Technische Dokumentation zum

ASE - Finanzplaner

Student: Jan Braunsodrff
Kurs: TINF18B2
Matrikelnummer: 5012578

Inhaltsverzeichnis

Benutzung

Modus: Experte

-a → Acroym

Bank

bank add -a acronym -n name -t type	Legt eine neue Bank an
bank all	zeigt alle Bank an
bank delete -a acronym	Löscht eine Bank + Account + Transactions
bank	HIlfe

Account

account add -na name -nr number -ac acronym -a bank	Legt einen Account an
account all -a bankAcronym	Zeigt alle Accounts der Bank an
account delete -a account	Löscht Account + Transaction
account	Hilfe

Transaction

transaction add -a acronym -val 123,56 -dat 01.01.2020 -thp third party -cat category -con	legt eine Transaction an
transaction all -a acronym	Zeigt letzte 20 Transaktionen einer Bank an
transaction all -a acronyn -n 30	Zeigt letzte 30 Transaktionen einer Bank an
transaction all -a acronym -f	Zeigt letzte 20 Transaktionen einer Bank mit Id an
transaction delete -id 123 345	Löscht Transaction mit Id 123 und 345
transaction group -a test	Fasst alle monate Zusammen
transaction export -a account -s 01012020 -e 01022020	Erstellt einen export für den Account von 1.1.2020 bis zum 1.2.2020

Modus: Directory

Bank

State	Command	Beschreibung	New State
Bank	cat acronym	Zeit alle Accounts von der Bank id an	Bank
Bank	cd acronym	Zeit alle Accounts von der Bank id an	Account
Bank	ls	Zeigt alle Banken an	Bank
Bank	rm acronym	Löscht Bank	
Bank	touch -n name -a acronym -t type	Legt eine Bank an	Bank

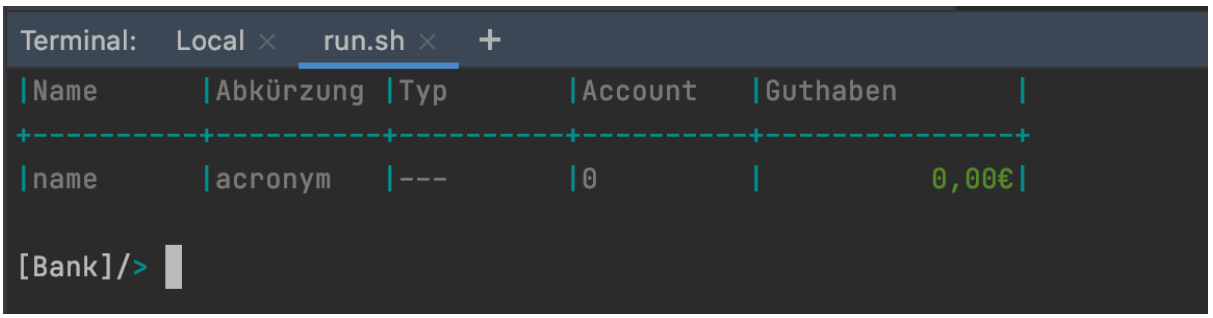
Account

State	Command	Beschreibung	New State
Account	cat acronym	Zeigt die letzten 20 Transaktionen an	Account
Account	cd ..	Zeigt alle Banken an	Bank
Account	cd acronym	Zeigt die letzten 20 Transaktionen an	Transaction
Account	ls	Zeit alle Accounts an	Account
Account	rm acronym	löscht ein Account	Account
Account	touch -na name -nr number -ac accountAcronym	legt ein Account an	Account

Transaction

State	Command	Beschreibung	New State
Transaction	rm id	löscht Transaction	Transaction
Transaction	cd ..	Zeigt alle Accounts an	Account
Transaction	group	Gruppiert alle Monate	Transaction
Transaction	ls	Zeigt die letzten 20 Transaktionen an	Transaction
Transaction	ls -n 240	Zeit die letzten 240 Transaktionen an	Transaction
Transaction	ls -f	Zeigt Transaktionen mit Id an	Transaction

Transaction	print 022020	Exportiert den Monat Februar	Transaction
Transaction	print 012020 042020	Exportiert den Monate Januar bis April	Transaction
Transaction	touch -val 20,70 -thp thp -dat 20.12.2021 -cat cat -con	Legt eine Transaktion an	Transaction



Begründung von Bibliotheken

Spring:
Spring bietet mir mehrere Vorteile:

1. Ein Teil der Interaktion mit der Anwendung ist über eine REST-Schnittstelle. Spring bietet mir die Möglichkeit ohne JEE einfach und schnell eine API mit Routes bereitzustellen. Die Syntax ist einfacher zu lesen und der Code lässt sich leichter warten.
2. Spring JPA bietet mir eine Einheitlich Abstraktion über verschiedene Datenbanken. Ich brauch nur das Interface zu implementieren, Spring baut für mich den ORM und die SQL Statements und managt die Verbindung. Zusätzlich kümmert sie sich ums Cachen und die Kommunikation.
3. Spring bietet über Beans, Profile und Configdatein die Möglichkeit über das Dependency Injection Principal die Software zu steuern bzw. gibt die Möglichkeit zum Starten der Software zu sagen, welche Module wie zusammengebaut werden sollen.

