Advanced SWE Softwareprojekt - Dokumentation

Finanzmanager von Cora Hartmann & Laura Obermeyer

Inhaltsverzeichnis

[Clean Architecture 3](#_Toc73013969)

[Klassendiagramm vor der Clean Architecture Vorlesung 3](#_Toc73013970)

[Klassendiagramm mit Berücksichtigung der Clean Architecture 5](#_Toc73013971)

[Änderungen während der Implementierung 6](#_Toc73013972)

[Refactoring 6](#_Toc73013973)

[Refactoring der StartApplikations-Klasse 6](#_Toc73013974)

[Rename Method *emailAnlegen* 6](#_Toc73013975)

[Extract Method *startGuiBestimmenUndAufrufen()* 7](#_Toc73013976)

[Refactoring des EintraegeAnzeigenAdapters 8](#_Toc73013977)

[Domain Driven Design 8](#_Toc73013978)

[Bearbeiten und NeuAnlegen – 1 oder 2 Klassen? 8](#_Toc73013979)

[Analyse der Ubiquitous Language 10](#_Toc73013980)

[Analyse und Begründung der DDD Grundbausteine 10](#_Toc73013981)

[Repositories 10](#_Toc73013982)

[Aggregate 11](#_Toc73013983)

[Entities 11](#_Toc73013984)

[Value Objects 12](#_Toc73013985)

[Programming Principles 12](#_Toc73013986)

[SOLID 12](#_Toc73013987)

[GRASP 14](#_Toc73013988)

[DRY 14](#_Toc73013989)

[Entwurfsmuster 15](#_Toc73013990)

[Erbauer 15](#_Toc73013991)

[UML-Diagramm vor Einsatz des Erbauer Entwurfsmusters 16](#_Toc73013992)

[UML-Diagramm nach Einsatz des Erbauer Entwurfsmusters 17](#_Toc73013993)

[Beobachter (Observer) 18](#_Toc73013994)

[UnitTests 18](#_Toc73013995)

[ATRIP-Regeln 18](#_Toc73013996)

[Automatic 18](#_Toc73013997)

[Thorough 18](#_Toc73013998)

[Repeatable 19](#_Toc73013999)

[Independent 19](#_Toc73014000)

[Professional 20](#_Toc73014001)

[Einsatz von Mocks 20](#_Toc73014002)

[UnitTest mit Verwendung eines Spys 20](#_Toc73014003)

[UnitTests mit Verwendung von Mocks 20](#_Toc73014004)

[Code Coverage 21](#_Toc73014005)

Machine generated alternative text:
Unit Tests 
. 10 Unit Tests 
• ATRIP-Regeln 
• Code Coverage 
• Einsatz von Mocks 
Programming 
Principles 
• Analyse und 
Begründung für 
• SOLID 
• GRASP (Kopplung 
und Kohäsion) 
• DRY 
Praxisprojekt 
Code und schriftliche Dokumentation 
Objektorientierte Mainstream- 
Programmiersprache (bspw. Java, C#) 
• > 2k Zeilen Code 
. > 20 Klassen 
• Klar definierter (sinnvoller) Nutzen 
• Desktop-, Web- oder Mobilanwendung 
Entwurfsmuster 
. >= 1 Entwurfs- 
muster einsetzen 
und begründen 
• UML-Diagramm 
vorher und 
nachher 
Clean 
Architecture 
• Schichtarchitek 
tur planen und 
begründen 
. 2 Schichten 
umsetzen 
Refactoring 
• Code Smells 
identifizieren 
. 2 Refactoring 
anwenden und 
begründen 
Domain Driven Design 
• Analyse der 
Ubiquitous Language 
• Analyse und 
Begründung 
• Repositories 
• Aggregates 
• Entities 
• Value Objects 

**Link zu unserem Repository:**

<https://github.com/LauraObermeyer/SWEFinanzmanager>

# Clean Architecture

## Klassendiagramm vor der Clean Architecture Vorlesung

Machine generated alternative text:
EMail 
-vordererTeiI 
-domain 
-laenderkennzeic.hen 
Person 
-beschreibung 
Ko nto 
-kontostand 
-bezeichnung 
Eintra 
-bezeichnung 
-beschrei bung 
-d atum 
-prcH3uktIiste 
-bezeichnung 
-beschreibung 
Name 
-Vorname 
-nach name 
UebersichtsGLlI 
aenutzerAnI 
nGUl 
Au 
abenAnzei 
Wecker 
-limit 
-enddatum 
-bezeichnung 
-beschreibung 
einnahme 
ausgabe 
stemaenderun 
-zeitstempel 
EinnahmenAnzei 
Au 
abenEin 
nGUl 
nGUl 
nGUl 
benGUl 
EinnahmenEin 
orieAnI 
StatistikenGlJI 
nGUl 
csVReader 
GUIEvent 
IJIControI 
«nterface—— 
IGLlIEventListener 
« Interface» 
IGIJIEventSender 
likation 

Die optionalen Funktionalitäten wurden mit im Klassendiagramm modelliert. Ob sie später implementiert werden, ist noch offen. Es wird zu einem späteren Zeitpunkt ein Klassendiagramm der tatsächlich umgesetzten Klassen erstellt.

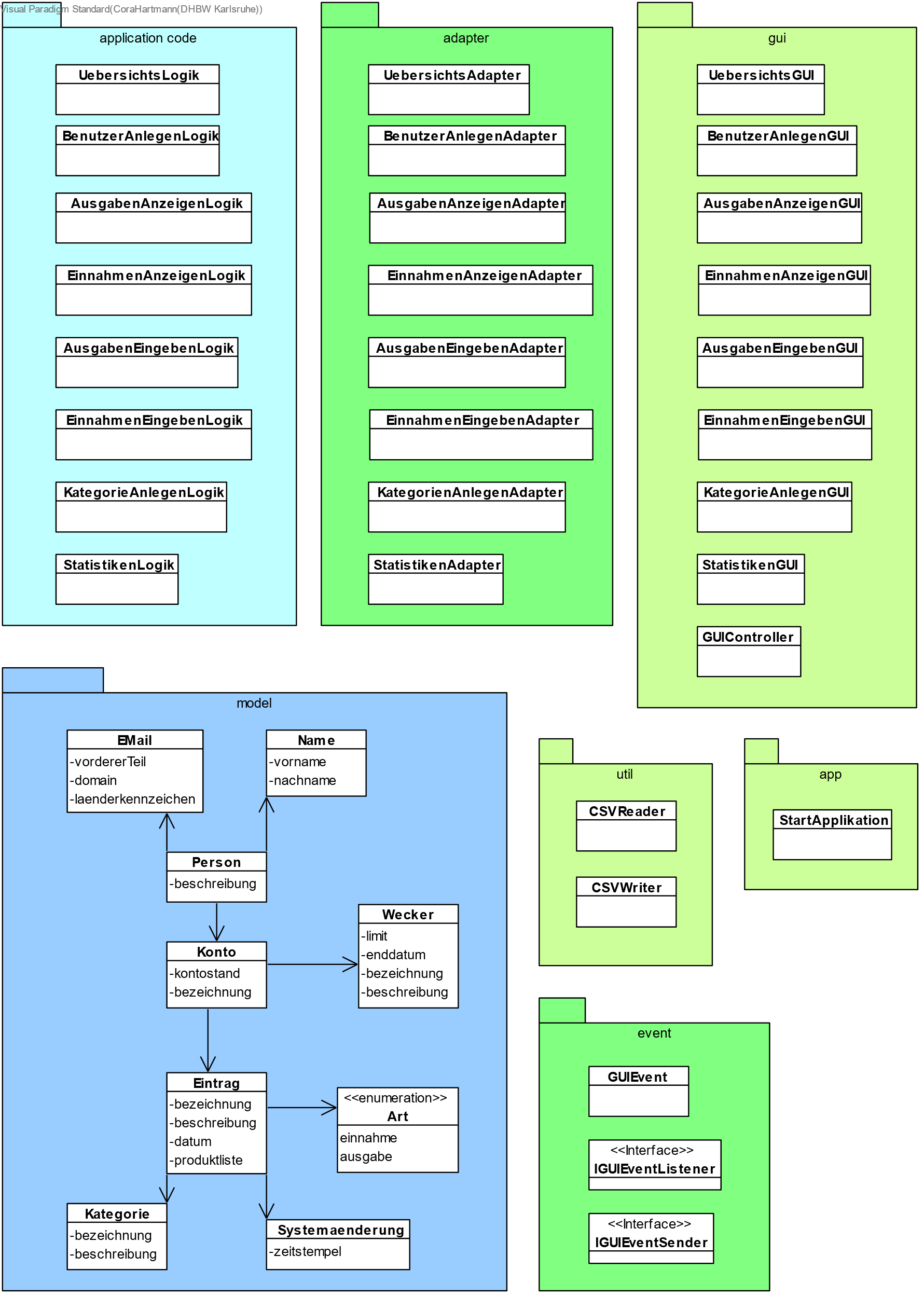
Wir nutzen das MVC Prinzip.

