Advanced SWE Softwareprojekt - Dokumentation

Finanzmanager von Cora Hartmann & Laura Obermeyer

Inhaltsverzeichnis

[Clean Architecture 2](#_Toc72750853)

[Klassendiagramm vor der Clean Architecture Vorlesung 2](#_Toc72750854)

[Klassendiagramm mit Berücksichtigung der Clean Architecture 4](#_Toc72750855)

[Refactoring 4](#_Toc72750856)

[Refactoring der StartApplikations-Klasse 4](#_Toc72750857)

[Rename Method *emailAnlegen* 4](#_Toc72750858)

[Extract Method *startGuiBestimmenUndAufrufen()* 5](#_Toc72750859)

[Refactoring des EintraegeAnzeigenAdapters 6](#_Toc72750860)

[Domain Driven Design 6](#_Toc72750861)

[Bearbeiten und NeuAnlegen – 1 oder 2 Klassen? 6](#_Toc72750862)

[Analyse der Ubiquitous Language 7](#_Toc72750863)

[Analyse & Begründung 7](#_Toc72750864)

[Repositories 7](#_Toc72750865)

[Aggregate 8](#_Toc72750866)

[Entities 8](#_Toc72750867)

[Value Objects 9](#_Toc72750868)

[Programming Principles 9](#_Toc72750869)

[SOLID 9](#_Toc72750870)

[GRASP 9](#_Toc72750871)

[DRY 9](#_Toc72750872)

[Entwurfsmuster 9](#_Toc72750873)

[Erbauer 9](#_Toc72750874)

[UnitTests 10](#_Toc72750875)

[ATRIP-Regeln 10](#_Toc72750876)

[Einsatz von Mocks 11](#_Toc72750877)

[Code Coverage 12](#_Toc72750878)

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 

**Wir hangeln uns am MVC Prinzip lang.**

**Startklasse als Plugin:**

* Bisher Teile der notwendigen Logik im Controller. Auslagern?
* Im Kern nochmal Main-Methode,

Während der Implementierung wurde die Änderung vorgenommen, dass die Klasse Konto zunächst nicht umgesetzt wurde, sondern die Finanzverwaltung zunächst imaginär nur mit einem Konto zu führen ist, indem allgemein Einnahmen und Ausgaben 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.

# 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 

* Diese erste Version des Klassendiagrams entstand bevor wie die Clean Architecture in der Vorlesung behandelt haben. Im Folgenden werden die notwendigen Anpassungen beschrieben.
* Die optionalen Funktionalitäten wurden mit im Klassendiagramm modelliert. Ob sie später implementiert werden, ist noch offen.
* Wir haben gemerkt, dass die Aufteilung nicht klein/genau genug ist, insbesondere, dass unsere **GUI** Klassen mehr als eine Aufgabe haben (Single Responsibility) und zu viel Logik haben, was sie laut Clean Architecture nicht haben sollten. Sie sollten nur zum Anzeigen der Benutzeroberfläche genutzt werden. Unsere GUI Klassen werden daher aufgeteilt, so dass möglichst keine Logik in den Plugins vorhanden ist. Die Logik wird in die Schicht Adapter/ApplicationCode ausgelagert
* Unser **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 würde eine komplette Anpassung des Codes mit sich ziehen. Die Klassen sind also direkt von "der Außenwelt" abhängig.
* Prüfen, ob wirklich keine Anwendungsspezifische Logik vorhanden ist; Single Responsibility erfüllt?; alle Methoden der Klassen benötigt?
* **Paket Event:** Vermutlich Adapter, da die Klassen zur Kommunikation dienen und somit nicht zum anwendungsspezifischen Code des Application Codes gehören. Wenn wir Adapter und Application Code nicht unterscheiden, ist es doch auf einer Ebene.
* 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 wird ein StartController eingeführt. Die StartApplikation ganz außen enthält die Main Methode, welche die Applikation startet und mit dem StartController für die Instanziierungen der Plugins zuständig ist. Der GUI Controller soll nur für die Kommunikation zuständig sein.

**Wenn wir die Adapter Schicht noch einführen sollten, sind die Controller ein Adapter und nicht im Application Code**

## Klassendiagramm mit Berücksichtigung der Clean Architecture

Machine generated alternative text:
application Code 
EMail 
-vordererTeiI 
-domain 
-laenderkennzeic.hen 
Person 
-beschreibung 
Ko nto 
-kontostand 
-bezeichnung 
Eintra 
-bezeichnung 
-beschreibung 
-d atum 
produktliste 
Ka orie 
-bezeichnung 
-beschreibung 
Name 
-Vorname 
-nachname 
Uebersichts 
BenutzerAnI 
ik 
UebersichtsGLlI 
BenutzerAnI 
nGUl 
Au 
abenAnzei 
Wecker 
-limit 
-enddatum 
-bezeichnung 
-beschreibung 
einnahme 
ausgabe 
stemaenderun 
-zeitstempel 
ik 
benLo 
Au 
abenAnzei 
EinnahmenAnzei 
ik 
ik 
ik 
EinnahmenAnzei 
Au 
abenEin 
Au 
abenEin 
nGUl 
nGUl 
nGUl 
EinnahmenEin 
EinnahmenEin 
orieAnI 
StatistikenGlJI 
benGUl 
nGUl 
orieAnI 
Statistiken L 
csVReader 
controller 
GUIEvent 
GLIIControIIer 
« Interface» 
StartControIIer 
IGLlIEventListener 
« Interface» 
IGLlIEventSender 
likation 
ik 

**Paket Application Code:**

* Diese Klassen repräsentieren die überwiegenden UseCases, die wir in unserer Anwendung umsetzen wollen, wie z.B., Ausgaben hinzufügen, aktuelle Kontoübersicht anzeigen, etc.

# 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

## Analyse & Begründung

### 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.

### 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 ist 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.

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.

**TODO: Schlüsselwahl erklären**

### 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?)*

**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.

**TODO: Wie sieht das mit der Dependency Inversion bei uns aus?**

## GRASP

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

**TODO: Wie sieht das mit Low Coupling und High Cohesion bei uns aus?**

## 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.

TODO: ANSIEDELUNG DES ERBAUERS IN DER CLEANARCHITECTURE

# UnitTests

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

### 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.

#### 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 ein Test hinzuzufügen.

#### 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 unserem Software-Projekt bislang nicht häufig vorkommen, ist aber denkbar, wenn in Zukunft die Testabdeckung erhöht würde.

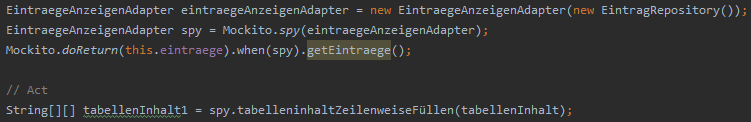
Außerdem wurde darauf geachtet, dass jeder UnitTest einen einzelnen Aspekt (z.B. Methode) der Komponente abdeckt. So ist z.B. die Ursache für einen fehlgeschlagenen Test leicht zu finden.

#### 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.

### Einsatz von Mocks

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 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, wurde das Eintrags-Repository gemocked. 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()* mocked, 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