Dokumentation Programmentwurf

von

Tim Leistner

Mai 2021

Inhaltsverzeichnis

1 Projektübersicht

1.1 Allgemeine Beschreibung

Das Ziel dieses Programmentwurfs ist die Erstellung eines Prototyps für einen Finanzausgabenmanager. Der Prototyp soll als Desktopanwendung mit C# realisiert werden. Als Framework für die Benutzeroberfläche wird die Windows Presentation Foundation verwendet, kurz WPF. Der Prototyp beinhaltet die im nächsten Abschnitt beschriebenen Use Cases.

Der Quellcode ist im folgenden GitHub Repository abgelegt:
https://github.com/TimLeistner/AdvancedSoftwareEngineering

1.2 Use Cases

1.2.1 Erstellung von Kategorien

Der Nutzer ist kann auf einer dafür bereitgestellten Seite Kategorien erstellen, zu denen später die Ausgaben zugeordnet werden. Bei der Erstellung kann der Nutzer den Namen, die farbliche Darstellung, das monatliche Ausgabenlimit und die Währung der Kategorie bestimmen. Nach der Erstellung sind die Kategorien für die Dauer der Sitzung verfügbar.

1.2.2 Bearbeitung von Kategorien

Nach der Erstellung einer Kategorie kann der Nutzer alle bei der Erstellung gesetzten Eigenschaften bearbeiten.

1.2.3 Eingabe von Ausgaben

Nach der Erstellung einer Kategorie können der Nutzer ihr Ausgaben hinzufügen. Zu den Eigenschaften der Ausgaben gehört der Betrag, das Datum und eine optionale Beschreibung. Die Währung entspricht der der zugehörigen Kategorie.

1.2.4 Anzeige von Ausgaben

Es gibt für den Nutzer zwei Möglichkeiten sich Ausgaben anzeigen zu lassen.

1. Im Hauptfenster kann eine Kategorie und ein Monat ausgewählt werden. Es wird dann die Summe der Ausgaben in diesem Monat angezeigt und ob das monatliche Limit überschritten wurde.
2. Im Ausgabenübersichtsfenster kann eine Kategorie und ein Zeitraum ausgewählt werden. Es werden alle Ausgaben in dem gewählten Zeitraum, sortiert nach dem Datum, in einer Liste ausgegeben.

2 Domain Driven Design

2.1 Ubiquitos Language

- Kategorie/Category
  Eine Kategorie beinhaltet Ausgaben und das Limit der Ausgabensumme für einen Monat.
- Ausgabe/Spending
  Eine Ausgabe ist einer Kategorie zugeordnet und beinhaltet einen Geldwert, ein Datum und eine Beschreibung.
- Limit
  Ein Limit ein Limit gehört zu einer Kategorie und beschreibt den vom Nutzer gesetzten Höchstwert an Ausgaben in dieser Kategorie für einen Monat. Die Überschreitung verändert nur die visuelle Anzeige.

2.2 Taktische Muster

2.2.1 Value Objects

Value Objects im Domain Driven Design dienen dazu Einheiten eine Struktur zu geben. Für einen Finanzausgabenmanager ist Geld der wichtigste Wert, der festgehalten werden muss. Um Geld zu beschreiben, benötigt man einen Zahlenwert und die Währung des Geldes. Diese beiden Eigenschaften müssen zusammengeführt werden. Ein Geldwert muss nur festgehalten und nicht verändert werden. Bei einer Änderung wird ein neuer Geldwert abgespeichert. Deshalb eignet sich hierfür die Verwendung von Value Objects.

2.2.2 Entities

Entities bilden Strukturen ab, die sich während der Laufzeit des Programms verändern können und mehr Funktionalität beinhalten als nur das auslesen von Werten. Dies umfasst für dieses Projekt Kategorien und Ausgaben. Beides soll theoretisch vom Nutzer verändert werden können, nachdem eine Instanz vom Nutzer erstellt wurde.

Aktuell ist dies nur für Kategorien implementiert. Eine Kategorie muss veränderbar sein, damit der Nutzer Eingabefehler korrigieren kann. Man könnte zwar auch eine neue Kategorie erstellen, würde aber alle zugeordneten Ausgaben verlieren. Außerdem müssen diese Ausgaben auch zu einer Kategorie hinzugefügt werden, was wiederum eine Veränderbarkeit des Objektes voraussetzt. Bei Ausgaben ist theoretisch vorgesehen, dass die Beschreibung, das Datum und der Geldbetrag verändert werden kann.

