ASE Projekt

Financial Planner - Patrick Welter - TINF19B4

Projektbeschreibung

In diesem Softwareprojekt wurde ein Finanzplaner entwickelt. Damit sollen Monatliche Ausgaben getrackt und verwaltet werden. Folgende Use Cases wurden dabei umgesetzt:

1. Ein Finanzplan kann angelegt und gelöscht werden. Er enthält einen Namen und ein verfügbares Budget.

2. Eine regelmäßige / monatliche Ausgabe kann unter dem Finanzplan registriert und gelöscht werden (z.B. Strom, Miete). Auch sie enthält einen Bezeichner und einen Geldbetrag.

3. Eine unregelmäßige Abgabekategorie kann unter dem Finanzplaner angelegt und gelöscht werden (z.B. Essen, Freizeit). Name und Budget sind auch hier Eigenschaften.

4. Unter diese Abgabekategorie können Ausgaben registriert werden. Sie sind datiert und besitzen einen eigenen Betrag.

Daneben können Reports über die einzelnen freien Budgets erstellt werden. Dazu im Kapitel DDD mehr. Aufgrund von zeitlichen Druck, wurde kein zusätzliches Frontend entwickelt. Die implementierten Funktionen können jedoch über Swagger getestet werden. Dazu muss die Anwendung gestartet werden und anschließend kann über Swagger alle REST-Endpunkte getestet werden.

Das Repository ist unter diesem Link zu finden. Im Readme ist auch die Anleitung zum Starten zu finden.

Domain Driven Design

Der erste Teil des Domain Driven Design ist die Analyse der Ubiquitous Language. Damit ist das Vokabular gemeint, welches in der Domain existiert. Es ist wichtig dies am Anfang zu definieren, damit keine Missverständnisse auftreten:

- ***Financial Plan***: Dies ist der Finanzplan. Er verwaltet alle Informationen.

- ***Category***: Damit ist die Kategorie gemeint, unter die unregelmäßige Ausgaben, wie Essen, gebucht werden können.

- ***Regular Expense***: Ein Ausgabe-Eintrag, welcher jeden Monat fällig wird und dementsprechend nicht für andere Ausgaben zur Verfügung steht (z.B. Miete).

- ***Irregular Expense***: Ein Ausgabe-Eintrag, welcher spontan fällig wird. Ein Beispiel wäre ein Restaurantbesuch.

Daraus lassen sich folgende taktische Muster implementieren:

Value Objects

Value Objects sind finale Objekte, welche ihre Identität nur aus ihrem inneren Zustand gewinnen. Das bedeutet, zwei Value Objects mit den gleichen Werten sind das gleiche Value Object. Die beiden folgenden Value Objects (repo) verwalten in diesem Fall einzelne Werte, welche durch Assertions geschützt sind. Dadurch können in der Domain geltende Regeln verfolgt werden:

- ***Name***: Der Name ist ein Bezeichner für alle möglichen Entitäten. Dabei wurde festgelegt, dass der Name nicht leer und zwischen 3 und 127 Zeichen lang sein muss.
- ***MoneyAmount***: Es wird viel mit Beträgen gearbeitet (ob nun Budget oder eine Buchung). Dabei ist eine Eigenschaft von Beträgen, dass diese immer positiv sein müssen.

Entities

Eine Entity ist innerhalb der Domäne eindeutig und wird über die ID identifiziert. Das bedeutet, dass zwei Entitäten vom gleichen Typ und mit den gleichen Werten existieren können, aber zwei verschiedene Instanzen darstellen (durch unterschiedliche ID). Die Entitäten sind hier zu finden.

- ***FinancialPlan***: Diese Entity repräsentiert den Finanzplan. Sie hat einen Name, eine Budget und zwei Listen mit weiteren Entitäten.

- ***RegularExpense:*** Diese Regelmäßige Ausgabe füllen die eine Liste des Finanzplans. Sie selbst besitzen Bezeichner und ein Wert (value).

- ***Category***: Die Category füllt die zweite List des Finanzplan. Neben Bezeichner und Budget besitzt sie eine Liste, unter die IrregularExpenses gebucht werden können.