H2:
Ist eine In-Memory Datenbank, die auch in eine Datei schreiben kann. Ist die schnellste und einfachste Lösung, um die Anwendung Portabel zu machen, aber gleichzeitig auch schnell zu machen. Eine JSON Datei ist zwar Menschenlesbar, aber nicht performant auf Filtern und Suchen.

GSON:
GSON kann JSON Strings in Objekte und Objekte in ein JSON Strings umwandeln. Das erleichtert das Speichern von Objekten in eine Datei. Als ich noch alles in ein File-Repository geschrieben habe, brauchte ich es.

JUNIT:
JUINT benutz ich zum Ausführen von den Unit-Tests. Die Jupyter Plattform ist die aktuellste Version von dem Framework und spielt gut im Intellij zusammen.

Hamcrest:
Hamcrest ist ein schöners Assert. Es bietet mir die Möglichkeit abfragen, wie und ob ein Liste folgende Elemente enthält, Prüfung auf null und Vergleiche leserlicher zu schreiben. Notfalls kann auch eine eigener Matcher geschrieben werden.

Programmierprinzipien

SOLID

Single Responsibility Principle:
Das extremste Beispiel ist sind die Action im Pakage **de.janbraunsdorff.ase.layer.presentation.console.expert.action**. Hier hat jede Funktionalität eigenen Klasse bekommen. Jedes Kommando, welches anhand der ersten zwei Wörter bestimmt werden kann, eine eigene Klasse mit eigenen Abhängigkeiten. Dadurch sind alle Klassen Vergleichsweise sehr klein, haben maximal zwei Abhängigkeiten, die über ein Interface abgebildet werden und sind einfach zu Testen und zu warten. Dasselbe Prinzip ist auch ist auch in den Klassen aus den Packages **de.janbraunsdorff.ase.layer.presentation.console.directory.*** anzuwenden. Wobei in diesen Klassen keine Abhängigkeiten haben.

Die Builder **DistributorUsecaseFactory** und ActorFactory haben nur die Aufgabe Syntaktischen Zucker zu geben. Mehr dazu in Design Patterns.

Schaut man sich die Klasse **BarChart** an, so ist eine die Klasse **Grid** fest an diese Klasse gekoppelt. Das ist eine bewusste Entscheidung. Die Klasse Grid lagert nur Funktionalität aus, und dient somit zur besseren Lesbarkeit des Codes.

Bewusst gegen das Prinzip habe ich mich bei den Servicen (**TransactionService, BankService, AccountAnalytics, AccountIO**) entschieden. Diese haben viele Funktionalitäten, welche aber in der Domänenlogik zusammen gehören. Deswegen sind diese Klassen größer. Die Spaltung in habe ich aus lesbarkeit gemacht. Die Logik, um den Accountbereich, ist über Entwicklungszeit größer geworden. Außerdem wenden von den Konsumenten der Interfaces von den Services nicht jede Funktion verwendet.
Selbes gilt für die Repositories.

Open Closed Principle:
Die besten Beispiele sind die Klassen **TransactionActor, BankActor, AccountActor, DistributorAction, DistributorUseCase**. Diese Klassen haben eine Map, die durch die Methode addAcction, addUseCase oder addBuilder erweitert werden kann. Alles sind nur Klassen, die dazu da sind aufgrund eines Strings den Input an die richtige Stelle weiterzuleiten. Zusätzlich ist alles Interface-Basiert. Das bedeutet der Klasse ist es egal, wie sie benutzt wird, bietet aber die Funktionalität der Verteilung an weiter Interfaces an. Die Klassen können nach der Instanziierung einfach um Funktionen erweitert werden.

Liskov Substitution Principle:
Gute Beispiele sind die Factories zum ausgeben auf der Konsole. Die Basisklasse ist **PrinterInputFactory**. Hier stehen die Grundfunktionalitäten zur Verfügung. Das sind das Anzeigen von Info und Error Texten, neue Zeile Einfügen und alle Bestandteile der Ausgabe als String zusammenfassen. Davon werden dann die Klassen **InformationPrinterInputFactory**, **HelpPrinterInputFactory**, **ErrorPrinterInputFactory**,

und **TablePrinterInputFactory** abgeleitet. Hierbei werden keine Methoden überschrieben oder geändert. Dadurch ist die korrektheit der Basisklasse gewährleistet. Alle Abgeleitet Typen verhalten sich wie die Basisklasse mit zusätzlicher Funktionalität.