Diese erste Version des Klassendiagrams entstand, bevor die Clean Architecture in der Vorlesung behandelt wurde. Im Folgenden werden die notwendigen Anpassungen beschrieben, die sich nach dieser Vorlesung durch die Umsetzung der Clean Architecture ergaben:

* Es ist aufgefallen, dass die Aufteilung nicht klein/genau genug ist, insbesondere, dass die **GUI** Klassen mehr als eine Aufgabe hatten (Single Responsibility) und zu viel Logik enthielten, was sie gemäß der Clean Architecture nicht haben sollten. Sie sollten nur zum Anzeigen der Benutzeroberfläche genutzt werden. Die GUI Klassen werden daher aufgeteilt, sodass möglichst keine Logik in den **Plugins** vorhanden ist. Die Logik wird in die Schichten Adapter / ApplicationCode ausgelagert.
* Das **Model Paket** konnte so beibehalten werden. Die Klassen entsprechen Entities auf **Domain Code** Ebene der Clean Architecture.
* Die Klassen CSVReader und CSVWriter des **Util Pakets** sehen wir als **Plugins** an, da eine hohe Abhängigkeit zum Aufbau unserer csv-Dateien, die als Datenbank dienen, besteht. Eine Änderung der Dateien/des Dateiformats würde große Anpassungen des Codes der Plugin Klassen nach sich ziehen. Die Klassen sind also direkt von "der Außenwelt" abhängig.
* Das **Event Paket** ist der **Adapter** Schicht zugehörig, da die Klassen zur Kommunikation dienen und somit nicht zum anwendungsspezifischen Code des Application Codes gehören.
* Bisher wurden die GUIs zu Beginn in unserem **GUI Controller** instanziiert, der ebenfalls für die Kommunikation zwischen den GUI Klassen zuständig war. In Berücksichtigung der Clean Architecture werden die GUIs nun in der StartApplikation instanziiert. Der GUI Controller soll nur für die Kommunikation zuständig sein.

Es lässt sich festhalten, dass wir zunächst eigentlich nur mit drei Schichten, also ohne eine Adapterschicht, planen wollten, uns während der Implementierung aber noch für die Einführung einer Adapterschicht entschieden haben. Damit wird die Entkopplung besser dargestellt und der Application Code, also die Implementierung der Use-Cases weniger abhängig von der verwendeten Datenbank. Weiterhin ist zu sagen, dass diese Umsetzung sich nicht vollständig im Programmcode wieder spiegelt, dies gilt es, in Zukunft weiter umzusetzen. Ein Beispiel dafür, dass diese Trennung von Application Code und Plugins über die Adapter noch nicht vollständig ist, ist die Klasse *Eingeben*, welche den Use-Case des Anlegens bzw. Bearbeitens eines Eintrags darstellt. Derzeit wird noch direkt mit Hilfe des csv-Readers und csv-Writers Daten in den csv-Dateien geändert. Dieser Code sollte in einen Adapter ausgelagert werden, damit der Application Code mit den Use-Cases sauber getrennt und unabhängig von der restlichen Peripherie ist. Die Klasse *Eingeben* würde dann weiterhin das Erstellen eines neuen Eintrags mit den eingegebenen Eigenschaften übernehmen und dann auf einen Adapter zurückgreifen, um die Datenbank zu aktualisieren.

## Klassendiagramm mit Berücksichtigung der Clean Architecture



Zur besseren Ersichtlichkeit der Schichten der Clean Architecture wurden diese farblich eingefärbt. Dabei wurde sich an der Farbgebung aus der Vorlesung orientiert:

* Dunkelblau: Domain Code
* Hellblau: Application Code
* Dunkelgrün: Adapter
* Hellgrün: Plugins

Das *model* Paket stellt die geplanten Entitäten dar, welche im Domain-Code anzusiedeln sind, während die einzelnen Use-Cases, z.B. das Anlegen von Benutzern oder Einträgen.

# Änderungen während der Implementierung

Während der Implementierung wurde die Änderung vorgenommen, dass die Klasse Konto zunächst nicht umgesetzt wurde, sondern die Finanzverwaltung vorerst nur mit einem Konto zu führen ist, in welchem allgemein Einnahmen und Ausgaben als Einträge verwaltet werden können. Die Funktionalität des Weckers wird dementsprechend übertragen, dass z.B. bei Übersteigen eines festgelegten Betrags der Ausgaben einer bestimmten Kategorie der Wecker aktiviert wird und der Nutzer einen Hinweis bekommt. Zukünftig ist denkbar, dass die Funktion, die Finanzverwaltung mit verschiedenen Konten zu füllen, integriert wird.

Es ist weiterhin festzuhalten, dass noch nicht alle Funktionalitäten ausprogrammiert wurden. Diese könnten in Zukunft folgen. Dazu zählen die Filter-Funktionalität und die Wecker-Funktion.

# Refactoring

## Refactoring der StartApplikations-Klasse

Ziel dieses Refactorings war es, die StartApplikations-Klasse mit der Main-Methode lesbarer zu machen. Hierfür wurden einige Methoden und Variablen in sprechende Namen umbenannt und zudem zusammenhängende Code-Fragmente ausgelagert, um einen verständlichen Code ohne Inline-Kommentare zu erzeugen.

Im Folgenden werden Teile dieses Refactorings genauer erläutert.

Für weitere Informationen und den konkreten Code siehe den [Commit](https://github.com/LauraObermeyer/SWEFinanzmanager/commit/f81bea100e16fd5605d3033a3fdc2920ffc0148e).

### Rename Method *emailAnlegen*

Es wurde beispielsweise der Methodenname *emailAnlegen(String emailString)* in *emailDesBenutzersErzeugenAus(String emailString)* umgewandelt, da dieser die Intention der Methode besser wiedergibt. In der Methode geht es nicht, wie der frühere Methodenname fälschlicherweise vermuten ließ, um das Anlegen einer neuen Email-Adresse, sondern lediglich um das Erzeugen eines Email-Objekts eines bestimmten Benutzers aus einem String, indem dieser String in seine Bestandteile (Lokalteil, Hostname und Top-Level-Domain) aufgeteilt und dem Email-Konstruktor übergeben wird.