- ***IrregularExpense:*** Diese Entity repräsentiert eine eindeutige Ausgabe. Sie existiert unter einer Kategorie, hat einen Namen, Betrag und besitzt ein Datum als Kontext.

Da beide Expense Typen sich das Budget und den Bezeichner teilen, wurde dafür ein Interface angelegt.

Aggregate

Aggregate sind logische Verbindungen zwischen Entitäten und Value Objects, welche als Objekte dargestellt werden. In diesem Projekt existieren 3 Aggregate. Sie repräsentiere alle

drei so genannte Budget Reports. Diese geben Informationen darüber, wie viel Budget in verschiedenen Kontexte bereits ausgegeben und noch zur Verfügung stehen:

- ***FinancialPlanBudgetReport***: Hier wird angegeben, wie viel Budget ein FinancialPlan zum einplanen noch hat. Das bedeutet im Klartext, das Budget welches für den Plan angegeben wurde minus die regelmäßige Ausgaben und minus die definierten Kategorien.

- ***IrregularExpenseBudgetReport***: Dieser Report gibt an, wieviel des insgesamten Budget schon ausgegeben wurde, bzw. wie viel noch ausgegeben werden kann. Also Finanzplan Budget minus regelmäßige Ausgaben und minus unregelmäßige Ausgaben.

- ***CategoricalBudgetReport***:Hier wird berechnet, wie viel Budget einer Kategorie schon von Ausgaben verbraucht wurden. Sprich Kategorie-Budget minus unregelmäßige Ausgaben der Kategorie.

Repositories

Über Repositories können Entities serialisiert und persistiert werden. Aufgrund der Hierarchie der Objekte, muss nur ein Repo angelegt werden:

- ***FinancialPlanRepository***: Hierüber können Finanzpläne ausgelesen, angelegt, upgedatet und gelöschte werden.

Domain Services

Die Domain Services enthalten die Geschäftslogik. Bis auf den BudgetReportService, verwalten alle Services ihre entsprechenden Entitäten. Es wurden folgende Services angelegt:

- ***FinancialPlanService***

- ***RegularExpenseService:*** Beim Anlegen eines solchen Expense muss per FinancialPlanBudgetReport geprüft werden, ob noch genügend Budget frei ist.

- ***IrregularExpenseService:*** Hier spielen die beiden anderen Budget Reports eine Rolle. Denn eine unregelmäßige Ausgabe darf a) nicht den gesammt Budget (IrregularExpenseBudgetReport) und b) die der Kategorie aufbrauchen (CategoricalBudgetReport).

- ***CategoryService:*** Gleich zum RegularExpenseService. D.h. auch hier muss per FinancialPlanBudgetReport beim Anlegen geprüft werden.

- ***BudgetReportService:*** Hierüber können die BudgetReports berechnet werden.

Clean Architecture

Es wurden folgende Schichten der Clean Architecture implementiert. Dabei ist zu beachten, dass alle Schichten per Maven-Module implementiert wurde. Dabei wurde darauf geachtet, dass die korrekte Sichtbarkeit (von Außen nach Innen) eingehalten wird.

Domain - 3

Enthält alle Domain Objekte (ValueObjects, Entities, Aggregate). Außerdem wird für das Repo ein Interface programmiert, damit die Applikations Schicht mit dem Repository interagieren kann.

Außerdem ist folgende Besonderheit anzumerken: ValueObjects können bei Spring Data nicht ohne weiteres persistiert werden, wenn sie ein Attribute einer Entität sind. Eine Möglichkeit wäre dem ValueObject eine Id zuzuweisen. Eine bessere und auch hier implementierte, ist die Verwendung von Converter. Diese Konvertieren ValueObjects zu simplen Wert-Variablen um, was in diesem Fall bei allen ValueObjects möglich ist. Diese Umwandlung passiert kurz vorm persistieren im Repo. Dementsprechend müssten die Converter eigentlich in die Adapter oder Plugins Schicht. Jedoch müssen die Converter in den Entitäten angeben werden. Damit die korrekte Sichtbarkeit nicht verletzt wird, wurden sie deshalb in dieser Schicht platziert.

