Schriftliche Ausarbeitung

Domain Driven Design (DDD)

Analyse der Ubiquitous Language

Die Ubiquitous Language ist ein Konzept des Domain Driven Design, welches Verständnisprobleme zwischen Entwicklern und Domänenexperten, durch das Definieren einer gemeinsamen Projektsprache, vorbeugen soll. Die Projektsprache beinhaltet Definitionen für das gemeinsame Verständnis von Konzepten, Prozessen und Regeln aus der Domäne.

Wichtige Begriffe und Prozesse innerhalb der Problemdomäne "Aufgabenliste" sind:

- Aufgabe (Task)
- Fälligkeitsdatum (Due Date)
- Aufgabenliste (Task List)
- Arbeitsschritt (Sub Task)
- Erinnerung (Reminder)
- Liste anlegen
- Aufgabe verschieben/löschen/bearbeiten
- Liste löschen/bearbeiten

Aufgabe (Task):

Eine Aufgabe besteht aus Name/Titel, Erinnerungsdatum, Arbeitsschritte, Beschreibung/Notiz, Fälligkeitsdatum. Zudem hat eine Aufgabe mehrere mögliche Zustände:

- Erledigt und nicht erledigt
- Fälligkeitsdatum überschritten/nicht überschritten

Alle Bestandteile, außer dem Namen, sind optional. Die Anzahl Arbeitsschritte ist logisch nicht begrenzt.

Fälligkeitsdatum (Due Date):

Ein Fälligkeitsdatum kann auf einer Aufgabe gesetzt werden. Es ist darauf zu achten, dass das Fälligkeitsdatum nicht in der Vergangenheit liegen darf. Ist das Fälligkeitsdatum auf einer Aufgabe erreicht muss dies in der Aufgabe gekennzeichnet werden. Als Fälligkeitsdatum werden nur Jahr, Monat und Tag betrachtet.

Aufgabenliste (Task List):

Eine Aufgabenliste kann mehrere (theoretisch unbegrenzt viele) Aufgaben beinhalten. Sie kann aber auch keine Aufgabe beinhalten. Aufgabenlisten können vom Benutzer erstellt werden. Dabei wird ein Name vom Benutzer festgelegt, welcher über alle Listen eindeutig sein muss. Die Bestandteile einer Aufgabenliste sind folglich ihr Name und eine Reihe von Aufgaben.

Arbeitsschritt (Sub Task):

Ein Arbeitsschritt kann selbst wieder als Aufgabe verstanden werden, mit dem Unterschied, dass ein Arbeitsschritt keine weiteren Arbeitsschritte enthalten darf.

Erinnerung (Reminder):

Eine Erinnerung kann auf einer Aufgabe gesetzt werden. Es ist erneut darauf zu achten, dass das Erinnerungsdatum nicht in der Vergangenheit liegt. Ist das Erinnerungsdatum auf einer Aufgabe erreicht, muss der Benutzer darüber in Kenntnis gesetzt werden. Als Erinnerungsdatum werden nur Jahr, Monat und Tag betrachtet.

Liste anlegen:

Eine neue Liste anzulegen heißt ihr einen eindeutigen Namen zu geben. Nach dem Anlegen der Liste ist diese zunächst leer. Der Benutzer hat jetzt die Möglichkeit zu dieser Liste Aufgaben hinzuzufügen.

Aufgabe verschieben/löschen/bearbeiten:

Eine Aufgabe zu verschieben heißt, sie von einer Liste in eine andere zu bewegen. Dafür muss die entsprechende Aufgabe aus der Aufgabenliste, in der sie sich momentan befindet, gelöscht und anschließend einer anderen Liste hinzugefügt zu werden.
Eine Aufgabe zu löschen heißt alle mit ihr assoziierten Daten aus der Applikation zu entfernen, sodass diese anschließend nicht mehr abrufbar sind.
Eine Aufgabe zu bearbeiten, heißt eines der oben genannten Bestandteile einer Liste zu verändern oder die Aufgabe als erledigt bzw. unerledigt zu markieren. Wird eine Aufgabe als erledigt markiert wird sie aus ihrer aktuellen Liste entfernt und einer nicht vom Benutzer verwalteten Liste von erledigten Aufgaben zugeteilt, damit diese später noch abgerufen werden können. Eine Aufgabe als erledigt zu markieren ist also nicht dasselbe wie sie zu löschen! Wird eine Aufgabe wieder als unerledigt markiert wird ihr Status entsprechend verändert und sie wird der ursprünglichen Liste hinzugefügt aus der sie gekommen ist.

Liste löschen/bearbeiten:

Wird eine Liste gelöscht, dann löschen sich auch alle in ihr enthaltenen Aufgaben. Eine Liste zu bearbeiten bedeutet ihren Namen zu verändern. Dabei muss erneut geprüft werden, dass der Name eindeutig ist.