Um diese Intention deutlicher zu machen, wurden auch die lokalen Variablen innerhalb der Methode umbenannt:

private static EMail emailDesBenutzersErzeugenAus(String emailString){  
 String[] emailAdresseAufgeteiltInLokalUndDomänenteil   
 = emailString.split("@");  
 String[] domänenteilAufgeteiltInHostnameUndTopLevelDomain   
 = emailAdresseAufgeteiltInLokalUndDomänenteil[1].split("\\.");  
 EMail fertigeEmailAdresse = new EMail  
 (emailAdresseAufgeteiltInLokalUndDomänenteil[0],   
 domänenteilAufgeteiltInHostnameUndTopLevelDomain[0],   
 domänenteilAufgeteiltInHostnameUndTopLevelDomain[1]);  
 return fertigeEmailAdresse;  
}

### Extract Method *startGuiBestimmenUndAufrufen()*

Vor dem Refactoring sah die Main-Methode wie folgt aus:

public static void main( String[] args ) throws Exception {  
 if(*neuerNutzer*() == true){  
 *benutzerAnlegenGUI* = new BenutzerAnlegenGUI();  
 new GUIController(*benutzerAnlegenGUI*);  
 } else {  
 *buildUebersichtsGUI*();  
 new GUIController(*uebersichtsGUI*);  
 }  
}

Es waren bereits einige Code-Fragmente ausgelagert in die Methoden *neuerNutzer()* und *buildUebersichtsGui()*, dennoch war aufgrund der Verzweigung noch nicht auf den ersten Blick ersichtlich, was eigentlich getan wird. Statt einen Kommentar hinzuzufügen, wurde daher die komplette Verzweigung in die Methode *startGuiBestimmenUndAufrufen()* ausgelagert. So ist sofort klar, dass in der Methode bestimmt wird, mit welcher GUI die Anwendung starten soll und dass diese daraufhin aufgerufen wird.

Des Weiteren wurde die Methode *neuerNutzer()* in *nutzerIstNeu()* umbenannt, sodass die if-Anweisung *if(nutzerIstNeu() == true)* in der neuen Methode *startGuiBestimmenUndAufrufen()* selbstsprechend ist.

Auch die zwei Zeilen

*benutzerAnlegenGUI* = new BenutzerAnlegenGUI();  
new GUIController(*benutzerAnlegenGUI*);

und

*buildUebersichtsGUI*();  
new GUIController(*uebersichtsGUI*);

wurden jeweils in die Methoden *benutzerAnlegenGuiAufrufen()* und uebersichtsGuiAufrufen() ausgelagert, sodass auch hier verständlich ist, was getan wird.

Letztendlich sieht der behandelte Code dann wie folgt aus:

public static void main( String[] args ) throws Exception {  
 *startGuiBestimmenUndAufrufen*();  
}  
  
private static void startGuiBestimmenUndAufrufen() throws Exception {  
 if(*nutzerIstNeu*() == true){  
 *benutzerAnlegenGuiAufrufen*();  
 } else {  
 *uebersichtsGuiAufrufen*();  
 }  
}

## Refactoring des EintraegeAnzeigenAdapters

Die Klasse *EintraegeAnzeigenAdapter* bestand vor dem Refactoring im Wesentlichen aus einer sehr großen, 80 Zeilen langen Methode *getEintragListe(),* die von der Klasse *EintraegeAnzeigenGUI* aufgerufen wurde, um den Tabelleninhalt für die Übersichtstabelle der Ein- und Ausgaben zu erhalten. Diese Methode bestand zudem teilweise aus einem Code-Duplikat, da der Inhalt einer For-Schleife separat einmal für die Ausgaben und einmal für die Einnahmen ausgeführt wurde. Ziel des Refactorings war also die Beseitigung der Code Smells "Duplicated Code" und "Long Method".

Dies wurde erreicht, indem die Methode *getEintragListe()* aufgespalten und der doppelt vorhandene Code in die Methoden *auslesenAusCsvDatei(String art)* und *eintraegeGenerierenBasierendAuf(List<String[]> dateiInhalt)* zusammengeführt wurde. Mit der Übergabe des Paramters *art* bei der ersten Methode kann differenziert werden, ob es sich um Ausgaben oder Einnahmen handelt.

Zum Aufspalten der Methode wurde das Refactoring Extract Method angewandt, indem zusammenhängende Code-Fragmente in eigene Methoden mit sprechenden Namen ausgelagert wurden. Die lange Methode konnte so letztendlich auf folgende Methode reduziert werden:

public static String[][] getTabelleninhalt(){  
 *eintraegeGenerierenBasierendAuf*(*auslesenAusCsvDatei*("Ausgaben"));  
 *eintraegeGenerierenBasierendAuf*(*auslesenAusCsvDatei*("Einnahmen"));  
 return *tabelleninhaltAufbauen*();  
}

Neben dem Auslagern wurde auch der Methodenname geändert, da getTabelleninhalt() treffender ist, als getEintragListe(), weil der fertige Inhalt für die Tabelle, also die anzuzeigenden Attribute im richtigen Format zurück gegeben werden, und nicht lediglich eine Liste aller Einträge, wie dies der frühere Methodenname vermuten ließ.

Letztendlich konnte ein besser lesbarer und feingranularer Code erreicht werden, der zudem eine höhere Kohäsion hat.

Für die konkreten Codeänderungen siehe den zugehörigen [Commit](https://github.com/LauraObermeyer/SWEFinanzmanager/commit/0a90745485204121b31c7cd959d450bb24da4894).

# Domain Driven Design

## Bearbeiten und NeuAnlegen – 1 oder 2 Klassen?

Im Rahmen der Überlegungen zum Domain Driven Design (DDD) wollten wir die Mehtoden- und Klassennamen möglichst aussagekräftig wählen. Durch die Wahl von gleichen Begriffen in der Domäne und im Sourcecode sollte das Konzept der Ubiquitous Language umgesetzt werden, um den Übersetzungsaufwand für Domänenexperten zu reduzieren.

Bei der Plugin-Klasse "EingebenGUI" und der zugehörigen Application Code-Klasse "Eingeben" ist hierbei jedoch ein Problem aufgefallen. Diese Klassen spiegeln streng genommen zwei Use Cases wieder: "Eingeben" eines neuen Eintrags wird sowohl im Rahmen der Erstellung eines neuen Eintrags als auch beim Bearbeitungsvorgang eines bestehenden Eintrags getätigt. Gemäß der Domänensprache wären "NeuAnlegen" und "Bearbeiten" also noch aussagekräftigere Bezeichnungen für die Klassen (und für Teile der in den Klassen vorhandenen Methoden). Deshalb musste an dieser Stelle abgewogen werden, ob es sinnvoll ist die Klassen aufzuspalten.

Der ursprüngliche Grund für die Zusammenführung der Use Cases *Bearbeiten* und *NeuAnlegen* war es, redundanten Code zu vermeiden. **Duplicated Code** ist eines der wichtigsten Probleme, welches normalerweise im Rahmen von Refactoring behoben wird. Nun extra durch ein Refactoring doppelten Code einzubauen wäre widersprüchlich.

Eine Überlegung *gegen* die Zusammenführung der beiden Klassen war jedoch das **Single Responsibility Prinzip**, wonach jede Klasse nur eine Verantwortlichkeit haben sollte. Aus technischer Sicht ist dies hier verletzt, da z.B. bei *NeuAnlegen* eine neue Zeile in einer CSV-Datei generiert wird, während bei *Bearbeiten* eine bestehende Zeile verändert wird. Jedoch kann man aus konzeptioneller Sicht argumentieren, dass in beiden Fällen lediglich ein Eintrag in einen neuen Zustand gebracht wird. Bearbeiten ist letztendlich auch nur das Neu Anlegen eines bereits vorhandenen Eintrags. Weil sich bei NeuAnlegen und Bearbeiten aus der fachlichen Sicht zum gleichen Zeitpunkt ein Eintrag ändert, hat die Klasse in dieser Hinsicht nur einen Grund zu existieren, wonach das Single Responsibility Prinzip nicht verletzt ist.

Führt man diesen Gedanken weiter, sieht man, dass das Zusammenlegen der beiden Klassen sogar den positiven Effekt hat, eine **Shotgut Surgery** zu vermeiden, welche in Zukunft auftreten könnte. Eine Shotgun Surgery liegt dann vor, wenn eine fachliche Änderung eine Anpassung von mehreren unterschiedlichen Stellen im Sourcecode erfordert. Dies wäre hier z.B. der Fall, wenn der Kunde ein neues Feld wünscht und dann Modifikationen in mehreren Klassen notwendig wären, von denen keine vergessen werden dürfte.