Result und **TypedResult** stehen ebenfalls in einer Beziehung. Hier wird ein Interface um eine weitere Funktion erweitert. Die Typen werden impliziert durch Java weitergegeben. Ein **TypedResult** ist auch ein **Result.**

Im Bereich Reporting ist gibt es verschiedene Abschnitte von der PDF. Alle Implemtieren das Interface **PdfPart**. Das sorgt bei allen Ableitungen dafür das es gerendert werden kann. Darauf stützen sich die explizite Ableitung **PdfChapter** und die implizierten Ableitungen **MonthSummary** und **MonthlySummary**. Die Ableitung von **PdfChapter** nach **MonthSummary** / **MonthlySummary** ist lediglich eine Erweiterung der Funktionen. Dasselbe Muster ist auch im Pakage **de.janbraunsdorff.ase.layer.domain.reporting.pdf.part.***. Hier liegen alle Teile der PDF und Implementieren das Interface **PdfPart.**

Interface Segregation Principle:
Alle Schichten sind durch Interfaces getrennt. Dadurch hat die Domäne keine Abhängigkeit auf die anderen Schichten. Die Möglichkeit, die ich sehe es auf Krampf einzubauen ist **AccountAnalytics, AccountIO** in eine Klasse zu mergen und diese Implementiert die jeweiligen Interfaces. Das habe ich aus gründen der Lesbarkeit nicht vor.

Dependency Inversion Principle:
Gut zusehen ist es an der Klasse **ApplicationWeb**. Hier werden durch “Spring Magic” die die richtigen Implementierung von den Interfaces instanziiert und in den CommandLineRunner eingefügt. Das ermöglicht das Austauschen von Schichten durch anpassen der Klasse **BeanApplication**. Hier werden alle Teiel aus einer Schichten definiert wie sie gebaut werden sollen. Das ganze lässt sich in der Theorie soweit treiben, dass Spring Profile für Beans anlegen kann. Dadurch ist es möglich zum Starten der Anwendung ein Profil auszuwählen. In meiner Anwendung ist es bewusst über Beans geregelt und dass alle Interfaces, die Irgendwo als Dependencies erstellt werden von mir erstellt werden. Damit habe ich Spring an den Rand meiner Anwendung gedrückt und es einfach austauschen ohne jede Klasse anzupassen (Never use @Autwired!!).

GRASP

Low Coupling
Zwischen Schichten findet jeglicher Austausch über Interfaces statt. Keine Schicht ist von der konkreten Implementierung der anderen Schicht abhängig. Innerhalb der Schichten wird auf geringe Abhängigkeiten geachtet. Um Objekte zu erzeugen wird in der Regel kein bis maximal ein weiteres Objekt benötigt. Ausnahmen sind die Services in der Domäne. Hier werden mehrer Repositories pro Services benötigt. Diese werden aber nur durch ein Interface definiert. Dadurch binden sich ein Service an viele andere Klassen aber kann auch als Zentralen Punkt in der Domäne Angesprochen werden.

Das Value Object **Value** ist in vielen Klassen und Entitäten enthalten. Hier besteht ein Starke kopplung.

Der CommandBuilder und die DistributorUsecaseFactory so wie die DistributorActionFactory arbeiten nur auf Interfaces. Dadurch ist hier ein lose Kopplung möglich. Fest verdrahtet sind Objekte im Package **de.janbraunsdorff.ase.layer.presentation.console.expert.printing**. Da diese Objekt nur in einer Einheit Sinn ergeben.

Die Klasse PrinterInputFactory ist die Stammklasse zum erstellen eines Outputtextes und biete die

High Cohesion
Ein gutes Bespiel ist, dass **BarChart** und **ChartData** zwei unterschiedliche Klassen sind. So kann die Daten getrennt von der Logik zum bauen und die Klasse **ChartData** kann in anderen möglichen Charts verwendet werden,

DRY

Bewusste Duplication von Code:
Die Klassen **AccountActor** und **TransactionActor** haben denselben Aufbau und auch eine fast identische Logik. Die einzige Unterscheidung ist das Default Behavior, falls ein Kommando nicht gefunden wurde. Fachlich gesehen ist es keine Doppelung. Deswegen halte ich hier eine zusammenlegung nicht für notwendig.

Ein Grund das Value Objekt **Value**, welches den Wert einer Transaction abbildet, einzuführen ist die Methode **getDecimalString** und **getFormatted**. Die Ausgabe in verschiedenen bereichen des Codes soll einheitlich sein. Ändert sich das Format, so darf es nur eine Stelle zum ändern geben.

Die Parselogik, um den Input von User in die Form zu bringen, die von der Domäne verstanden werden kann ist aus einer Methode im Interface in das Value Objekt **ExpertCommand** gewandert.

Entwurfsmuster

Builder Pattern

Das Builder Pattern hilft dabei komplexe Objekte leserlich zu erstellen.

ApplicationConsoleBuilder:
Der Application Builder baut alle Bestandteile zusammen, wie man es will. Er baut die Repositories, Verknüpft diese mit der Domäne, die wiederum mit der Präsentationsschicht verknüpft wird. Das Zusammenstellen der Komponenten ist dadurch leichter und man weiß was man zusammen bauen will.