2.2.3 Aggregates

Aggregate bestehen aus Entities und Kapseln deren Logik nach außen hin ab. Bei diesem Projekt gibt es nur ein Art von Aggregat, dass eine Kategorie als Zugangspunkt hat, über die auf die beliebig vielen zugeordneten Ausgaben zugegriffen wird. Auf die Ausgaben kann nicht direkt zugegriffen werden, die Kategorie bildet die Aggregate Root.

2.2.4 Repositories

Repositories dienen der Verwaltung von Aggregates. Über sie kann auf die Aggregate Roots zugegriffen und es können neue Aggregate erstellt werden. Da es nur ein Aggregate gibt, gibt es auch nur ein Repository in diesem Projekt. Hierbei handelt es sich um die Klasse CategoryRepository. Diese verwaltet eine Liste, in der die Kategorien gespeichert werden und stellt Funktionen zur Erstellung, Löschung und zum Abrufen der Kategorien bereit.

2.2.5 Domain Services

Domain Services bilden Funktionalitäten entweder Funktionen mit einem Interface ab, die dann von anderen Diensten implementiert und in der Domäne genutzt werden können, oder Funktionalitäten, die nicht genau einer Entity oder einem Value Object zugeordnet werden können. In diesem Projekt gibt es zwei Klassen, die Domain Services implementieren. Diese befinden sich in der Application Schicht und werden als SpendingTools zusammengefasst.

Sie bilden Funktionalitäten bearbeiten von Listen von Ausgaben an. Dies beinhaltet das herausfiltern von Ausgaben, die in einem bestimmten Monat getätigt worden, das sortieren der Liste nach Datum oder das bilden der Summe aller Ausgaben in einer Liste. Diese Funktionalitäten können nicht direkt der Ausgaben Entity zugeordnet werden, da diese immer nur eine Ausgabe beinhaltet. Man braucht aber die Informationen von vielen Ausgaben. Theoretisch könnten in der derzeitigen Implementierung auch Ausgaben aus verschiedenen Kategorien gemeinsam verarbeitet werden.

3 Clean Architecture

Das Thema dieses Kapitels ist die Erklärung der gewählten Schichtarchitektur für dieses Projekt. Es wurden folgende Schichten implementiert.

1. Domain Code
2. Application Code
3. Adapters
4. Plugin

Klassische Schichtarchitekturen enthalten weiterhin noch eine Abstraction Code Schicht, die eine Stufe unter der Domain Code Schicht liegt. Diese wurde zwar mit eingeplant, aber nicht implementiert, da für den Prototyp keine Funktionalität implementiert wurde, die dieser Schicht zuzuordnen wäre. Im Folgenden soll erklärt werden welche Teile des Projektes die einzelnen Schichten beinhalten und warum die Aufteilung in der Form erfolgte.

3.1 Domain Code

Diese Schicht stellt den Kern des Programms dar, mit Code der sich am wenigsten ändern sollte. Dies beinhaltet die in Kapitel 2 beschriebenen Value Object und Entities. Diese beschreiben die Kategorien und Ausgaben, welche den Kern der Anwendung darstellen. Egal welche Benutzeroberfläche oder weitere Funktionalität hinzukommt, diese Teile des Programms bleiben unverändert.

3.2 Application Code

Der Application Code beinhaltet die Umsetzung der Use Cases. Bestehender Code sollte sich kaum verändern. Aber wenn neue Funktionalitäten hinzukommen, werden sie hier implementiert. Der Code verändert sich öfter als der Domain Code und ist deswegen getrennt von diesem.

3.3 Adapters

Es wurde eine Adapterschicht implementiert, da die Benutzeroberfläche teilweise Ansprüche an Funktionalitäten hat, die anders im Application Code implementiert sind. Die Adapter dienen als Schnittstelle, um der Benutzeroberfläche die benötigten Funktionen bereitzustellen, ohne das Kernprogramm verändern zu müssen. Beispielsweise wird eine einzelne Funktion zur Bearbeitung von Kategorien bereitgestellt, die in den anderen Schichten in mehreren verschiedenen Funktionen implementiert ist. Dies verringert auch die Notwendigkeit für die Benutzeroberfläche, umfassende Kenntnis über die Implementierung in tieferen Schichten zu haben.

3.4 Plugin

Die Plugin-Schicht ist die dynamischste Schicht. Derzeit beinhaltet sie nur den Code für die Benutzeroberfläche. Durch die Entkopplung könnte das verwendete Framework jederzeit ausgetauscht werden, ohne etwas am Domain Code oder Application Code verändern zu müssen. Maximal die Adapterschicht müsste angepasst werden, wenn sich die Anforderungen dabei verändern. Bei einer umfangreicheren Implementierung des Finanzausgabenmanagers wäre hier z.B. auch der Code zum abspeichern und auslesen der Kategorien und Ausgaben in externen Dateien zu finden.