Analyse und Begründung der verwendeten Muster

Value Objects

Value Objects (VOs) kapseln einen oder mehrere Werte in einem neuen Wert/Typ. Das Value Object kann daraufhin die Einhaltung von bestimmten Regeln, die in der Problemdomäne gegenüber dem gekapselten Wert bestehen, prüfen. Ein VO muss unveränderlich sein. Das hat den Vorteil, dass der Zustand nach der Erzeugung eines VOs nicht mehr ungültig gemacht werden kann. Zwei VOs sind gleich, wenn ihre gekapselten Werte übereinstimmen.

Die Klasse *dev.fg.dhbw.ase.tasktracker.domain.vo.DateInFuture* ist ein solches Value Object. Es dient der Kapselung eines normalen Werts vom Typ *java.util.Date* um zu kontrollieren, dass dieses nicht in der Vergangenheit liegt. Nach der Instanziierung lässt sich der gekapselte Wert nicht mehr ändern (immutability). Die beiden Methoden *hashCode()* und *equals()* wurden überschrieben, um die Gleichheit zweier Objekte vom Typ *DueDate* bei Gleicheit des gekapselten Datums zu gewährleisten.

Weitere Value Objects sind:

- dev.fg.dhbw.ase.tasktracker.domain.vo.Title (der Titel darf nicht *null* oder leer sein)

- dev.fg.dhbw.ase.tasktracker.domain.vo.EMail (kapselt einen String um zu überprüfen, ob dieser dem Format einer E-Mail entspricht)
- dev.fg.dhbw.ase.tasktracker.domain.vo.Password (prüft die Anforderungen an das Passwort in der Anwendung)

(der obige Abschnitt bezieht sich auf das noch nicht modular aufgebaute Projekt)

Entities

Entitäten zeichnen sich durch eine eindeutige Identität und einen eigenen Lebenszyklus aus. Entitäten aggregieren Werte (auch VOs) zu einem Ganzen und repräsentieren die, für die Domäne relevanten, Daten/"Dinge", mit denen die Applikation arbeitet. Entitäten werden in der Persistenzschicht persistiert.

Entitäten im vorliegenden Projekt sind:

- dev.fg.dhbw.ase.tasktracker.domain.entities.Task
- dev.fg.dhbw.ase.tasktracker.domain.entities.TaskList
- dev.fg.dhbw.ase.tasktracker.domain.entities.User

Beispielsweise ist die Klasse *dev.fg.dhbw.ase.tasktracker.domain.entities.Task* eine Entität, da sie das Kernstück der Daten darstellt mit dem ein Aufgabenplaner/eine Aufgabenliste offensichtlicher Weise arbeitet. Außerdem ist sie eindeutig durch eine ID identifizierbar und muss, um für den Anwender und die Anwendung von Nutzen zu sein, persistiert werden. Sie aggregiert dafür mehrere Daten darunter auch die VOs *dev.fg.dhbw.ase.tasktracker.domain.vo.Title* und *dev.fg.dhbw.ase.tasktracker.domain.vo.DateInFuture*. Die Daten, die von der Entität aggregiert werden, können des Weiteren vom Anwender geändert werden d.h., es existiert ein eigener Lebenszyklus.

(der obige Abschnitt bezieht sich auf das noch nicht modular aufgebaute Projekt)

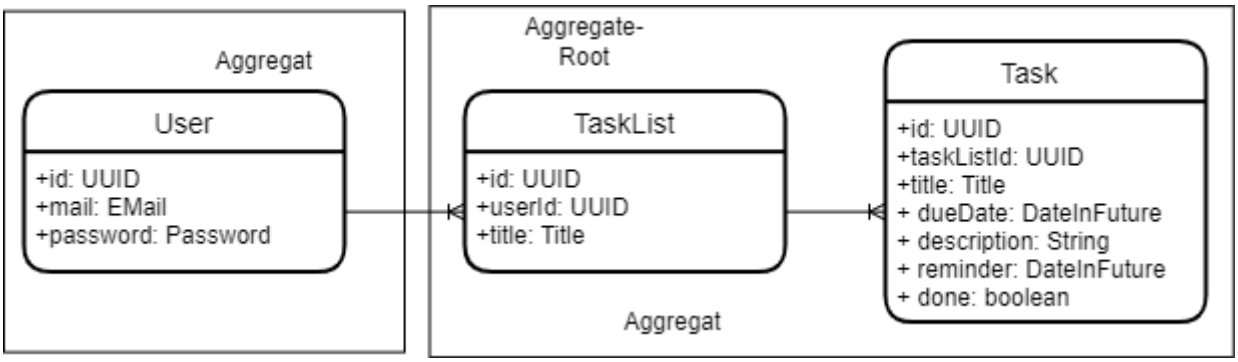
Domain Services

Ein Domain Service ist eine Art Hilfsklasse, die komplexe Regeln innerhalb des Domänenmodells implementiert. Die Einhaltung dieser Regeln ist keinem VO oder Entity eindeutig zuzuordnen. Dies ist bspw. der Fall, wenn mehrere Entitäten und VOs zur Überprüfung benötigt werden oder externe Dienste herangezogen werden. Um diese versehentliche Komplexität aus der Domäne herauszuhalten werden solche Prüfungen in Domain Services ausgelagert.