PrinterInputFactory:
Diese Klasse ist die Basisklasse für alle Klassen, die den Output bauen. Der Anwender dieser Klasse braucht nur zu sagen er will einen Info Text und dieser wird richtig formatiert angezeigt. Von dieser Klasse erben alle Weiteren Factories, die zum Generieren von Konsolen Output gedacht sind. Abgelegt in dem Package **de.janbraunsdorff.ase.layer.presentation.console.expert.printing.factory.** Am Beispiel TablePrinterInputFactory ist das Komplexe zusammenbauen der Tabellenausgabe zu sehen. Hierbei wird Logik ausgeführt, um die Tabelle ordentlich anzeigen zu können. Das erspart dem Programmiere viel Zeit im Code schreiben und es ist deutlich leselicher. Die Methoden sagen aus, was gemacht wird. Um Details muss sich nicht mehr gekümmert oder immer wieder neu geschrieben werden.

DistributorUsecaseFactory und **DistributorActionFactory**:
Das sind zwei Klassen die auf dem Open-Close Principle Aufsetzen. Diese Builder instanziieren eine Klasse um dann im Build verfahren diese Klasse von außen zu erweitern. Im Grunde ist es nur syntaktischer Suggar. Aber es macht die Aufrufe deutlich besser zum lesen in der Klasse **CommandBaseCli**.

ActorFactory:
Selbes Spiel in der Klasse ActorFactory. Nur mit dem Unterschied, dass sie Generisch ist. Hier kann jeder beliebiger Actor, in diesem Falle für den CommandOverlay, gebaut werden. Das liegt daran, dass der eigentliche Actor, der das Übersetzen vom Overlay in den Expertenmodus übersetzt durch ein Interface definiert wird. Auch hier ist es nur Syntaktischer Sugger. Aber es macht den Code leserlicher und spart Schreibaufwand.

Factory Pattern:

Das Factory Pattern hilft dem Programmiere nicht immer die selben Objekte mit denselben Attributen zu erstellen. Das spart Tipparbeit und man muss nur ein Stelle ändern.

Part:
Part das Elementarste Teil der Konsolenausgabe. Die Parameter ist der Text und die Farbe, in der der Text gedruckt werden soll. Zusammengehörige Teile sollen immer identisch aussehen. Damit nicht einer auf die Idee kommt die Farbe der Tabelle mal in Rosa oder

Gelb zu färben, gibt es die Objekte in dem Package **de.janbraunsdorff.ase.layer.presentation.console.expert.printing.part**. Hie sind die Verschieden Standardtypen aufgezählt. Soll die Farbe geändert werden muss es nur in einer Klasse gemacht werden.

Domain Driven Design

Analyse der Ubiquitous Language

Bank: In der echten Welt wird auch das Geld auf eine Bank gebracht.
Account: Eine Bank legt Konten (engl. Account) an. Dadurch wird das geld strukturiert verwaltet.
AccountNumber: Ist die Kontonummer. Ebenfalls verständlich
Transaction: Hier kann man diskutieren. Redet man nur von Umsätzen auf dem Konto könnte man es auch Posting nennen. Da in der Domäne auch Transaktionen von Fonds und ETFs abgebildet werden können, ist der allgemeinere Begriff Transaktion besser zu verstehen.
Value: Beschreibt den Wert einer Transaction. Alternative könne man auch Amount nehmen. Allerdings ist dabei die Gefahr, dass diese mit Anzahl übersetzt werden kann. Value beschreibt den Wert der Transaktion

Repositories

Alle Repositories werden in der Domäne über ein Interface beschrieben. Da die Domäne nicht weiß, wie Daten geschpeichert werden, werden in das Repository die Domänentitäten nur gegeben. Die Repositories haben die Aufgabe diese Entitäten abzuspeichern, und nach Bedarf wieder zuladen und in eine Domänenentität zu transformieren. Aufgrund der Verwendung von Spring Data sind die Repositories, die auf eine Datenbank zugreifen, sind die Repositories zweigeteilt. Zum einen weil das Repository für Spring nur ein Interface ist, welches durch Spring instanziiert wird. Zum anderen um nicht der Domäne von den Entitäten in der Datenbank abhängig zu machen. Jede Entität, die in die Datenbank gespeichert wird, verfügt über eine Methode **toDomain**, um die Entität in das Domänemodell zu überführen.

Repositories Definition: **AccountRepository, BankRepository, TransactionRepository**
Repositories Implementierung: **AccountDatabaseRepository, BankDatabaseRepository, TransactionDatabaseRepository**
Persistance Entitäten: **AccountDatabaseEntity, BankDatabaseEntity, TransactionDatabaseEntity**

Aggregates

Auf Aggregate konnte ich verzichten in der Domäne. In der Domäne sind die Banken, Accounts und Transaktionen so zu kapseln, dass sie nur in der Datenbank über ein Akronym verbunden sind. Abhängigkeiten sind im Code selbst nicht zu finden. Deshalb macht es keinen Sinn künstlich etwas einzuführen, was man nicht braucht.