Nach dieser Abwägung wurde entschieden die beiden Use Cases Bearbeiten und NeuAnlegen zusammenzuführen. Durch die Entkräftung des Arguments mit dem Single Responsibility Prinzips sprach lediglich noch die konsequente Verwendung der Ubiquitous Language dagegen. Da "Eingeben" aber auch ein Begriff aus der Domänensprache ist, den der Kunde versteht, wogen die Probleme Duplicated Code und Shotgut Surgery ungleich schwerer und die Zusammenlegung der Klassen erschien somit als sinnvoll.

Dies ist ein Beispiel für eine Stelle, wo die Konzepte des Domain Driven Designs nicht vollständig zufriedenstellend umgesetzt werden konnten, weil andere Argumente schwerwiegender waren und dagegen sprachen.

## Analyse der Ubiquitous Language

Ziel des Domain Driven Designs und der Ubiquitous Language ist es, den Übersetzungsaufwand zwischen dem Sourcecode und der Domäne möglichst gering zu halten. Dies kann erreicht werden durch die Verwendung von gleichen Begriffen in Code und Domäne. Ubiquitous Language ist die gemeinsam verwendete Sprache von Entwicklern und Domänenexperten.

Daher war es zunächst von Nöten die Fachsprache der Domäne, der Finanzwelt, zu verstehen und nicht zu versuchen, diese Fachsprache in eigene Begriffe zu übersetzen. Da aufgrund der Rahmenbedingungen und Durchführung des Projekts im Rahmen einer Vorlesung, kein Kontakt zu "echten" Domänenexperten bestand, musste die Fachsprache selbst angeeignet bzw. nach bestem Wissen umgesetzt werden.

Im Kapitel des Refactorings finden sich Beispiele, wie die Ubiquitous Language konkret umgesetzt wurde. Besonders bei Klassen- und Methodennamen wurde darauf geachtet, Domänenbegriffe zu verwenden und nicht in eine eigene "Entwicklersprache" zu verfallen.

## Analyse und Begründung der DDD Grundbausteine

### Repositories

Die beiden verwendeten EntityManager zum Verwalten der Benutzer bzw. Einträge können als Repositories des Domain Driven Design angesehen werden.

Sie bieten eine Schnittstelle der Anwendung zur Datenbank, in unserem Fall die csv-Dateien, ohne, dass die Anwendung wissen muss, wie die Speicherung aussieht. Der "Vorratsschrank" (EntityManager) verwaltet die entsprechenden Aggregate. Bei der Erstellung der Interfaces, die im Domain Code liegen, wurde darauf geachtet, die Ubiquitous Language zu verwenden.  
Im Interface *EintragEM* wurden zunächst mehr Methoden definiert, als wir zur konkreten Umsetzung benötigen. In Zukunft könnten jedoch weitere Funktionalitäten umgesetzt werden.

Außerdem denkbar, ist die Modellierung eines weiteren Repositories zur Verwaltung der

Kategorien. Der Plan hierzu ist, dass der Nutzer zukünftig eigene Kategorien erstellen kann

und um diese verwalten zu können, bietet sich die Nutzung eines Kagetorie-Repositories an.

Die Repositories werden über ein Interface definiert, das im Domain Code angesiedelt wurde. Die konkrete Implementierung wurde in der Adapter-Schicht der Clean-Architecture platziert. Es kam die Überlegung auf, die Implementierung der Repositories in den Application Code zu verschieben, da diese dort verwendet werden. Letztendlich wurde sich dagegen entschieden und die Implementierung in der Adapter-Schicht belassen, da es sich aus unserer Sicht um einen Aufruf und keine direkte Abhängigkeit handelt, so dass der Clean-Architecture mit der Regel, dass äußere Schichten von den Inneren abhängen und nicht andersherum, unserer Meinung nach nicht widersprochen ist. Des Weiteren würde sich diese Überlegung bezüglich der Platzierung der Repositories erübrigen, wenn die striktere Trennung von Application Code und Adapter-Schicht umgesetzt wird, wie zu Beginn im Kapitel *Clean Architecture* beschrieben, da dann der Zugriff auf die Repositories mit in die Adapter-Schicht ausgelagert werden könnten. Bevor dies umgesetzt wird, sollte an dieser Stelle noch eine tiefere Überlegung erfolgen, falls doch etwas dagegen sprechen sollte.

### Aggregate

Entities können zu gemeinsam verwalteten Einheiten gruppiert werden. Dies findet auch in unserem Softwareprojekt Anwendung. So kann allgemein jede Entity als Aggregat betrachtet werden, im Zweifelsfall besteht das Aggregat nur aus diesem einen Entity und die Entity entspricht ebenfalls der Aggregat Root Entity.

Bei uns sind jedoch auch zusammengesetzte Aggregate zu finden:

Die Entities *Benutzer* und *Email* sind als ein Aggregat zusammengefasst und werden durch ein Repository verwaltet (s. vorherigen Abschnitt). Dabei ist der Zugriff über den Benutzer geregelt.

Ein weiteres Aggregat bilden die Entities *Eintrag* und *Systemaenderung*. Die Rolle der Aggregat Root Entity übernimmt die Entity *Eintrag*. An dieser Stelle könnte theoretisch die Entity *Kategorie* ebenfalls zu diesem Aggregat zählen. In unserer Anwendung wäre dies jedoch nicht mehr der Fall, sobald in Zukunft dem Nutzer die Möglichkeit geboten wird, eigene Kategorien anzulegen. Hierfür wäre ein direkter Zugriff auf die Entity *Kategorie* notwendig, der unabhängig der Einträge ist. Das Aggregat wird ebenfalls über ein Repository verwaltet.

### Entities

Entities bilden eine kontinuierliche Existenz ab, deren Eigenschaften und Werte innerhalb des Lebenszyklus des Entities verändert werden können. Identifiziert werden Entities durch eine ID, umgesetzt durch Schlüssel.

Im *model*-Paket unseres Projekts können die von uns verwendeten Entities gefunden werden: *Benutzer, Eintrag, EMail, Kategorie* und *Systemaenderung*.

Jede dieser Entities hat veränderliche Eigenschaften, einen Lebenszyklus und repräsentiert etwas der Domäne. So steht die Entity *Eintrag* für eine Einnahme oder Ausgabe, die von einer Person getätigt wurde und mittels einer *Kategorie* kategorisiert werden kann. Die Entity *Benutzer* vertritt den Anwender der Applikation, der seine Finanzen anhand von Einnahmen und Ausgaben überwachen möchte. Eine *EMail* stellt eine Kontaktmöglichkeit zur nutzenden Person dar. Die *Systemaenderung* zeigt den Zeitpunkt der Anlage eines Eintrags oder der letzten Änderung an.

Die Wahl der Schlüssel wird für die einzelnen Enitites im Folgenden kurz erläutert:

* Einträge werden über einen künstlichen Schlüssel (Surrogatschlüssel) mit einer UUID unterschieden. Entschieden wurde sich für eine UUID, die jederzeit generierbar und anwendungsübergreifend (oder benutzerübergreifend) eindeutig ist, da eine Kombination von Attributen oder einzelne Attribute eines Eintrags sich nicht als Schlüsselkandidaten eignen. Der Nutzer soll nicht eingegrenzt werden, indem z.B. die Bezeichnung eines Eintrags oder die Kombination aus Bezeichnung und Art eines Eintrags nur einmal vorhanden sein darf.
* Benutzer werden ebenfalls und aus gleichen Gründen wie die Einträge über eine UUID als Surrogatschlüssel identifiziert. Hinzu kommt, dass der Fall, dass Nutzer gleiche Namen haben, nicht ausgeschlossen werden darf, so dass die Kombination des Vor- und Nachnamen als Schlüssel nicht empfehlenswert ist.
* Eine Kategorie wird derzeit über einen natürlichen Schlüssel, über ihre Bezeichnung, von anderen unterschieden. Wenn in Zukunft die Kategorie ebenfalls mittels eines Repositories verwaltet wird, sollte ggf. ebenfalls auf einen Surrogatschlüssel umgestiegen werden.
* Die Email eines Nutzers wird anhand der Kombination aller Attribute (also der vollständigen Email) identifiziert, um die Eindeutigkeit zu gewährleisten. Ein Nutzer kann ebenfalls nur eine Email-Adresse in unserer Finanzverwaltung anlegen, so dass eine Unterscheidung derzeit nicht notwendig ist.
* Eine Systemänderung wird über den natürlichen Schlüssel des Zeitstempels identifiziert, da ein Zeitstempel eindeutig ist.

