudem wurde die Breadcrumb-Funktionalität grundlegend überarbeitet, da sie in der Bachelorarbeit von Herrn Görg nicht vollständig umgesetzt wurde. Durch diese Optimierungen können Finanzanlagen, wie Portfolios und Konten, nun effizienter verwaltet werden. Die bestehenden Beziehungen – etwa die Zuordnung von Konten zu einem Portfolio oder Depot – lassen sich zudem nun intuitiv überblicken.

Aufgrund des unerwartet hohen Aufwands für die Implementierung der Datenbank-Inkonsistenz-Erkennung und -Behandlung konnte jedoch weder das Hauptmenü der Depot-Verwaltung noch der Dialog zur Depoterstellung umgesetzt werden. Dies lag unter anderem daran, dass die erforderlichen Abfragen zur Inkonsistenz-Erkennung vertiefte SQL-Kenntnisse voraussetzten. Zudem musste für jeden Entitätstyp (Inhaber, Portfolio, Konto), zur Darstellung der Inkonsistenzen, ein zusätzlicher Dialog implementiert werden, um die Inkonsistenzen beheben oder die betroffene Entität löschen zu können. Darüber hinaus erwies sich die Nutzung von Hibernate als komplexer als erwartet. Ein weiterer erheblicher Aufwand entstand durch die Überarbeitung der Breadcrumb-Funktion und die Anpassung der Tab-Steuerung auf Basis der Arbeit von Herrn Görg.

Es ist jedoch zu beachten, dass das Aktualisieren von Breadcrumbs für Depots bislang nur rudimentär implementiert wurde und in zukünftigen Arbeiten ausgearbeitet werden muss. Falls bestehende Funktionen angepasst oder erweitert werden, die mit Listeners in Verbindung stehen, sollten die Änderungen sorgfältig überprüft werden, um unerwartete Fehler zu vermeiden. Darüber hinaus ist das Übergeben und Laden der Inhaberdaten aus der Tab-Property retrospektiv betrachtet keine elegante Entscheidung gewesen. Zunächst war gedacht die Inhaberdaten den Breadcrumbs zu übergeben und aus den Breadcrumbs dann die Daten beispielsweise in der Übersicht wieder zu entnehmen und darzustellen, jedoch sah die alte Breadcrumb-Funktion dazu keine Möglichkeit vor. Erst zu einem späteren Zeitpunkt wurde die Entscheidung getroffen die Breadcrumb-Funktion neu zu implementieren, wodurch dies erst möglich wurde.

Damit bilden die Entwicklungen eine solide Grundlage für zukünftige Erweiterungen. Zusätzlich gibt es weitere Optimierungspotenziale. Einerseits könnte die Breadcrumb-Funktion verbessert werden, indem die Breadcrumbs auch Informationen darüber enthalten, in welchen Menüpunkt man sich genau befindet. Befindet sich der Benutzer beispielsweise in der Konten-Ansicht eines Inhabers aus Abbildung 5.12 und navigiert von da aus zu eines seiner Konten und möchte anschließend über den Breadcrumb wieder zurück navigieren, wäre es intuitiver, dass der Benutzer sich dann wieder in der Konten-Ansicht des Inhabers anstelle der Inhaber-Übersicht aus Abbildung 5.8 befindet. Andererseits könnte die Darstellung der Datenbank-Inkonsistenzen optimiert werden:

Statt für jede Inkonsistenz einen separaten Dialog anzuzeigen, könnte eine Oberfläche entwickelt werden, die alle inkonsistenten Entitäten übersichtlich auflistet und bei Auswahl detailliert darstellt. So hätte der Benutzer eine bessere Gesamtübersicht über die Inkonsistenzen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AWS] AWS. *Was ist ein Ereignis-Listener?* Stand 02.03.2025. URL: https://aws.amazon.com/de/what-is/event-listener/ (siehe Seite 3).

[bae] baeldung. *Java equals() and hashCode(). Contracts.* Stand 28.01.2025. URL: https://www.baeldung.com/java-equals-hashcode-contracts (siehe Seite 63).

[Dal23] Emre Dalci. *Hibernate Persistence Context. First-Level Cache.* Juli 2023. URL: https://medium.com/emlakjet/hibernate-persistence-context-c4d3bc1e84d3#:~:text=unit%20of%20work.-,First%2DLevel%20Cache,-The%20persistence%20context (siehe Seite 44).

[Gör23] Niklas Görg. *Konzeption einer Benutzeroberfläche zur Vermögensverwaltung.* September 2023 (siehe Seiten 6, 71).

[Lin] Margit Link-Rodrigue. *BSD 3-Clause.* Stand 08.01.2025. URL: https://de.wiki.bluespice.com/wiki/BSD_3-Clause (siehe Seite 48).

[Mei24] Prof. Dr. Klaus Meißner. *Konzeption und Implementierung eines Portfolio-Managements. Anforderungen und Hinweise.* September 2024 (siehe Seiten 2, 4, 6).

[Söm24] Leander Sömisch. *Anwendung von generativen KI-Technologien auf die Java-Programmierung.* September 2024 (siehe Seite 1).

[Wik24a] Wikipedia. *Entität (Informatik) — Wikipedia, die freie Enzyklopädie.* [Online; Stand 28. Februar 2025]. 2024. URL: https://de.wikipedia.org/w/index.php?title=Entit%C3%A4t_(Informatik)&oldid=251523352 (siehe Seite 2).

[Wik24b] Wikipedia. *Konsistenz (Datenspeicherung) — Wikipedia, die freie Enzyklopädie.* [Online; Stand 31. Januar 2025]. 2024. URL: https://de.wikipedia.org/w/index.php?title=Konsistenz_(Datenspeicherung)&oldid=241742149 (siehe Seite 41).