Entities

Als Entitäten in der Domäne werden **Account**, **Transaction** und **Bank** angesehen. Jede Entität besitzt eine Id. Darüber kann technisch unterschieden werden, welche Entität gerade vorliegt. Fachlich gesehen ist das Attribut **Acronym** auch ein eindeutige Identifikator, da eine **Bank** und ein **Account** auch über das **Acronym** eindeutig erkennbar sein soll (Domänen

regel). Diese Objekte sind reine Datenhaltung. Keine Funktionalität wird in der Klasse abgebildet. Die Attribute sind private und nur über getter erreichbar.

Weitere Entitäten werden in der Persistence definiert. Diese Entitäten haben auch eine Id, um in der Datenbank eindeutig identifizierbar zu sein.

Value Objects

Value:
Es gibt nur ein mal 1,50€. Wenn sich der Wert ändert ist es ein neuer Wert. Es ist einfacher der Wert als Integer abzubilden und zu rechnen. Jedoch ist der Integer die Technische Sicht auf eine Zahl. Value wird in der Domäne eingesetzt und ist dadurch komplexer zu betrachten. Formatierungen nach verschiedenen Standards und Währungen können in diesem Objekt abgebildet werden. Jede Veränderung resultiert in ein neues Objekt.

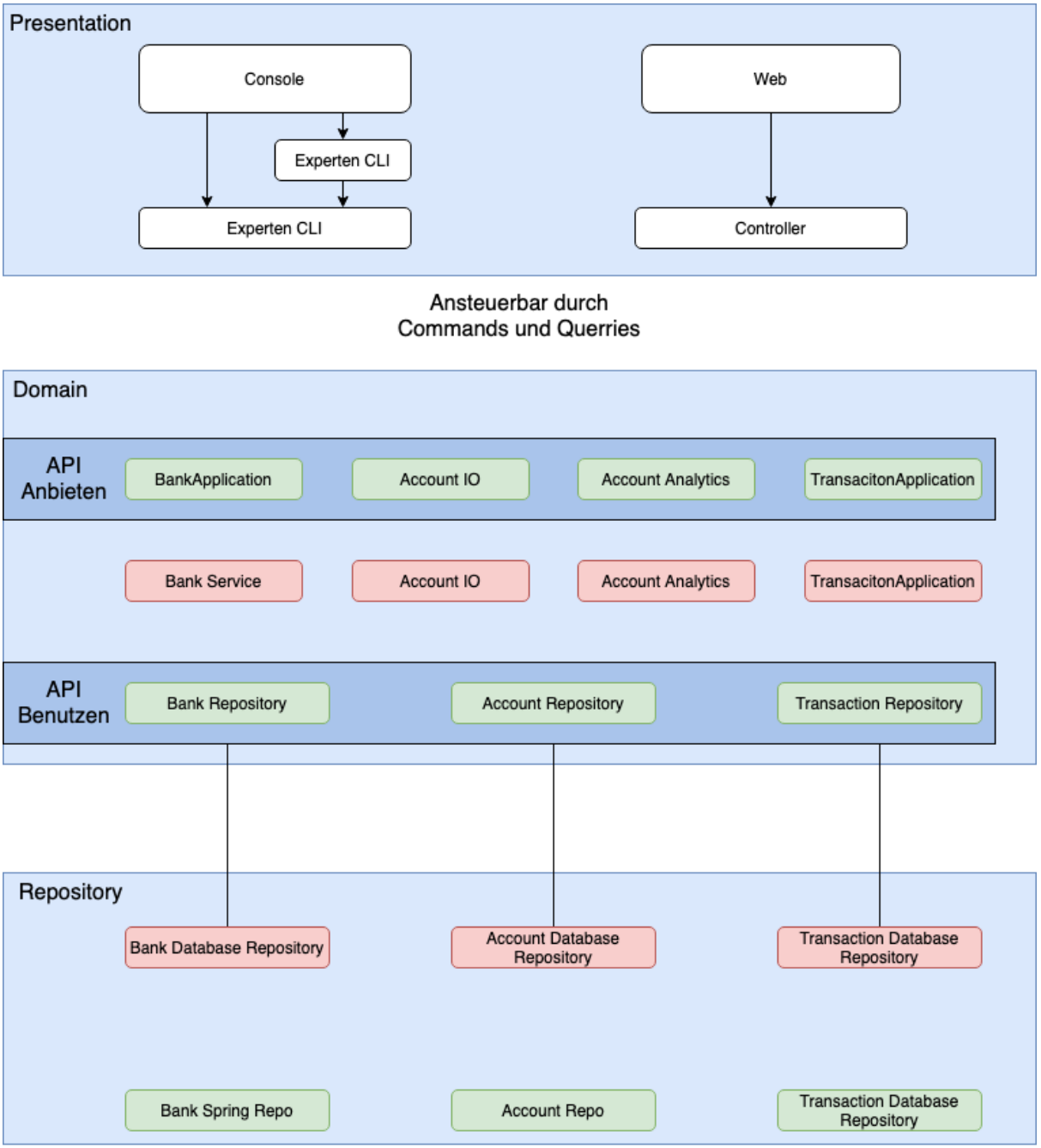
DataPoint:
Hier kann man sich die Frage stellen Value Object oder Entity. Aber ein Datenpunkt hat für sich keine Identität. Aus diesem Grund kein Entity. Der Datenpunkt darf nicht verändert werden, ohne dass sich die Wertigkeit verändert. Deswegen ist das ein Hinweis auf ein Value Objekt. Ändert sich der Wert in x oder y Richtung ist es ein neuer Datenpunkt. Referenzen auf den alten Punkt führen zu fehlern.