Zukünftig ist denkbar, zusätzlich zu den bereits vorhandenen Entities, die Entity *Konto* einzubinden. Der Anwender würde dadurch die umfangreiche Möglichkeit erhalten, seine Finanzen sortiert nach spezifischen Konten zu überwachen. Mögliche natürliche Schlüssel der Klasse Konto könnten die IBAN oder Konto-Nummer sein, da diese bereits eindeutig sind und Begriffe aus der Domänensprache sind und somit die Anforderungen der Ubiquitous Language erfüllen.

### Value Objects

In unserem Software-Projekt sind derzeit keine Value Objects vorhanden. Die Eigenschaften jedes unserer Objekte können im Laufe des Lebenszyklus geändert werden. Dies entspricht dem Gegenteil von Value Objects, da diese ihre Eigenschaften bzw. ihren Wert nach der Erzeugung nicht mehr ändern.

Potentielle Kandidaten für Value Objects stellen in unserer Finanzverwaltung z.B. die Eigenschaften *Betrag* und *Datum* dar, die Teil eines Eintrags sind. Da wir dem Nutzer die Möglichkeit, Anpassungen bezüglich seiner Einträge zu machen und fehlerhafte Einträge zu verbessern, nicht nehmen wollten, sind beide Eigenschaften (*Betrag, Datum*) auch im Nachhinein noch änderbar. Dafür existieren Setter-Methoden in der Eintrag-Klasse.

# Programming Principles

## SOLID

**Single Responsibility**: Dieses Prinzip basiert auf der einzigen Zuständigkeit einer Klasse, so dass eine Klasse nur einen einzigen Grund haben sollte, sich zu ändern. Während der Implementierung wurde das Prinzip weitestgehend beachtet, so dass eine Klasse immer eine Zuständigkeit bzw. Funktionalität umsetzt. Jedoch sind auch Fragestellungen aufgekommen, mit denen es sich auseinander zusetzen galt. Diese wurden bereits im Kapitel *Domain Driven Design* vorgestellt, als es darum ging, die Use Cases *Bearbeiten* und *Anlegen* in einer gemeinsamen Klasse oder in zwei Klassen umzusetzen. Es wurde entschieden, beide Use Cases in einer Klasse umzusetzen, mit der Argumentation, dass aus konzeptioneller Sicht in beiden Fällen lediglich ein Eintrag in einen neuen Zustand gebracht wird. (Für weitere Ausführungen s. Kapitel *Domain Driven Design – Bearbeiten und Neu anlegen – 1 oder 2 Klassen?).*

Auch bei der Auseinandersetzung mit der Clean Architecture im ersten Kapitel wurde das Single Responsibility Prinzip beachtet, als es um die Aufteilung der GUI Klassen ging, die zu viel Logik enthielten und mehr als eine Aufgabe hatten. Dies stellte eine Verletzung des Single Responsibility Prinzips dar, weil die Klassen sozusagen "Zuständigkeitsklumpen" waren, sprich Klassen mit vielen Zuständigkeiten. Im überarbeiteten Klassendiagramm nach Berücksichtigung der Clean Architecture haben wir die GUI Klassen feinteiliger gestaltet, um das Single Responsibilty Prinzip zu erfüllen.

**Open/Closed**: Klassen, etc. sollten offen für Erweiterungen, aber abgeschlossen für Veränderungen sein. Veränderungen spiegeln sich z.B. in geänderten Anforderungen wieder. Bei neuen Anforderungen sollte der Code der Klasse, etc. nicht verändert werden müssen. In Anbetracht der Tatsache, dass noch nicht alle Funktionalitäten vollständig implementiert sind, sind zukünftige Änderungen der Klassen und Entitäten derzeit nicht auszuschließen. Ziel ist es dabei, den Code bei Fertigstellung so gestaltet zu haben, dass sich ändernde Anforderungen abgefangen werden und z.B. durch zusätzliche neue Funktionalitäten umgesetzt werden können.

**Liskov Substitution:** Es gilt zu beachten, dass laut diesem Prinzip Objekte des Software-Projekts durch ihre Subtypen ersetzt werden können sollen, ohne, dass sich die Korrektheit des Programms ändert. Da in unserem Software-Projekt sehr wenig bis gar nicht mit Vererbung gearbeitet wurde, können hierzu in Bezug auf unser Projekt keine Aussagen gemacht werden. Letztendlich sollte jedoch bei Einführung von Vererbung darauf geachtet werden, dass sich die Subtypen genauso verhalten, wie ihr Basistyp, also die Funktionen des Basistypen erweitert und nicht einschränkt.

**Interface Segregation:** Das Interface Segregation Prinzip unterstützt das Single Responsibility Konzept. Schnittstellen sollten möglichst passgenau auf den Client zugeschnitten sein, so dass verschiedene spezifische Interfaces anstatt eines einzigen, das alles definiert, verwendet werden. In unserer Implementierung sind zwar nicht viele Interfaces vorhanden, dennoch kann gesagt werden, dass z.B. für die Repositories für Benutzer und Eintrag verschiedene Schnittstellen definiert wurden, die speziell auf den Benutzer bzw. Eintrag zugeschnitten sind.

**Dependency Inversion**: Die Dependency Inversion stellt das Prinzip der Entkopplung dar. Abhängigkeiten auf konkrete Klassen stellt eine hohe Kopplung dar. Klassen höherer Ebenen sollten nicht von Klassen niedriger Ebenen abhängig sein, sondern beide von Interfaces. So kann auch die Abhängigkeit auf konkrete Klassen aufgelöst werden, indem Dependency Injection angewandt wird. Wir haben darauf geachtet, dass die Klassen im Kern nicht von den Klassen in der Peripherie abhängig sind, wie auch im ersten Kapitel zur Clean Architecture beschrieben. Mit Interfaces haben wir nicht gearbeitet, da wir hierfür bei der Größe des Programmentwurfs keine Anwendung sahen. Eine Möglichkeit, die wir in Erwägung zogen, war, bei Einnahmen und Ausgaben mit einem Interface "Eintrag" zu arbeiten. Da diese beiden Klassen aber sowieso zu einer gemeinsamen Klasse Eintrag zusammengefasst wurde, erübrigte sich das Interface.

## GRASP

Low Coupling – High Cohesion stellt das Grundkonzept der General Responsibility Assignment Software Patterns/Principles dar.

**Kopplung:** Kopplung ist das Maß für die Abhängigkeit einer Klasse von ihrer Umgebung. Wir haben versucht, lose Kopplung zu erreichen, indem wir mit Events gearbeitet haben, die über den GUIController laufen. Ziel war es, dass sich Sender und Empfänger nicht mehr kennen. So können beispielsweise die Events "Abbrechen" und "Speichern" von verschiedenen GUIs (Sendern) getriggert werden, ohne dass dies für den GUIController von Relevanz ist. Wir haben jedoch entschieden, dass wir ein bisschen Kopplung in Kauf nehmen, da dies viel Arbeit erspart. Hauptziel war demnach, die Kopplung im Griff zu haben, nicht jedoch vollständig entkoppelt zu programmieren.

**Kohäsion:** Kohäsion ist das Maß für den inneren Zusammenhalt einer Klasse. Es geht hierbei um die semantische Nähe der Methoden und Attribute einer Klasse. Ein Problem ist hierbei, dass dies nicht technisch messbar ist.

Wir haben beispielsweise bei der Gestaltung unserer Klasse "Benutzer" auf eine hohe Kohäsion geachtet, indem wir dieser nur Attribute gegeben haben, die einen Benutzer direkt identifizieren (id, vorname, nachname, email). Wir haben bewusst darauf verzichtet, dem Benutzer eine Kontonummer zuzuweisen, obwohl bis jetzt jeder Benutzer nur ein Konto haben kann. Dies kann sich aber in Zukunft ändern. Wir wollten die Typen Benutzer und Konto trennen, da durch die Erweiterung des Typsystems eine Erhöhung der Kohäsion erreicht werden kann.