Application - 2

Hier wird die Geschäftslogik implementiert. Das bedeutet, dass alle geplanten Services hier implementiert werden. Dabei wurden die einzelnen Use Cases von der Services getrennt und in Interactoren verpackt. Damit repräsentiert jeder Interactor einen eigen UseCase. Dies hat 2 Vorteile:

1. Die Services, die häufig Geschäftslogik von vielen UseCases enthalten, können sehr unübersichtlich und groß werden. So kann direkt erkannt werden, welcher Interactor, welchen UseCase verwendet.

2. Es kann vorkommen, dass ein UseCase auf der Geschäftslogik eines anderen UseCases zugreift. Mittels Interactoren, können einfach andere UseCases in andere UseCases als Subroutinen eingebunden werden, ohne über den komplexen Service zu arbeiten.

Adapters - 1

In der Adpater Schicht, werden Objekte von internen, Domänenobjekte zu externen verwendeten Objekten transformiert. Dafür wird für jedes Entität, welche über ein Controller versendet oder empfangen wird, ein DTO (Data Transfer Object) angelegt. Für die Umwandlung, werden zwei Mapper (Entity -> DTO und DTO -> Entity) angelegt.

Plugins - 0

In der äußersten Schicht befindet sich einerseits die Verbindung zu den Repositories, als auch die Controller. Für das FinancialPlan-Repository wird zunächst das in Schicht 3 angelegte Interface implementiert. Dies ist jedoch nur eine Bridge-Object, denn die eigentlichen Repo-Funktionen werden automatisch durch Spring Data realisiert. Dafür muss nur das entsprechende JPA-Repository angelegt werden und anschließend von der Bridge aufgerufen werden. Spring Data regelt automatisch die Verbindung zur DB.

Für Controller werden die einzelnen REST-Schnittstellen angegeben werden. Außerdem wird in dieser Schicht die Verbindung zu Spring etabliert, da hier die Spring Application liegt.

Programming principles

SOLID

1. Single Responsibility Principle: Dies sagt aus, das jede Klasse oder Funktion nur in einem Aufgabenbereitch zuständig ist. Dies ist vor allem durch die Clean Architecture gegeben. So übernehmen Klassen wie Services nur die UseCases für einen ganz bestimmten Bereich.

2. Open Closed Principle: Hiermit wird gesagt, dass zwar Erweiterungen des Code zwar weiterhin möglich sein sollen, aber dass das sehr beschränkt und kontrolliert passieren soll. Das wurde gut umgesetzt, da durch die Services / Interactoren sehr einfach neue UseCases hinzugefügt werden können. Aber auch neue Änderungen bezüglich den Domain-Objekt-Regeln lassen sich (auch wenn das möglichst selten passieren sollte) hinzufügbar (durch neue assertions in den valueObjects)

3. Liskov Substitution Principle: Dieses Prinzip sagt aus, dass eine Klasse ohne Probleme durch eine andere Klasse des gleichen Types ausgetauscht werden kann. Auch das wird durch Clean Architecture gegeben. Nach den Eigenschaften von Clean Architecture sollen die äußersten Schichten sehr einfach austauschbar sein. So ist es relativ leicht, den DB-Treiber zu wechseln.

4. Interface Segregation Principle: Hier ist es wichtig, Interfaces möglich aufzuteilen und diese klein und spezifisch zu halten. Die verwendeten Interfaces sind allesamt sehr klein (FinancialPlanRepository oder Expense in der Domain Schicht).

5. Dependency Inversion Principle: Dieses Prinzip findet man so in der Clean Architecture. Es sagt lediglich aus, dass der High-Level Code vom Low-Level Code strikt getrennt werden muss. Es existieren klar erkenntliche Schnittstellen zwischen den einzelnen Leveln (z.B. Service: schnittstelle zwischen repos und entities)

GRASP

1. Low Coupling: Bei Low Coupling sollen die Klassen möglichst wenige Abhängigkeiten untereinander aufweisen. Ein negativBeispiel ist sind hier die Services. Obwohl zunächst eine klare Unterteilung und keine cross Abhängigkeiten

zwischen Interactoren existierte, hat sich das bei den letzten implementierten Use Cases verschlechtert. Der BudgetReportService wird bei vielen anderen Interactoren als Referenz aufgezeigt, was jedoch auch kaum zu unterbinden ist, da die geforderten Reports für die UseCases essentiell sind. Ein anderes Problem ist, das in Interactoren gleichzeit Services und andere Interacotren referenziert werden (siehe AddIrregularExpenseInteractor). Hier sollte wieder eine klare Zuordnung eingeführt werden. In den oberen und unteren Schichten, vor allem bei den Controllern, hat das Low Coupling deutlich besser funktioniert. So haben die Controller nur ihren zugehörigen Service für die Logik und die Mapper für die Umwandlung der DTOs.