ChartData:
Analog zu Datapoint. Ändern sich die Daten für ein Diagramm, ist es ein andere Datensatz.

State:
Ein State beschreibt im ordnerbasieren Interface in welche “Ordner” man sich befindet. Dieser Ordner wird definiert durch die Bank und den Account. Ein Ordner kann in diese Kombination nur einmal existieren. Ändert man den Aufenhaltsort vom User ist es ein neuer State. Die Übergänge von Ordner nach Ordner müssen konsistent sein. Deswegen ist die Logik der Übergänge an den State an sich gekoppelt. Zusätzlich um den aktuelle Status anzuzeigen gibt es eine einheitliche Methode.

ExpertCommand:
Die ursprüngliche Implementierung hat vorgesehen das Parsen eines Kommandos im Interface zu machen, welches von dem Actions bzw Buildern implementiert wurde, Jedoch ist das ein Code Smell. Spätestens nachdem auf den Experten Interface noch das benutzerfreundliche Interface gebaut wurde war es keine Lösung mehr das Parsen an der zu benutzenden Klasse zu hängen. Deswegen wurde es zu einem eigenen Value Object. Ein Command ist unveränderlich. Als User würde ich mir verarscht vorkommen, wenn das Programm meine Eingabe umschreiben würde. Die Logik, welche Attribute enthalten sind, welchen Usecase und welche Action bedient wird und das bekommen der Attribute im Kommando sind deswegen an den String gekoppelt.

Architektur



Die Anwendung hat drei Schichten.

1) Presentation (Technisch): In dieser Schicht kann der User in den Status der Anwendung einsehen und mit der Anwendung interagieren. Dieser Teil ist kann als eigenständiges Programm angesehen werden, welches die Anfragen für die Domain vorbereitet, Dadurch ist sichergestellt, das die Domain unabhängig vom Rest der anwendung ist. Interaktion mit der Domäne nur über Interfaces und Befehle und Abfragen, welche von der Domäne ebenfalls bereit gestellt werden.
2) Domain (Fachlich): Hier ist die Geschäftslogik. Diese Ist Technologieunabhängig. Deswegen bietet die Domain sowohl die API an, mit der Befehle und Abfragen in die Domaine kommen, als auch eine API, um Daten zu speichern. Damit ist die Domäne nicht von der art der Persistierung oder Graphischen Aufbereitung der Daten

verantwortlich und kann prinzipiell als eigenes Programm genutzt werden. Das Austauschen der Speicherung und des Frontends ist beliebig je nach anwendungsfall. Die Domain ist nicht Parallel.

3) Repository (Technisch): Hier werden die Daten gespeichert. Dies Schicht implementiert die Interfaces zur Speicherung der Daten. Im Falle von Spring ist das nur eine weitere Abstraktion von der Spring JPA.

Das Spring Framework wird nur in der 1. und 3. Schicht verwendet. Alle Instanzen werden in der Klasse **BeanApplication** definiert, wie sie zu bauen sind. Dadurch halte ich die Domäne fern von Abhängigkeiten dritter Bibliotheken.

Unit Tests

Alle Tests sind nach dem AAA Prinzip aufgebaut. Arrange → Act → Assert

ATRIP

Automatic:

Tests müssen einfach ausführbar sein → In Intellij Rechtsklick auf den Test Ordner und Run drücken. Auf der Kommandoziele: ./gradlew test

Tests müssen automatisch ablaufen → Jeder Test hat seine eigene Eingaben, die gemacht werden. Entweder stehen sie im Code oder werden im ParameterizedTest und da im CsvSource angegeben. Beispiele: ValueTest, AccountAddActionTest

Tests müssen sich selbst überprüfe → In jedem Test ist mindestens ein assertThat enthalten. Zum überprüfen wird die Bibliothek hamcrest verwendet. Diese kann auch überprüfen, ob die Richtige exception geschmissen wurde. Ausßerdem ist es dadurch einfacher Listen und Sets auf vollständigkeit zu überprüfen. Beispiele: AccountIOTest.getAccountsByAcronymMissingAccount, AccountIOTest.getAccountsByAcronym (2. assetThat)