## DRY

„Don’t repeat yourself“ – Dieses Prinzip einzuhalten, erleichtert das Warten des Codes. Das Vermeiden von Redundanzen hilft dabei, dass Anpassungen im Code nur an einer Stelle gemacht werden müssen, anstatt an mehreren, wenn dieser Code-Ausschnitt an verschiedenen Stellen zu finden ist.   
Während der Implementierung wurde versucht, gleiche Codestellen in eine gemeinsame Methode auszulagern, um die Redundanz der Codestelle zu vermeiden. Im Refactoring wurde zum Beispiel ein gefundener Code Smell des „Duplicated Codes“ beseitigt. Für weitere Ausführungen, s. Kapitel *Refactoring – Refactoring des EintraegeAnzeigenAdapters*.

Insgesamt wurde im Software-Projekt an verschiedenen Stellen gleiche Codestellen in eine separate Methode ausgelagert, um Redundanzen zu vermeiden und das DRY-Prinzip einzuhalten.

Ein weiteres Beispiel hierfür ist in der Klasse *Eingeben* zu finden, die die beiden gemeinsam implementierten Use-Cases Bearbeiten und Neu anlegen darstellt. In beiden Fällen, Bearbeiten und Neu anlegen eines Eintrags, wird der Inhalt der csv-Dateien zu aufgebaut und dies jeweils mit dem entsprechenden Eintrag, der bearbeitet oder neu angelegt wurde. Diese Funktionalität wurde in die Methode *zeilenInhaltAufbauen(…)* ausgelagert, um den Code nicht doppelt vorhanden zu haben.