Aggregates

Aggregate sind Zusammenfassungen von Entitäten oder VOs, die miteinander in Beziehung stehen (1:n, n:m oder 1:1 usw.). Dabei hat jedes Aggregat eine sog. *Aggregate Root*, über die auf Elemente des Aggregats zugegriffen werden kann. Aggregate dienen dazu Objektbeziehungen zu entkoppeln und Domänenregeln einzuhalten, indem die *Aggregate Root* den direkten Zugriff auf, in der Beziehungshierarchie weiter unten liegende Objekte, verhindert und die Änderung von Daten kontrollieren kann.

Als Aggregate wurden gewählt das *User-Aggregat* (enthält nur die Entität *User*) und das *Task-List-Aggregat* (enthält die Entitäten *TaskList* und *Task*).



Repositories

Ein Repository ist die Schnittstelle zwischen der Domäne und der Persistenzschicht. Für jedes *Aggregate Root* (s.o.) wird ein Repository benötigt. In diesem Fall sind dies das *dev.fg.dhbw.ase.tasktracker.persistence.TaskListRepository* und das *dev.fg.dhbw.ase.tasktracker.persistence.UserRepository*. Ein Repository ist zunächst nur eine Schnittstelle (interface) die aber eine Vielzahl von Ausprägungen haben kann. Wichtig ist, dass diese konkreten Ausprägungen nicht mit der Domäne vermischt werden, da diese sich nicht für die konkrete Implementierung der Persistenz interessiert. Der Domäne wird nur das Interface präsentiert. Dies hat den Vorteil, dass im Hintergrund beliebige Implementierungen genutzt und diese auch reibungslos ausgetauscht werden können. Folgende Implementierungen sind in dieser Applikation vorhanden:

- *TaskListDatabaseRepository*: Verwendet eine Datenbankverbindung (*MySQL*) und den *Object Relational Mapper* Hibernate um Aufgabenlisten und Aufgaben zu persistieren (Laden, Ändern, Löschen).
- *TaskListFileSystemRepository*: Stellt eine Möglichkeit bereit Aufgaben und Aufgabenlisten in dem Dateisystem des Nutzers zu persistieren, falls dieser kein Benutzerkonto anlegen möchte.
- *UserDatabaseRepository*: Wie *TaskListDatabaseRepository* aber zur Verwaltung von Nutzern in der Datenbank.

Um die *accidental complexity* (Peristierung) von der *essential complexity* zu trennen wird nur das Repository-Interface nach außen gegeben und alle Klassen die mit der Persistenzschicht zu tun haben sind in einem separaten Package abgelegt.

(der obige Abschnitt bezieht sich auf das noch nicht modular aufgebaute Projekt)

Factories

Factories dienen der Erzeugung von Objekten unter Einhaltung domänenspezifischer Reglementierungen. Sie sind v.a. dann nützlich, wenn diese Reglementierungen und damit die Erzeugung des Objekts sehr komplex sind. Dabei kann eine Factory eine dedizierte Klasse zur Erzeugung anderer Objekte oder eine simple Methode sein. Ein Beispiel für eine Factory ist die Klasse *dev.fg.dhbw.ase.tasktracker.domain.factories.TaskFactory*. Diese stellt einige statische Methoden zur vereinfachten Erzeugung von Objekten des Typs *Task* bereit. Der Standardkonstruktor wird versteckt damit keine Objekte vom Typ *TaskFactory* erzeugt werden können.

(der obige Abschnitt bezieht sich auf das noch nicht modular aufgebaute Projekt)

Clean Architecture

Das Ziel von *Clean Architecture* ist es, Software so zu gestalten, dass sie über einen langen Zeitraum bestehen bleiben kann. Dies wird unter anderem dadurch erreicht, dass eine klare Trennung zwischen technologieabhängigen und technologieunabhängigen Komponenten stattfindet. Die Software muss so aufgebaut sein, dass die zugrundeliegenden Technologien ausgetauscht werden können ohne, dass der *langlebige Kern* der Anwendung verändert werden muss.

Schichtarchitektur planen und begründen

In der *Clean Architecture* wird eine Software in mehrere Schichten eingeteilt. Dabei ist es wichtig darauf zu achten, dass Abhängigkeiten zwischen den Schichten immer nur von außen nach innen gehen (innere Schichten wissen nichts von den Äußeren).