4 Programming Principles

4.1 SOLID

4.1.1 Single Responsiblity Prinicple

Nach dem Single Responsibility Principle soll jede Klasse eine klar definierte Aufgabe haben. Dadurch haben sie auch nur einen bestimmten Grund sich zu ändern, was zur niedriger Komplexität und Kopplung führt. Ein gutes Beispiel hierfür ist die Klasse *Money*. Dieses Value Object hat die Aufgabe Geldbeträge und ihre Währung abzubilden. Die Klasse würde sich nur ändern, wenn sich die Darstellung von Geld grundlegend verändert.

4.1.2 Open Closed Principle

Laut dem Open Closed Principle sollen Klassen offen für Erweiterungen und gleichzeitig immun gegen Änderungen sein. Dies wird in diesem Projekt vor allem durch den Einsatz von Interfaces erreicht. Für jede Funktion sind Interfaces definiert. Sollte es neue Anforderungen geben, wie diese Funktionen umgesetzt werden müssen, kann eine neue Klasse gegen das bestehende Interface implementiert werden, ohne den bestehenden Code ändern zu müssen.

4.1.3 Liskov Substitution Principle

Das Liskov Substitution Principle besagt, dass Objekte eines abgeleiteten Typs als Ersatz für Instanzen ihres Basistyps verwendet werden können müssen. Dieses Prinzip

findet bisher in diesem Projekt keine wirkliche Anwendung. Aufgrund der überschaubaren Komplexität des implementierten Modells, wurde keine Vererbung im Projekt angewendet. Deshalb kam das Liskov Substitution Principle nicht zum Einsatz.

4.1.4 Interface Segregation Principle

Durch die Umsetzung des Interface Segregation Principles soll verhindert werden, dass Klassen, die Interfaces implementieren, Funktionen beinhalten müssen, die sie eigentlich gar nicht benötigen. Zu diesem Zweck soll eher viele kleine Interfaces anstatt weniger großer implementiert werden. Dieses Prinzip wurde in diesem Projekt nicht verletzt. Aufgrund der geringen Komplexität der einzelnen Klassen wird immer nur ein Interface implementiert. Diese erfüllen gleichzeitig immer nur eine bestimmte Aufgabe, wodurch die Klassen keine für sie unnötigen Funktionen implementieren müssen.

4.1.5 Dependency Inversion Principle

Es soll keine direkte Abhängigkeit zwischen Low-Level und High-Level Module bestehen. Beide sollen von Abstraktionen abhängig sein. Dies ist in diesem Projekt der Fall. Einerseits sind alle Implementierungen von Interfaces abhängig. Weiterhin sorgt die Adapterschicht für eine Entkopplung der Abhängigkeit zwischen der Plugin- und der Appication-Schicht. Ein Beispiel hierfür ist die Klasse *CategoryAdapter*. Diese implementiert das Interface *ICategoryAdapter* und stellt die Funktionalitäten zum abrufen und verwalten der Kategorien zur Verfügung. Sollten sich die Anforderungen der Plugin-Schicht ändern, dann muss nur der Adapter verändert werden und nicht die unbedingt die Impementierung in den tieferen Schichten.

4.2 GRASP

4.2.1 Low Coupling

Ich denke, dass Low Coupling in diesem Projekt gewährleistet ist. Die meisten Referenzen bestehen zu den Entities und Value Objects, die zum Kern der Anwendung und der

Domäne gehören. Diese Abhängigkeiten sind nicht vermeidbar. Ansonsten sind durch den Einsatz von Interfaces die Abhängigkeiten zu bestimmten Implementierungen von Klassen sehr gering.

4.2.2 High Cohesion

Aufgrund der Anwendung der Prinzipien von Domain Driven Design hat jede Klasse eine bestimmte Aufgabe und enthält nur Elemente die semantisch zu ihr gehören. Ein Beispiel hierfür ist die Klasse *CategoryRepository*. Diese enthält eine Liste zur Verwaltung der erstellten Kategorien und Funktionen um Kategorien zu erstellen, zu löschen oder abzurufen. Alle diese Funktionen gehören zur Verwaltung von Kategorien. Andere Funktionalitäten sind in der Klasse nicht enthalten.