2. High Cohesion: Dies ist im Grunde das Gegenteil, sprich wie viel Logik in einer einzelnen Klasse in sich geschlossen existiert. Die Interactoren BudgetReportService entsprechend diesem, da ihre Logik völlig unabhängig funktioniert und nur von außen aufgerufen wird. So haben sie keine Abhängigkeiten im Konstruktor.

DRY

DRY steht für Dont Repeat Yourself und steht, vermutlich selbsterklärend, für möglichst wenig Redundanz. Wie noch im Refactoring Teil beschrieben, gibt es hier ein paar Defizite im Bereich einiger Interaktoren. Jedoch hilft auf der anderen Seite die Adapter Schicht sehr stark dieses Prinzip einzuhalten. Hier wird das Mapping zwischen DTO und Entity einmalig definiert und kann bei den Controllern beliebig oft eingesetzt werden. So sammeln sich keine Konvertierungs-Methoden oder -Code in den Controllern, egal wie oft mit den Entity gearbeitet wird.

Testing

Refactoring

Es wurden folgende beiden Code Smells identifiziert und behoben durchgeführt

Long Method

In der Klasse AddRegularExpenseInteractor ist die execute Methode zu lange. Die gesamte Geschäftslogik des UseCase befindet sich dort. Deshalb wird die Funktion in 5 Funktionen aufgeteilt. Dabei hat jeder Subroutine eine eindeutige Aufgabe, welche schnell durch die klaren Namen verständlich werden.

Commit: cc3dc9b1f2d85581bcc0428dea959fd34539b0d0

Duplicate Code

Bei den beiden Klassen AddIrregularExpenseInteractor und DeleteIrregularExpenseInteractor wird eine Kategorie anhand einem Finanzplan und einer

CategoryId herausgesucht. Diese Code hat 4 Zeilen welche völlig identisch sind. Deshalb wird diese Funktion in einen eigenen Interactor ausgelagert.

Commit: 5c90283dfc1f1fdbf686b1f0b6a46cb56ef4ef8d

Es ist anzumerken, dass sowohl bei den Klassen der Long Method und der Duplicate Code Smells der jeweils andere noch angewendet werden könnte.

Ausblick

Ein Großes Problem der aktuellen Architektur ist, dass die IrregularExpensions zwar mit einem Datum versehen sind, aber ansonsten ein zeitlicher Kontext nicht gegeben ist. Alle anderen Entitäten sind allgemein gültig ohne zeitliche Beschränkung. Dadurch wäre es aktuell notwendig, alle IrregularExpensions per hand zu löschen um eine neue Monatsabrechnung zu starten.

Eine Möglichkeite wäre, den Finanzplan, die regelmäßigen Ausgaben und Kategorien als reine Blaupause zu sehen. Sobald eine IrregularExpense für einen neuen Monat gebucht wird, wird ein neues Zustandsobjekt des Finanzplan angelegt. Dieses Zustandsobjekt hat den Kontext des Finanzplan und eines Monats. Dadurch wäre eine einfache monatliche Abrechnung und Darstellung möglich, ohne alte Einträge zu löschen.

Jedoch selbst hier existiert ein Problem: wenn die Blaupause (also z.B. das verfügbare Budget des Finanzplan) geändert wird, funktioniert das zwar für aktuelle oder zukünftige Abrechnung, aber nicht für alte. Alte Monatsabrechnungen würde korrumpieren und evlt keinen gültigen Zustand mehr besitzen. Entweder müsster das Zustandsobjekt zuätlich alle Blaupausen Informationen enthalten, oder alter Monatsabrechnungen müssen als Report zusammengefasst und persistiert werden.