In dem vorliegenden Projekt lassen sich theoretisch 5 Schichten unterscheiden. Diese sind (von außen nach innen):

1. Plugins: In der Plugin-Schicht liegt das *User Interface* und sonstige dafür benötigte Ressourcen. Außerdem befindet sich sonstiger Code dort, der direkte Abhängigkeiten zu externen Bibliotheken oder Frameworks besitzt.
2. Adapters: Die Adapter-Schicht enthält zu großen Teilen *Mapping-Code* der die Daten aus der Plugin-Schicht in ein Format umwandelt, welches von der Applikationsschicht verstanden wird und umgekehrt. Diese Schicht kann oft (zumindest am Anfang) weggelassen werden, da sich die Datenmodelle nicht groß unterscheiden.
3. Application Code: Diese Schicht enthält Code der im direkten Zusammenhang zu den Anwendungsfällen und Anforderungen der vorliegenden Applikation steht. Er fasst Elemente aus dem Domänen-Code zusammen und verwendet diese zur Umsetzung der Anwendungsfälle (benutzt die Entitäten und VOs). Änderungen an dieser Schicht dürfen nicht den Domänencode beeinflussen und Änderungen an der Datenbank oder der graphischen Benutzeroberfläche nicht den *Application Code*. Den *Use Case* des *Application Code* darf nicht interessieren wer ihn aufgerufen hat und wie das Ergebnis das er zurückliefert präsentiert wird. Solche *Use Cases* sind im folgenden Projekt bspw.: Aufgaben einer Liste abrufen, eine neue Aufgabe erstellen, eine neue Liste erstellen, eine Aufgabe löschen, eine Liste löschen, Statistiken abrufen...
4. Domain Code: Wie der Name bereits andeutet enthält diese Schicht den Code, der direkt mit der Problemdomäne im Zusammenhang steht. Wie bereits in dem Abschnitt über das *Domain Driven Design* festgestellt wurde sind dies im wesentlichen die Entitäten und *Value Objects*, die bereits in den Abschnitten *Entities* und *Value Objects* unter der Überschrift *Analyse und Begründung der verwendeten Muster* herausgearbeitet wurden.
5. Abstraction Code: Diese Schicht enthält domänenübergreifendes Wissen und Funktionalitäten die in vielen oder allen anderen Schichten benötigt werden. Sie abstrahiert bspw. vom Code (bzw. den Abhängigkeiten auf Code) der verwendeten Bibliotheken wie bspw. der verwendeten Bibliothek zur Umsetzung der Persistenz. Für das vorliegende Projekt wäre eine geeignete Abhängigkeit für diese Schicht das JUnit Test-Framework. Oft kann die Schicht aber auch eingespart werden.

Zunächst wurde das *TaskTracker*-Projekt ohne Bachtung der Regeln der *Clean Architecture* entwickelt. Das alte Projekt liegt im Ordner *./tasktracker/* vor. Das neue modularisierte Projekt befindet sich im Ordner *./tasktracker-modules/*. Die Schichten wurden als einzelne Maven-Module umgesetzt. Das Elternprojekt *tasktracker-parent* enthält die folgenden Module:

- *tasktracker-abstraction*: Repräsentiert die Abstraktionsschicht und definiert Abhängigkeiten zu der *javax.persistence* API sowie dem *JUnit*-Test-Framework. Diese Abhängigkeiten befinden sich in der Abstraktionsschicht, da sie öfters im Projekt verwendet werden müssen und über einen langen Zeitraum bestehen bleiben werden. Außerdem befinden sich in der Schicht die Klassen zur Umsetzung des Beobachtermusters. Hier gilt dasselbe: der Code wird sich nur sehr selten ändern und sollte aber innerhalb des gesamten Projektes nutzbar sein.
- *tasktracker-domain*: In der Domänenschicht befinden sich die Entitäten und VOs, da diese aus der Domäne und während des DDD-Prozesses entstanden sind und die dort geltenden Regeln einhalten müssen. Ebenfalls sind dort die Repositories für die vorhandenen Aggregate untergebracht, da die Verwaltung der Entitäten ebenfalls zur Problemdomäne gehört.
- *tasktracker-application*: Die Applikationsschicht enthält Klassen und Methoden zur Umsetzung der Use-Cases. Sie verwendet dafür die Repositories, Entitäten und VOs aus der darunterliegenden Schicht. In diesem Projekt wurden Use-Cases unterschieden die Aufgabenlisten, Aufgaben oder den Benutzer betreffen und diese jeweils in einer Klasse zusammengefasst. Die von den Klassen beereitgestellte Methoden werden von der Plugin-/Adapter-Schicht verwendet.
- *tasktracker-adapters*: Die Adapterschicht wurde in diesem Projekt ausgespart. Hier würde der nötige Mapping-Code liegen, um vom Datenmodell der Plugin-Schicht auf das Datenmodell der Applikationsschicht abzubilden.
- *tasktracker-plugins*: Die Plugin-Schicht ist die am Weitesten außen liegende Schicht deren Code sich am häufigsten ändert, da er am konkretesten ist und direkte Abhängigkeiten zu externen Bibliotheken und Frameworks besitzt. Hier liegen alle UI-Elemente und die dafür benötigten Ressourcen (in diesem Fall abhängig von JavaFX). Außerdem werden die Repository-Interfaces durch konkrete Implementierungen ersetzt.