4.2.3 Information Expert

Das Prinzip Information Expert wird für die meisten Klassen eingehalten. Eine Ausnahme bildet die Klasse *Category*. Diese hat beinhaltet die Liste der Ausgaben für eine Kategorie, aber nicht die Funktionen zur Berechnung der Gesamtausgaben. Die Hilfsfunktionen für Ausgaben wurden in die Klassen *SpendingCalculator* und *SpendingSorter* ausgelagert. Dies hat den Zweck, dass diese bereitgestellten Funktionen auch auf Listen von Ausgaben angewandt werden können sollen, die außerhalb von Kategorien erstellt wurden. Beispielsweise eine Zusammensetzung der Ausgaben aus mehreren Kategorien oder nur die Auswahl einiger bestimmter Ausgaben.

4.2.4 Creator

Das Creator Prinzip wird in diesem Projekt eingehalten. Die Entities *Category* und *Spending* werden jeweils von nur einer anderen Klasse erstellt. Die Erstellung der Kategorien findet in der Klasse *CategoryRepository* statt, da diese alle Kategorien verwaltet. Die Erstellung der Ausgaben findet in der Klasse *SpendingEditor* statt, da dort die Nutzereingaben zu Erstellung neuer Ausgaben verarbeitet werden.

4.2.5 Indirection

Durch die Implementierung der verschiedenen Schichten im Rahmen von Clean Architecture wird Indirection umgesetzt. Da die inneren Schichten kein Abhängigkeiten zu äußeren Schichten haben, können die Implementierungen der äußeren Schichten ausgetauscht werden, ohne Änderungen an den inneren Schichten durchführen zu müssen.

4.2.6 Polymorphismus

In diesem Projekt gibt es keine Anwendungsfall von Polymorphismus, da die Domäne keine Variationen der einzelnen Subjekte beinhaltet. Bei der derzeitigen Komplexität der Anwendung war Vererbung nicht notwendig. Sollte sich der Funktionsumfang jedoch erweitern könnte Polymorphie eine Rolle spielen. Es ist beispielsweise denkbar, dass es verschiedene Ausprägungen der Klasse *Spending* geben könnte, die durch Polymorphie implementiert werden.

4.2.7 Controller

Das Controller Prinzip wird in der Plugin-Schicht umgesetzt. Zu jedem Use Case bzw. jedem UI-Fenster existiert eine Klasse, die die Nutzereingaben verarbeitet und an den Rest der Anwendung weitergibt. Beispielsweise die Klasse *CategoryEditor* verarbeitet die Eingabedaten im Category Editor und verwaltet die Anzeige der richtigen UI Elemente, je nachdem welcher Modus gewählt wird.

4.2.8 Pure Fabrication

Pure Fabrication wird im Projekt angewendet. Beispielsweise die Helferklassen *SpendingCalculator* und *SpendingSorter* dienen zur Trennung der benötigten Funktionalität bei der Verarbeitung von Ausgaben und der Verwaltung der Ausgaben selbst.

4.2.9 Protected Variations

Theoretisch wird Protected Variations angewandt, durch die Kapselung der Klassen hinter Interfaces. Die Implementierungen sind immer nur von den Interfaces der anderen Klassen abhängig. Praktisch gibt es aber noch keine Fall, in dem unterschiedliche Implementierungen eines Interfaces notwendig gewesen wären.

4.3 DRY

Wenn man DRY befolgt sollen jegliche Informationen einer Software einen genau bestimmten Platz haben und keine Wiederholungen geben. Diese Anforderung wird in diesem Projekt erfüllt. Zu jedem Zeitpunkt im Programm gibt es immer nur eine Quelle für Informationen. Dies kommt durch die Umsetzung von Domain Driven Design und Clean Architecture zustande.

5 Refactoring

5.1 Code Smells

5.1.1 Duplicated Code

Dieser Code Smell kann in der Klasse *CategoryEditor* in der Plugin-Schicht gefunden werden. In den Methoden *ClickEditMode* und *ClickCreationMode* wird die UI verändert. Dafür wird immer wieder überprüft, ob Objekte vorhanden sind oder nicht. Der Code hierfür ist immer der gleiche. Der Code Smell wurde im Commit am 12.05.2021 um 17:26 behoben.

5.1.2 Long Method

In der Klasse *Spending Overview* befindet sich die Methode *SpendingSelectionChanged*. Diese ist 40 Zeilen lang und besteht aus der Prüfung der UI-Eingaben, sowie des hinzufügens der Ausgaben zu einer Textbox. Durch den Umfang ist die Methode recht unübersichtlich. Dies soll durch das Refactoring im Commit am 12.05.2021 um 18:16 behoben werden.