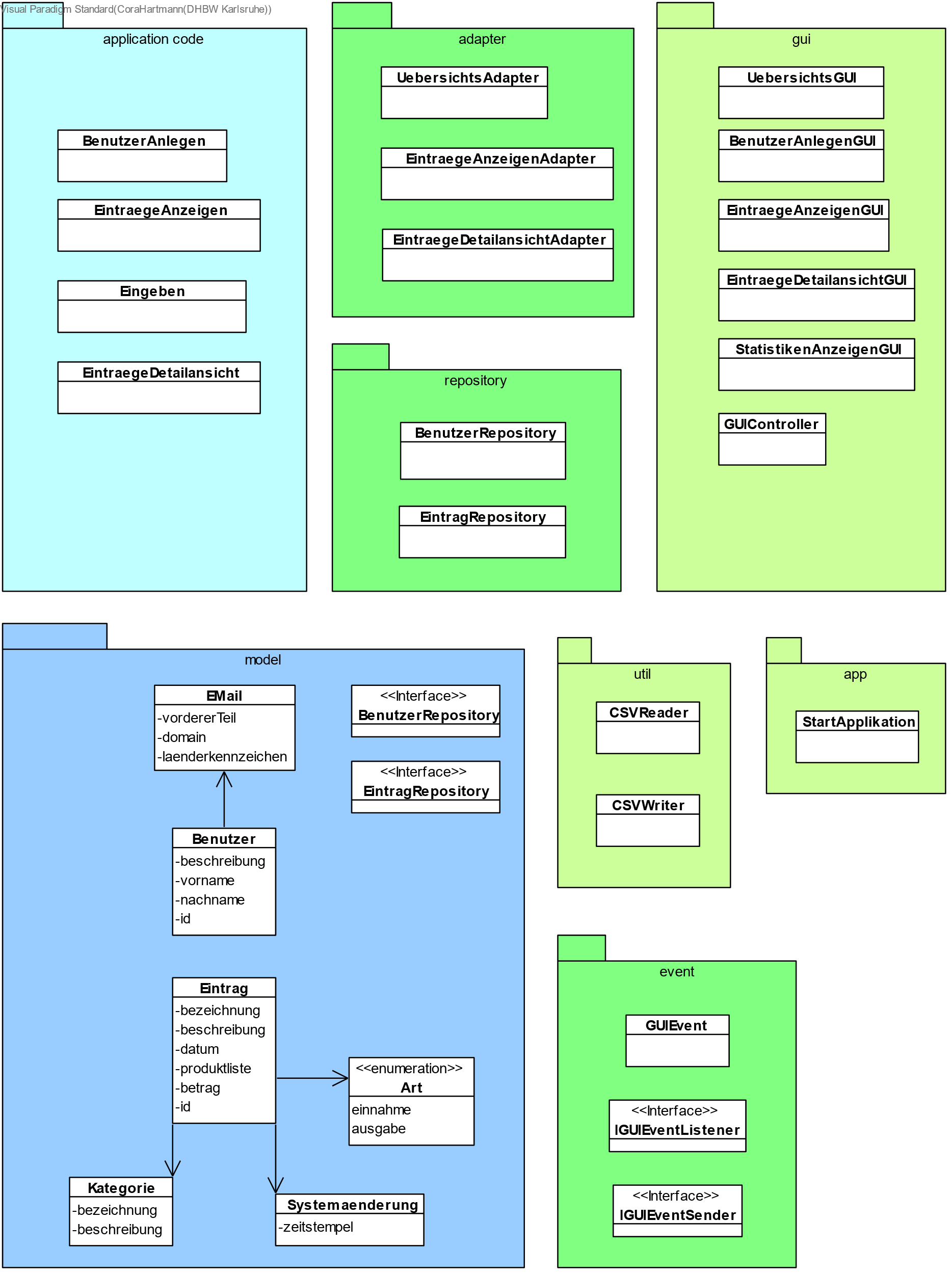
# Entwurfsmuster

## Erbauer

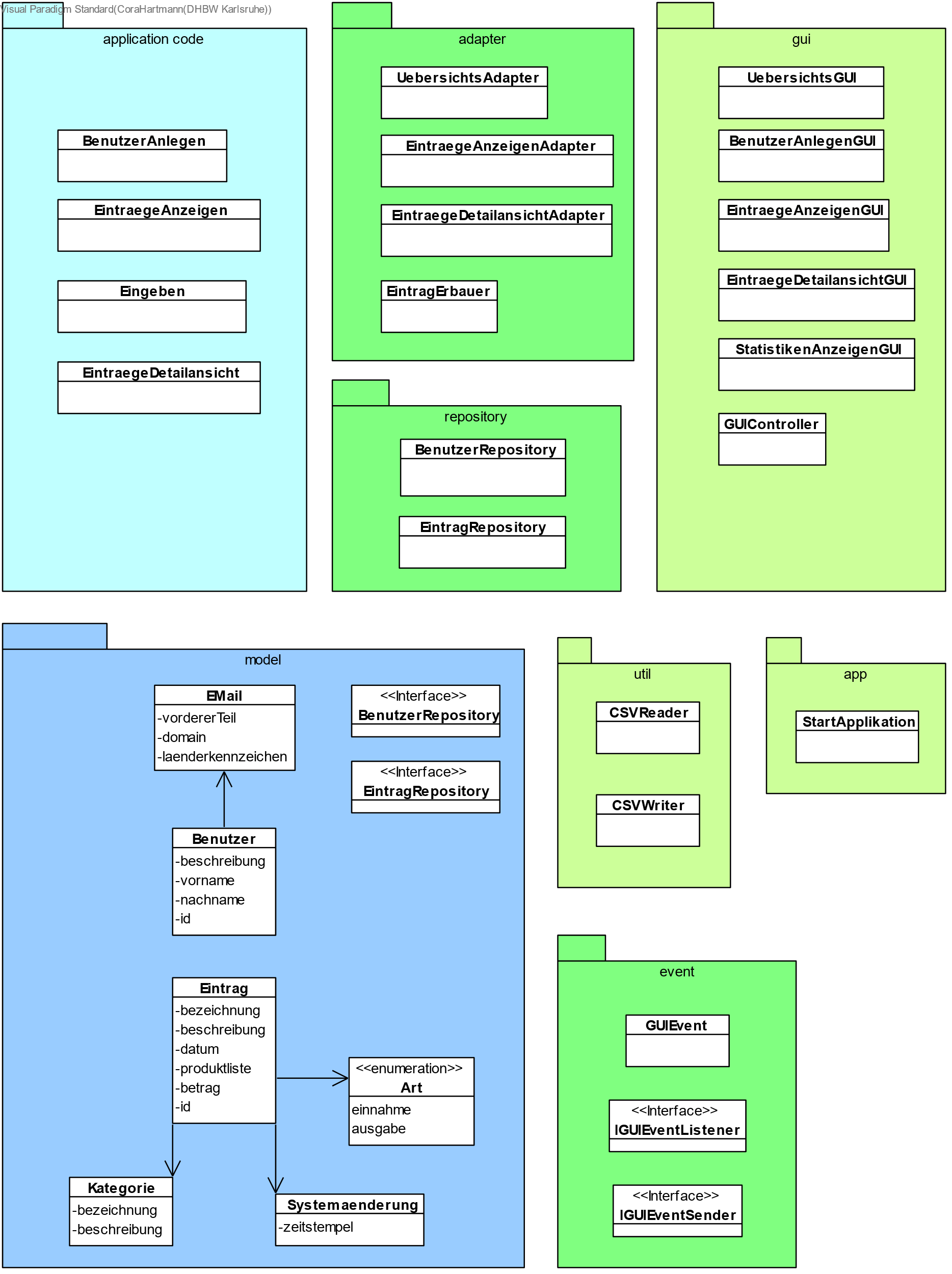
In unserem Software-Projekt findet das Entwurfsmuster des Erbauers Anwendung. Wir haben uns für den Einsatz eines Eintrags-Erbauers entschieden, da so an einer Stelle Einträge mit verschiedenen Eigenschaften erstellt werden können. Vorteil ist, dass für z.B. UnitTests default-Einträge leicht erzeugt werden können. Dies wird in den vorhandenen UnitTests bereits genutzt. Der zukünftige Plan ist, den Eintrags-Erbauer ebenfalls zur Erstellung der Einträge im Anwendungscode zu nutzen. Dafür wurde in der Eintrags-Erbauer Klasse bereits eine Methode (*mitAllenEintraegen()*)geschrieben, die diesen Anwendungsfall übernehmen soll. Dabei wurde darauf geachtet, dass der Erbauer für die Formatierung und Erstellung der Datumsangabe aus einem übergebenen String zuständig ist, damit die Erstellung des Datums nicht an verschiedenen Stellen im Anwendungscode stattfinden muss, sondern nur zentral im Erbauer. Die Verwendung des Eintrags-Erbauers in den UnitTests kann mit folgendem [Commit](https://github.com/LauraObermeyer/SWEFinanzmanager/commit/17b3fe4559918afb7d6f8a9c921ecd14c9b1b0ab) nachvollzogen werden. Die UnitTests sind in diesem Commit nicht in der aktuellsten Version, da weitere Änderungen vorgenommen wurden. Im UnitTest-Abschnitt kann ein Commit mit der aktuellsten Version der Tests gefunden werden.  
  
Der Erbauer wurde zunächst in der Clean Architecture im Domain Code angesiedelt. Da letztendlich aber kein Anwender des Erbauers im Domain Code vorhanden ist, wurde der Erbauer in die Adapter-Schicht ausgelagert, um ebenfalls möglichst viel Code in den äußeren Schichten ansiedeln zu können. Die Klassen des Projekts, die den Erbauer in Zukunft zur Erstellung von Einträgen nutzen sollen, befinden sich ebenfalls in der Adapter-Schicht.

Im Folgenden sind die Klassendiagramme vor und nach dem Einsatz des Erbauers zu sehen. Diese unterscheiden sich etwas von den Klassendiagrammen in dem Kapitel zur Clean Architecture, da es sich dort um die Planungs-Klassendiagramme handelte, während hier die Klassendiagramme an die tatsächliche Umsetzung angepasst wurden und optionale Funktionalitäten wegen Übersichtlichkeitsgründen aus der Modellierung entfernt wurden.

### UML-Diagramm vor Einsatz des Erbauer Entwurfsmusters



### UML-Diagramm nach Einsatz des Erbauer Entwurfsmusters



## Beobachter (Observer)

Wir haben uns für den Einsatz des Entwurfsmusters Beobachter entschieden, um die Kommunikation der GUIs zu entkoppeln. Die Kommunikation sollte nur im Änderungsfall stattfinden statt periodischen Anfragen. Wir haben dazu die Interfaces *IGUIEventListener* und *IGUIEventSender* sowie die Klasse *GUIEvent* erstellt. Die GUI-Klassen implementieren jeweils das *IGUIEventListener* Interface, während der GUIController das Interface *IGUIEventSender* implementiert. Den GUI-Klassen (Subjekte) können Beobachter angemeldet bzw. abgemeldet werden mit den Methoden addListener(IGUIEventListener listener) und removeListener(IGUIEventListener listener). Subjekte können ihre Beobachter über die Methode fireEvent(GUIEvent event) benachrichtigen. Vorteil ist die lose Kopplung zwischen Subjekten und Beobachtern.

# UnitTests

Zum Testen der korrekten Implementierung der Funktionalitäten wurden insgesamt 22 UnitTests geschrieben. Es sind damit jedoch nicht alle Klassen mit UnitTests abgedeckt, sondern nur ausgewählte Klassen bzw. Methoden getestet worden.

Für den Einsatz von Mocks wurde auf das Mock-Framework Mockito zurückgegriffen, das in der Vorlesung bereits vorgestellt wurde.

Der aktuellste Stand aller UnitTests ist unter diesem [Commit](https://github.com/LauraObermeyer/SWEFinanzmanager/commit/cfd61b33c7720dd7e5b1c6a630c3209c64fc2460) zu finden. Die im weiteren Verlauf des Texts verlinkten Commits zeigen die jeweiligen ersten Commits der Tests, diese haben sich jedoch im Laufe der Implementierung etwas geändert, z.B. durch die Einführung des Erbauers, so dass der hier angegebene Commit alle aktuellen Versionen enthält.

## ATRIP-Regeln

Während des Schreibens der UnitTests wurde darauf geachtet, die ATRIP-Regeln so weit wie möglich zu berücksichtigen.

### Automatic

Jeder Test ist eigeständig durchführbar und auch seine Ergebnisse werden eigeständig überprüft. Hierzu wurden neben den Bestandteilen *Arrange* und *Act* eines Tests, im Bestandteil *Assert* sog. Assertions verwendet, die überprüfen, ob das erhaltene Ergebnis dem entspricht, das erwartet wurde. Es ist daher kein manuelles Eingreifen in den Testablauf notwendig.

### Thorough

Es ist wichtig, dass alle Fehlerfälle abgefangen werden und jede missionskritische Funktionalität getestet ist. In unserem Software-Projekt sind bisher, aufgrund der Vielzahl an Klassen, nicht alle Funktionalitäten vollständig von UnitTests abgedeckt. Dies kann als Ziel angesehen werden, in Zukunft fehlende Tests zu ergänzen und auch für jeden vorhandenen Bug mindestens einen Test hinzuzufügen, so dass jeder mögliche Fehlerfall von einem UnitTest abgedeckt ist.

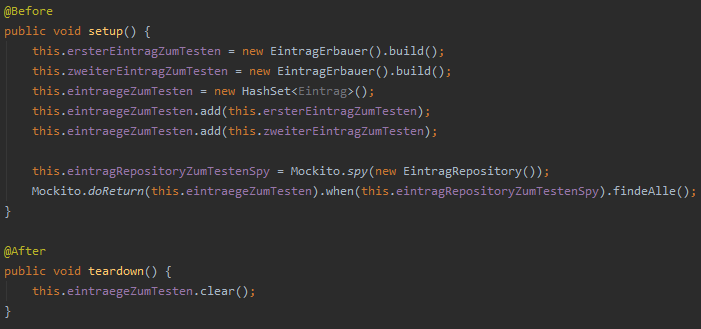
Als Beispiel für eine (fast) vollständig getestete Klasse kann der Eintrag-Erbauer angesehen werden. Es wurden UnitTests für jede Methode des Erbauers, bis auf die build()-Methode geschrieben, welche lediglich einen Eintrag aus den vorher gesetzten Eigenschaften erstellt. Es wurden so 82% des Codes der Klasse getestet. Weitere Beispiele zur Code Coverage können in einem folgenden Abschnitt gefunden werden.