Programming Principles

Analyse und Begründung für SOLID

SOLID ist ein Akronym welches mehrere Programmierparadigmen unter sich vereint, die im Folgenden vorgstellt und analysiert werden sollen.

Single Responsibility Principle (SRP)

Das SRP besagt, dass eine Softwarekomponente nur eine einzige bestimmte Aufgabe haben sollte für die sie zuständig ist. Das SRP lässt sich auf mehreren Ebenen anwenden:

- Module
- Klassen
- Methoden
- Variablen

Ein Beispiel für den Einsatz von SRP stellt die Methode `initialize()` dar, die sich in der Klasse `dev.fg.dhbw.ase.tasktracker.domain.controller.ListViewController` befindet. Die Methode wird von dem JavaFX-Framework automatisch aufgerufen, sobald alle mit der Annotation `@FXML` markierten Felder *befüllt* wurden und erfüllt damit eine ähnliche Funktion wie der Konstruktor. Die Befehle die innerhalb der Methode ausgeführt werden lassen sich in drei Aufgaben unterteilen:

1. Vorbereitung der UI

2. Erstellen der, von der Applikation verwalteten, Liste zur Speicherung abgeschlossener Aufgaben, falls diese nicht vorhanden ist
3. Abrufen aller vorhandenen Aufgabenlisten für den angemeldeten Benutzer und deren Darstellung in der UI

Das SRP fordert, dass diese Aufgaben in einzelne Methoden ausgelagert werden (oder falls dies angebracht ist auch in separate Klassen). Deshalb wurde für die Vorbereitung der UI die Methode `prepareUI` eingeführt. Anschließend wird die Methode `List<TaskList> getTaskListsForUser()` aufgerufen, die die Liste für die abgeschlossenen Aufgaben automatisch hinzufügt falls sie noch nicht vorhanden ist. Der Rückgabewert der Methode wird anschließend verwendet um die UI zu aktualisieren. (vgl. ListViewController)

Open/Closed Principle (OCP)

Das OCP besagt, dass Module offen für Erweiterungen aber geschlossen für Änderungen sein sollten. Mit anderen Worten sollte eine Software so designt sein, dass sich leicht neue Features hinzufügen lassen ohne große Änderungen an (öffentlichen) Schnittstellen vornehmen zu müssen, da dies zu weiteren Änderungen an allen möglichen Stellen in der Software führt.

Das OCP lässt sich durch die Anwendung von Interfaces umsetzen, da hierbei direkte Abhängigkeiten vermieden werden, die im Normalfall Erweiterungen der Anwendung erschweren. Dies wurde beispielsweise bei der Implementierung der Persistenzschicht beachtet. Hierfür wurden Repository-Interfaces in der Domänenschicht definiert die in der Plugin-Schicht durch konkrete Implementierungen ersetzt werden. Hierdurch lassen sich beliebig neue Implementierungen hinzufügen ohne, dass die Domäne dadurch beeinflusst wird (Offenheit gegenüber Erweiterung).

Liskov Substitution Principle (LSP)

Das LSP besagt, dass eine Klasse an jeder beliebigen Stelle durch ihre Kindklassen ersetzt werden können muss ohne, dass es zu unerwünschten Nebeneffekten kommt.

In diesem Projekt kann das LSP anhand der Klasse `dev.fg.dhbw.ase.tasktracker.abstraction.observer.Observable` und ihrer Kindklassen nachvollzogen werden. Man stelle sich eine Methode vor, die ein Objekt diesen Typs entgegennimmt und seine Methoden verwendet. Sollte zur Laufzeit das Objekt durch einen Subtyp von Observable ausgetauscht werden würde es zu keinen unerwünschten Nebeneffekten kommen womit das LSP erfüllt ist.

Interface Segregation Principle (ISP)

Das ISP besagt, dass es besser ist viele client-spezifische Interfaces zu haben als ein allgemeines, welches alle Clientanforderungen in sich vereint. Dadurch lässt sich die Anwendung unter Umständen leichter erweitern und es wird verhindert, dass viele leere Implementierungen bei Clients entstehen, die nur einen Bruchteil der Funktionalität des allgemeinen Interfaces brauchen.

Interfaces kommen in dem vorliegenden Projekt zur Umsetzung der Datenpersistierung zum Einsatz, indem für jedes Aggregat ein Repository in der Domäne definiert wird, welches die Schnittstelle definiert, die von den darüberliegenden Schichten genutzt werden kann. Außerdem kommt ein Observer-Interface in der Abstraktionsschicht zum Einsatz. Die Interfaces sind so abgestimmt, dass keine leeren Implementierungen in irgendeiner Klasse dieses Projekts entstehen.