5.2 Refactorings

5.2.1 Category Editor

Um den Code Smell Duplicated Code aus Abschnitt 5.1.1 wurde das Refactoring Ëxtract Methodängewandt. Es gibt zwei verschiedene Fälle, bei der Änderung der UI. Entweder

soll ein Objekt gelöscht oder es soll eins hinzugefügt werden. Deshalb wurde für jeden dieser Fälle jeweils eine neue Methode angelegt. Die Änderung ist im Commit am 12.05.2021 um 17:26 eingecheckt worden.

5.2.2 Spending Overview

In der Klasse Spending Overview wurden zwei Refactorings angewandt, um den Code Smell aus Abschnitt 5.1.2 zu beseitigen. Zunächst wurde das Refactoring "Replace Temp with Query"verwendet, um abgefragte Daten aus der UI nicht mehr in lokalen Variablen zu speichern. Dies wurde mit den Datumsangaben und der ausgewählten Kategorie gemacht, da diese Werte nur abgefragt, aber nicht manipuliert werden. Um die Methode noch übersichtlicher zu gestalten wurde zusätzlich noch das Refactoring Ëxtract Methodängewandt. Hierbei wurde die Prüfung, ob das Startdatum vor dem Enddatum liegt, in eine eigene Methode ausgelagert. Dies erhöht die Übersichtlichkeit der Ursprungsmethode. Diese Änderungen sind im Commit vom 12.05.2021 um 18:16 zu finden.

6 Entwurfsmuster

Entwurfsmuster sind allgemeine Lösungsansätze für häufige Probleme in der Programmierung. In diesem Projekt ist folgendes Problem aufgetreten. Die Benutzeroberfläche wird mit dem Framework WPF, Windows Presentation Foundation, implementiert. Wenn in WPF auf eine andere Seite navigiert wird ist es notwendig ein Objekt für diese Seite zu übergeben. Diese Objekte der Seiten brauchen Zugriff auf die Kategorien und die Ausgaben um sie anzeigen und manipulieren zu können.

Um diesen Zugriff zu gewährleisten, wurde das Singleton Entwurfsmuster verwendet. Ein Singleton Objekt hält eine statische Instanz von sich selbst bereit, die über eine öffentliche statische Methode abgerufen werden kann. Wenn noch keine Instanz vorhanden ist wird eine neue erstellt und ansonsten immer die gleiche zurückgegeben. Dadurch gibt es global nur eine Instanz dieses Objekts.

Dieses Entwurfsmuster wurde in der Klasse FinanceManager implementiert. Der Zweck dieser Klasse ist die Verwaltung und Bereitstellung der Repositories. Aktuell gibt es nur ein Repository, die Klasse CategoryRepository. Dadurch, dass es eine globale Instanz des der Klasse FinanceManager gibt, sind auch die Verweise auf die Repositories global verfügbar. Dies gibt die Möglichkeit in der Benutzeroberfläche, auch wenn bei der Navigation immer neue Objekte für die Seiten erstellt werden, jederzeit die aktuellen Daten abrufen zu können. Damit kann das zuvor beschriebene Problem gelöst werden. Die Veränderung des UML-Diagramms der FinanceManager Klasse ist in der Abbildung 6.1 erkennbar.

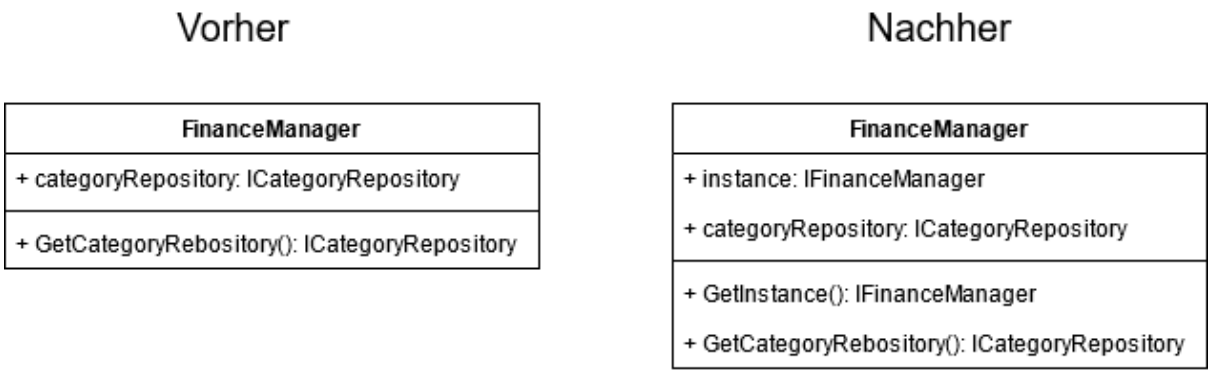

Abbildung 6.1: UML FinanceManager

Abbildungsverzeichnis