Thorough:

Ein Test muss alles notwendige überprüfen → Jeder Test hat nur eine Funktion, die Aufgerufen wird um sie zu testen. Dabei wird Überprüft, ob der der Rückgabewert so wie alle zu Methoden von Mocks mit den entsprechenden Parametern aufgerufen werden. Damit ist sichergestellt, dass die Funktion das macht, was man will. Jedoch ist mir bewusst, dass die Funktion Seiteneffekte verursachen kann, die nicht getestet werden. Beispiel: A**ccountIOTest.getAccountsByAcronym**, da prüfe ich ob alle Objekte in der Liste zurückgegeben werden und ob alle Objekte denselben wert haben. Außerdem ob das Repository mit dem richtigen Wert aufgerufen wurde

Iteratives Vorgehen zur Erstellung der Tests → Bei der Entwicklung habe ich es per Hand getestet, was gegen “Automatic” verstößt. Erst im Nachhinein habe ich angefangen zu testen. War nicht so ganz die Beste Idee

Fehler sind nicht gleichmäßig über den Code verteilt → Jeder Usecase ist durch die parametrisierten Test mit mehreren Eingaben abgesichter. Dadurch können fehler im selben Bereich minimiert werden. Beispiele: ValueTest

Repeatable:

Test muss beliebig wiederholbar sein und immer das gleiche Ergebnis liefern → Ich kann alle Test wiederholen, wie ich will. Im Test werden keine zufälligen werten getestet. Alle Methoden sind Deterministisch. Einige erzeugen ein Objekt, die eine UUID enthält. Diese wird nicht getestet, da es auf die Anderen Attribute ankommt, die im Objekt sind.

Tests, die ohne Änderung fehlschlagen, sind selbst fehlerhaft → Bisher konnte ich alles ausführen, und es ist immer grün gewesen.

Independent:

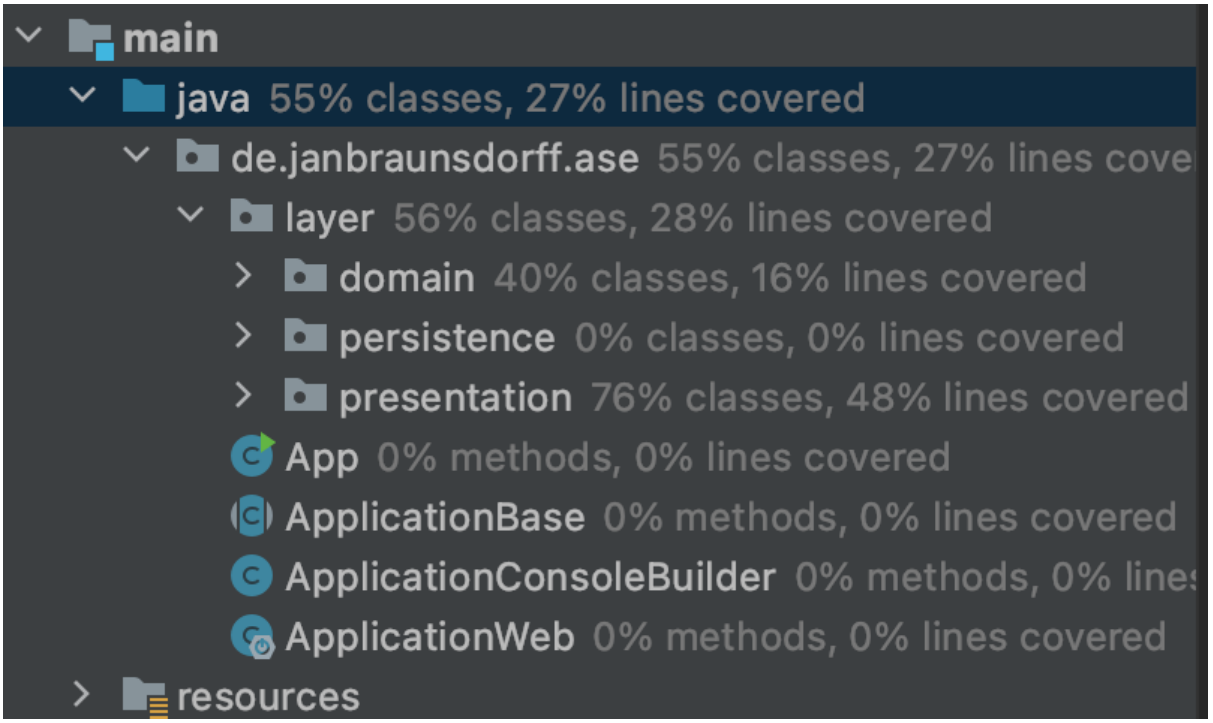
Tests dürfen keine Abhängigkeit zu anderen Tests haben → getestet wird nur gegen Daten in Memory. Jeder Test hat seine eigenen Daten, die nur im Scope vom Test verfügbar sind. Um sich auf dem Test zu konzertieren und nicht immer die selben Zeilen Code schreiben zu müssen ist in der Datei CommandBaseCliTest ein BeforeEach enthaten. Dadurch werden die privaten Felder in die Klasse geladen. Damit man nur noch testen muss, ob in der Map alles richtig erstellt wurde.