Dependency Inversion Principle (DIP)

Module auf höherer Ebene sollten keine Abhängigkeiten zu Modulen auf niedrigerer Ebene aufweisen. Stattdessen sollte eine Abstraktion dazwischen geschaltet werden von denen beide Ebenen abhängen. Abstraktionen sollten wiederum unabhängig von Details sein. Stattdessen sollten die Details von der Abstaktion abhängen.

Dieses Prinzip wurde durch die Einhaltung geltender Regeln der *Clean Architecture* umgesetzt. Die Anwendung ist in fünf Schichten eingeteilt. Die Abhängigkeiten der einzelnen Schichten gehen ausschließlich von den äußeren (niedrigere Ebene) zu den inneren (höhere Ebene) Schichten.

Analyse und Begründung für GRASP

GRASP ist ein Akronym und steht für *General Responsibility Assignment Software Patterns*. GRASP vereint eine Reihe an Prinzipien und Mustern die im Folgenden näher erläutert werden sollen.

Low Coupling (lose Kopplung)

Unter dem Begriff der Kopplung versteht man den Grad der Abhängigkeit von zwei Softwarekomponenten. Ziel bei der Entwicklung von Software ist meist eine lose Kopplung d.h., dass die Abhängigkeiten zwischen den Komponenten auf ein Minimum reduziert werden sollen.

Low Coupling kann bspw. durch den Einsatz von Interfaces erreicht werden, da diese von der eigentlichen Implementierung abstrahieren und die Komponenten, welche die definierte Schnittstelle nutzen, von der eigentlichen Implementierung entkoppelt werden (lose Kopplung). Dies ist bspw. durch den Einsatz von *Repositories* an diesem Projekt zu sehen (s. `dev.fg.dhbw.ase.tasktracker.persistence`).

(der obige Abschnitt bezieht sich auf das noch nicht modular aufgebaute Projekt)

High Cohesion (hohe Kohäsion)

Unter Kohäsion versteht man den Grad von Zusammengehörigkeit innerhalb einer Softwarekomponente. Es sollte versucht werden eine möglichst hohe Kohäsion zu erreichen d.h., dass bspw. Variablen und Methoden einer Klasse inhaltlich zusammenpassen. Innerhalb einer Softwarekomponente versucht man also den Zusammenhalt der Bestandteile hoch zu halten wohingegen die Softwarekomponenten untereinander möglichst schwach aneinander gebunden sein sollten.

In dem vorliegenden Projekt wurde hohe Kohäsion bspw. dadurch gefördert, dass die einzelnen UI-Komponenten weitestgehend in separate Klassen ausgelagert wurden (s. das Paket `dev.fg.dhbw.ase.tasktracker.domain.components`). Die Idee hierbei ist, dass jede UI-Komponente sich selbst verwaltet und nur mit den Daten arbeitet, die sie wirklich benötigt. Zusammengehörige Variablen und Methoden werden somit in separate Klassen abgekapselt was zu einer erhöhten Kohäsion führt.

(der obige Abschnitt bezieht sich auf das noch nicht modular aufgebaute Projekt)

Information Expert (Informationsexperte)

Hinter dem Experten-Prinzip verbirgt sich der Gedanke, dass eine neue Aufgabe von derjenigen Softwarekomponente übernommen werden sollte, die bereits das meiste *Wissen* zur Erfüllung der Aufgabe besitzt. Dadurch werden beispielsweise unnötige *Hilfsklassen* verhindert.

Das Prinzip des Informationsexperten wurde bspw. bei der Methode `formatDate(DateInFuture): String` eingesetzt die vormals in der Klasse `dev.fg.dhbw.ase.tasktracker.domain.components.TaskComponent` vorlag. Anstatt eine *unnötige* Hilfsklasse wie bspw. `DateUtils` o.ä. einzuführen wurde versucht die Methode in diejenige Klasse zu verlagern, die zu der vorliegenden Thematik/dem vorliegenden Domänenproblem passt. Deshalb wurde die Methode zu einem späteren Zeitpunkt in die `DateInFuture` Klasse selbst ausgelagert. (vgl. vorher, nachher)

Polymorphism (Polymorphie)

Unter dem Begriff der Polymorphie versteht man im Zusammenhang mit GRASP das Prinzip unterschiedliches Verhalten eines Typs durch Polymorphie auszudrücken.

In diesem Projekt kommt Polymorphie bspw. zur Unterscheidung von offenen und abgeschlossenen Aufgaben zum Einsatz (s. `dev.fg.dhbw.ase.tasktracker.domain.components.FinishedTaskComponent` und `dev.fg.dhbw.ase.tasktracker.domain.components.OpenTaskComponent`).

(der obige Abschnitt bezieht sich auf das noch nicht modular aufgebaute Projekt)

Pure Fabrication (reine Erfindung)

Unter der *Pure Fabrication* versteht man Klassen oder Module die in der Problemdomäne nicht existieren (deshalb *reine Erfindung*). Sie implementieren Methoden für die sie nicht Experte sind. Dies wird in diesem Fall toleriert, da somit eine Trennung zwischen Technologiewissen und Domänenwissen stattfinden kann. Allerdings sollte mit *reinen Erfindungen* sehr sparsam umgegangen werden (nur wenn es wirklich notwendig ist).