### Repeatable

Ein UnitTest sollte jederzeit durchführbar sein und stets das gleiche Ergebnis liefern. Daraus folgt, dass Abhängigkeiten von z.B. der Umgebung zu vermeiden sind. Eine Problemquelle bieten Datum und Zufalls. In unserem Software-Projekt wird sowohl mit zufällig generierten UUIDs gearbeitet, als auch mit Datumsangaben. Aufgrund der Tatsache, dass in unserem Fall bisher kein Test abhängig von z.B. der aktuellen Systemzeit ist, können weitere Probleme umgangen werden. In Zukunft ist bei Erstellung weiterer Tests oder Funktionalitäten, welche die Zeitstempel, etc. betreffen darauf zu achten, Abhängigkeiten zu vermeiden.

### Independent

Damit alle Tests jederzeit in beliebiger Reihenfolge und Zusammenstellung funktionsfähig sind, wurde darauf geachtet, keine Abhängigkeiten zwischen den einzelnen Tests zu erstellen. Teil dessen wäre z.B. das Rückgängigmachen von Änderungen von gemeinsam genutzten Ressourcen direkt im Anschluss an den Test, welcher die Änderungen vorgenommen hat. Dieser Fall, das mehrere UnitTests auf gleiche Ressourcen zugreifen ist in der Testklasse *EintragRepositoryTest* zu finden. Diese Klasse testet die Funktionalitäten des Eintrags-Repositories. Da jede zu testende Methode auf die Methode des Eintrag-Repositories *findeAlle()*, die alle vorhandenen Einträge zurückliefert, zugreift, wurde diese mit Hilfe eines Spys gemockt, um die vorhandenen Einträge zu Testzwecken selbst definieren zu können. Diese Einträge und auch der Spy werden als Vorbereitung vor jedem Test in der Methode *setup()* erstellt und nach jedem Test wird die Liste der Einträge, die bei *findeAlle()* zurückgegeben werden soll, geleert.



Außerdem wurde darauf geachtet, dass jeder UnitTest einen einzelnen Aspekt (z.B. einen möglichen Fall einer Methode) der Komponente abdeckt. So ist z.B. die Ursache für einen fehlgeschlagenen Test leicht zu finden.

### Professional

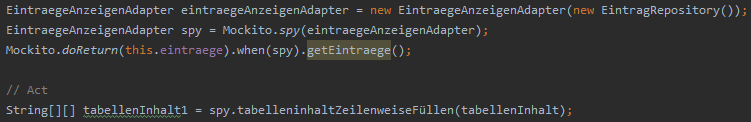
Da der in Tests geschrieben Code ebenfalls produktionsrelevant ist, ist es wichtig, den TestCode so leicht verständlich, wie möglich zu gestalten. Hierzu wurde, wie im gesamten Software-Projekt, die Domain-Sprache so gut, wie möglich eingesetzt und außerdem auf aussagekräftige Variablen- und Methodennamen geachtet. Aus diesem Grund wurden so gut wie keine Kommentare zu Erklärungszwecken benötigt, da der Code durch bloßes Lesen verrät, welche Funktionalität abgedeckt wird.

## Einsatz von Mocks

Bei den geschriebenen UnitTests werden an verschiedenen Stellen sowohl Mocks, als auch Spys eingesetzt. Der Unterschied besteht darin, dass ein Mock eine gesamte Klasse mit all ihren Methoden mockt und ein Spy einzelne Methoden einer realen Klasse mocken kann. Der Vorteil eines Spys ist, dass so eine Methode der zu testeten Klasse gemockt werden kann. Dies war für unser Software-Projekt an vielen Stellen notwendig, um Klassen testen zu können. Für Beispiele s. die folgenden Abschnitte.

### UnitTest mit Verwendung eines Spys

Der Spy wurde verwendet, um das Verhalten einer einzelnen Methode eines realen Objekt zu ändern. Im Beispiel wird die Methode *tabellenInhaltZeilenweiseFüllen* des *EintraegeAnzeigenAdapter*s getestet. Dabei werden mit der Methode *getEintraege()* die vorhandenen Eintraege zurückgegeben, um die im UI anzuzeigende Tabelle zeilenweise mit diesen Einträgen zu füllen. Um zu testen, ob die Methode richtig funktioniert, mussten selbst erstellte Einträge verwendet werden, um die Eigenschaften mit dem Zeileninhalt abgleichen zu können. Dazu wurde mit Hilfe eines Spys die Methode *getEintraege()* so verändert, dass selbst erstellte Einträge zurückgegeben wurden.



Der letzte Commit, der den UnitTest vervollständigt hat, ist unter folgendem Link zu finden: [Commit](https://github.com/LauraObermeyer/SWEFinanzmanager/commit/3d79aff5c6645dd4fdaecfbb807d5d30e74299b1)

### UnitTests mit Verwendung von Mocks

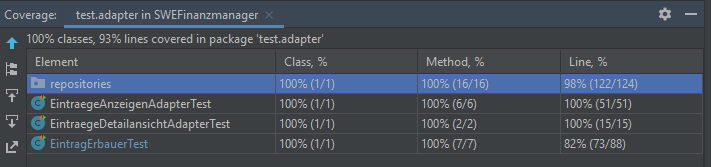
Beispiele für die Verwendung von Mocks bei der Erstellung von UnitTests sind die beiden Tests *tabellenInhaltAufbauenMitEintraegen()* und *tabellenInhaltAufbauenOhneVorhandeneEintraege()* der EintraegeAnzeigenAdapter-Testklasse. In der originalen Methode  *tabellenInhaltAufbauen()* werden Methodenaufrufe des Eintrags-Repositories abgesetzt. Um die Methode dennoch zu testen, ohne ein echtes Eintrag-Repository zu benötigen, wurde das Eintrags-Repository gemockt. Es sind zwei Tests zu der originalen Methode vorhanden, welche zwei unterschiedliche Szenarien prüfen – das Aufbauen des Tabelleninhalts ohne und mit vorhandenen Einträgen. Die Erstellung des Mocks des Eintrag-Repositories, der speziell die Methode *liefereAnzahlEintraege()* mockt, wurde, wie in folgendem Screenshot zu sehen, umgesetzt:



Beide UnitTests können unter folgendem Commit gefunden werden: [Commit](https://github.com/LauraObermeyer/SWEFinanzmanager/commit/446a291aa68461fb3fbe97b3abddf73dc53fd2eb)

## Code Coverage

Wie in folgender Abbildung zu sehen, decken die erstellten Testklassen nicht den gesamten Code ab:



Dies ist darauf zurückzuführen, dass aufgrund der Vielzahl an Klassen und möglichen Szenarien bzw. Fehlerfällen nicht alle davon abgedeckt werden konnte. Demnach kann eine Coverage der getesteten Klassen von 98%, 100%, 100% und 82% als vorerst zufriedenstellend erachtet werden. Zukünftig sollte jede Funktionalität umfangreich mit allen eintretbaren Fällen getestet werden, um die korrekte Implementierung der Komponenten sicherzustellen.

Jedoch ist auch zu bedenken, dass eine 100% Coverage nicht bedeutet, dass alle Funktionalitäten fehlerfrei implementiert sind. Man kann eine hohe Coverage erreichen und dabei nicht alle möglichen Fehlerfälle bedacht haben. Des Weiteren können sich ändernde Anforderungen dazu beitragen, dass eine Implementierung tendenziell nie zu 100% als fehlerfrei bezeichnet werden kann. In unserem Software-Projekt sind beispielsweise auch nicht alle möglichen Fehlerfälle abgefangen und auf diese getestet worden. Dies gilt es in zukünftigen Projekten umzusetzen.