Professional:

Keine unnötigen Tests schreiben → Meine Test testen nur Funktionen, wie sie sich in künstlich erzeugten Situationen verhalten.

Tests sind Teil der Dokumentation → In Tests für die CLI kann man ablesen, welche Befehle eingegeben werden können. In den Verteilern und Aktoren Tests kann abgelesen werden, wie diese Zusammenspielen. Beispiel: **TouchTransactionTest**, hier sind alle Befehle die gehen und nicht gehen aufgeschrieben.

Code Coverage



Element	Class, %	Method, %	Line, %
domain	40% (23/57)	26% (58/222)	16% (131/799)
persistence	0% (0/19)	0% (0/153)	0% (0/502)
presentation	76% (75/98)	48% (134/276)	48% (567/1165)

Einsatz von Mocks

Ich habe kein Mocking Framework benutzt, da ich es nicht benötige aufgrund meiner Architektur. Abhängigkeiten werden über Interfaces Definiert. Dadurch ist es einfacher eine Testimplementierung zu schreiben. Ein Beispiel ist die Klasse **AccountIOApplicationTestImplementation**. Das ist eine Testimplementierung vom Interface **AccountIOApplication**. Das kann genutzt werden, um die Funktionen so zu gestalten, dass sie die im Test die benötigten Spezialfälle wiedergeben. Zusätzlich werden alle Parameter in den Aufrufen als Instanzvariable gespeichert und können somit leichter im Test überprüft werden. Weitere Beispiel sind: **BankApplicationTestImplementation, TransactionApplicationTestImplementation, TestAccountRepository, TestBankRepository** und **TestTransactionRepository.**

Eine weiter Art von Mocks ist im **ActorFactoryTest** zu sehen. Da wird der Mock nicht als Abhängigkeit benutzt, sondern als Objekt, welches übergeben wird, bzw. Instanziiert. Dadurch ist auch einfacher das Verhalten im Test zu kontrollieren und zu überprüfen ob alle Methoden mit den richtigen Parametern aufgerufen werden.

Refactoring

Auslagern von Logik in Object Values:

Das größte Refactoring in diesem Projekt ist das erstellen von Object Values. Der Grund ist die Kapselung der Logik an die Daten. Somit werden keine Hilfsmethoden oder Util Klassen benötigt.

Command:
Das Refactoring ist zusehen unter https://github.com/janbraunsdorff/ASE-Finanzplaner/commit/14b48c4c9a17e22fb2154d6e862ea3efef9fc797 Hier wird aus der Klasse **UseCase** die Parselokig entfernt und mit dem String, der den Command vom User enthält, in ein Value Object mit dem Namen **Command** (jetzt **ExpertCommand**) mit Parse Logik getan. Dadurch ist Parsen an den Command gebunden und nicht mehr an die Klasse, die diesen Command verarbeitet.
Zusätzlich wurde nach einiger Zeit der Command in ExpertCommand umbenannt. Da es zur Experten Anwendung gehört und sich dadurch von dem benutzerfreundlichen Overlay unterscheidet.

Value:
Früher war der Wert von einer Buchung, Account oder Bank nur Integer, die geparst werden konnten wie man Lustig war. Mit dem Value Object Value ist das Formatieren einheitlicher geworden.
https://github.com/janbraunsdorff/ASE-Finanzplaner/commit/9ae7884e98b1c574e1b5fefe607ffa3899d07f89

Auslagern Doppelter Logik:

Durch das Auslagern von Logik muss weniger geschrieben werden und das Anpassen bei Fehlern muss nur einmal gemacht werden. In der Klasse **AccountAnalytics** gibt es die Methode getPercent. Diese wird von zwei weiteren Methoden aufgerufen. Um diese Funktionalität nur einmal zu haben, wird dies in eine Methode ausgelagert. So kann bei bedarf sie wieder verwendet werden.
Weiter Beispiele:
AccountIO.getAccount, AccountIO.getAccount
TransactionService.checkIfAccountExists

Auslagern Langer Logik:

Lange Methoden machen es dem Entwickler schwerer zu lesen und verstehen. Deswegen ist es wichtig Logik Abschnitte in eigene Funktionen zu Methoden zu kapseln und leserlich zu benennen. Ein gutes Beispiel ist in der Klasse **BarChart**. Im Konstruktor sollen sofort die skalar und die Balken gezeichnet werden im Grid. Um das nicht alles in dem Konstruktor zu machen, sind diese Beiden sachen ausgelagert wurden,