Ein Beispiel für eine reine Erfindung ist die Klasse `dev.fg.dhbw.ase.tasktracker.persistence.TaskListDatabaseRepository`. Es handelt sich dabei um *Pure Fabrication*, da die Klasse kein Objekt aus der realen Welt bzw. der Problemdomäne darstellt.

(der obige Abschnitt bezieht sich auf das noch nicht modular aufgebaute Projekt)

Indirection/Delegation (Indirektion/Delegation)

Mit Indirektion/Delegation ist gemeint, dass ein Objekt Aufgaben an ein weiteres Objekt delegiert. Das Objekt zu dem deligiert wird ist meist besser zur Erfüllung der Aufgabe geeignet (kann bspw. Experte sein). Das Prinzip kann Vererbung ersetzen da die Funktionsweise analog ist zu einer Kindklasse die Methodenaufrufe zu ihrer Elternklasse delegiert. Außerdem kann dadurch eine hohe Kohäsion gefördert werden, da inhaltlich zusammengehörige Bestandteile in Klassen ausgelagert und die Kommunikation über Delegation geregelt werden kann.

Beispielsweise delegieren die Controller aus der Plugin-Schicht des modularisierten Projekts Aufgaben an die Services aus der darunterliegenden Applikationsschicht zur Erfüllung der Use-Cases.

Protected Variations (geschützte Veränderungen)

Das Prinzip der geschützten Veränderungen besagt, dass konkrete Implementierungen mit Hilfe von Interfaces *versteckt* werden sollten. Die Softwarekomponenten greifen nur auf das Interface zu. Dadurch

können im Hintergund die Implementierungen ausgetauscht oder angepasst werden ohne, dass dies Auswirkungen auf das restliche System hat. Das System ist also *geschützt vor Veränderungen*.

In dem Package *dev.fg.dhbw.ase.tasktracker.persistence* befinden sich die Klassen zur Persistierung und Bereitstellung von Daten mit denen die Anwendung arbeitet. Für jede Aggregate-Root existiert ein Repository welches die Schnittstellen für den domänenspezifischen Teil der Anwendung definiert. Innerhalb diesen Teils der Anwendung wird nur mit dem Interface (bzw. dem Repository) gearbeitet, sodass die Implementierungsdetails der Persistenzschicht für die Domäne unsichtbar bleiben und beliebig ausgetauscht werden können. Dadurch bleibt die Domäne nicht nur von *Accidental Complexity* befreit sondern es wird auch das *Protected Variations Pattern* unterstützt, da durch die Abstraktion der Implementierung mit Hilfe von Interfaces die Domäne vor Veränderungen geschützt bleibt.

(der obige Abschnitt bezieht sich auf das noch nicht modular aufgebaute Projekt)

Controller (Steuereinheit)

Ein Controller stell die erste Instanz nach dem GUI dar. Er nimmt Events aus der Benutzerschnittstelle entgegen und delegiert diese an andere Klassen, welche die Events verarbeiten können. Der Controller beinhaltet das Domänenwissen und sollte möglichst wenig selbst tun und stattdessen Aufgaben delegieren.

Das in diesem Projekt eingesetzte UI-Framework *JavaFX* unterstützt die Verwendung von Controllern standardmäßig. Dabei kann eine beliebige Java-Klasse als Controller auf einem Root-Element der UI gesetzt werden. Die UI kann vollständig vom Code entkoppelt und im XML-Format in sog. FXML-Dateien ausgelagert werden (s. */main/resources/fxml/*). Beispiele für Controller-Klassen sind im Package *dev.fg.dhbw.ase.tasktracker.domain.controller* zu finden. Beispielsweise wird der *ListViewController* als Controller-Klasse auf dem Root-Element der *ListView.fxml* gesetzt. Durch Annotationen können Elemente aus FXML-Dateien im Code zugegriffen und Events verarbeitet werden. Dadurch sind alle Controller-Klassen die erste Instanz nach der UI und die Logik liegt im Controller und ist von der UI entkoppelt.

(der obige Abschnitt bezieht sich auf das noch nicht modular aufgebaute Projekt)

Creator (Erzeuger)

Das Erzeuger-Prinzip definiert Regeln die vorgeben wer für die Erzeugung von Instanzen zuständig sein darf. Klasse A darf eine Instanz von Klasse B erzeugen, wenn:

- A eine Aggregation von B ist oder Objekte von B enthält
- A Objekte von B verarbeitet
- A von B abhängt (starke Kopplung)
- A der Informationsträger/-experte für die Erzeugung von B ist (bspw. Factory)

Beispielsweise erzeugt in diesem Projekt die Klasse `dev.fg.dhbw.ase.tasktracker.domain.controller.ListViewController` Objekte vom Typ `...domain.components.TaskListComponent`. Dies ist nach dem Erzeugerprinzip damit zu rechtfertigen, da die Klasse `ListViewController` eine klare semantische Verbindung zu `TaskListComponent` hat und Objekte diesen Typs verarbeiten muss.

(der obige Abschnitt bezieht sich auf das noch nicht modular aufgebaute Projekt)

Analyse und Begründung für DRY

DRY ist ein Akronym und steht für *Don't Repeat Yourself* (z. Dt. wiederhole dich nicht). Die Idee ist hierbei, dass Redundanzen im Code vermieden werden sollten, d.h. bspw. die selbe Logik nicht mehr als einmal im Code implementiert wird.

DRY wurde in diesem Projekt beispielsweise bei der Umstellung des Observable-Interfaces auf eine Klasse eingesetzt. Durch die Umsetzung von Observable als Interface musste jede Implementierung alle Methoden implementieren obwohl deren Implementierung eigentlich immer gleich ist (Observer der Liste hinzufügen, Observer aus der Liste entfernen usw.). Durch die Umsetzung als vollwertige Klasse ist dies nicht mehr der Fall und es können Wiederholungen aus dem Code entfernt werden. (vgl. Beispielimplementierung vorher und Umsetzung als Klasse)

Refactoring

Code Smells identifizieren

Code Smell 1: Large Class

Dieser Code Smell gehört zur Klasse der *Bloaters*. Diese Klasse fasst Code Smells zusammen, die dafür sorgen, dass der Code an manchen Stellen gigantische Ausmaße annimmt und dadurch schwer zu Pflegen ist. Der Smell *Large Class* bezieht sich dabei speziell auf besonders große Klassen. Ein Beispiel ist die Klasse `dev.fg.dhbw.ase.tasktracker.domain.controller.ListViewController`, die zum aktuellen Zeitpunkt fast 300 Zeilen umfasst. Die Größe wird sich wahrscheinlich noch weiter verschlimmern, da der `ListViewController` momentan die Hauptaufgaben der Anwendung erfüllt (Listen und Aufgaben anzeigen, erstellen, löschen, ...) und voraussichtlich immer mehr Features hinzugefügt werden.

Der Code Smell kann gelöst werden indem die Refactorings *Extract Class*, *Extract Subclass* oder *Extract Interface* angewendet werden. Dabei wird Funktionalität aus einer großen Klasse in andere Klassen ausgelagert. (vgl. ListViewController)

Code Smell 2: Switch Statements

Dieser Code Smell gehört zur Klasse der *Object-Orientation Abusers*. Diese Klasse fasst Code Smells zusammen, welche eine unvollständige oder falsche Anwendung von objektorientierten Programmierprinzipien darstellen. Der Smell *Switch Statements* bezieht sich auf das Vorkommen komplexer `switch`-Statements sowie Abfolgen von `if`-Statements. Ein Beispiel für eine solche Abfolge findet sich in der Klasse `dev.fg.dhbw.ase.tasktracker.domain.controller.ListViewController`. Die Methode `notifyObserver(Object)` führt eine Reihe von Prüfungen durch, die die Art des Events feststellen sollen. Dies führt zu einer überaus langen und unübersichtlichen Methode, die mit der Zeit immer weiter anwächst, falls neue Events dazu kommen. Kommt ein neues Event hinzu muss außerdem die Methode um ein weiteres `if`-Statement erweitert werden. Der Code ist also gegenüber Erweiterungen sehr unflexibel.

Eine Mögliche Lösung für das Problem ist der Einsatz von Polymorphie. Durch die Einführung einer neuen Klasse für jedes Event und einem gemeinsamen Interface könnte der gesamte Körper der Methode durch einen einzigen Methodenaufruf ersetzt werden. Dafür würde jede Event-Klasse das Event-Interface implementieren. Dieses würde dann stellvertretend für die einzelnen Events als Übergabeparameter verwendet werden (`notifyObserver(IEvent)`). Anschließend kann auf dem Interface die Methode aufgerufen werden, die von jeder Event-Klasse implementiert wird. (vgl. switch-statements, oo-abusers und ListViewController)

Code Smell 3: Long Method

Dieser Code Smell gehört ebenfalls zur Klasse der *Bloaters* und bezieht sich auf besonders lange Methoden mit vielen Codezeilen. Als Daumenregel gilt, dass eine Methode nicht mehr als zehn Zeilen Code enthalten sollte. Eine sehr lange Methode legt nahe, dass diese Methode eine Reihe verschiedener Aufgaben übernimmt, die eigentlich in separate Methoden ausgelagert werden sollten. Beispielsweise wird auch das *Single Responsibility Principle* dadurch verletzt. Beispiele für lange Methoden finden sich aktuell in der Klasse `dev.fg.dhbw.ase.tasktracker.domain.controller.ListViewController`. Diese Klasse enthält leider viele lange und komplexe Methoden, die teilweise mehrere Aufgaben übernehmen. Als konkretes Beispiel kann wieder die Methode `notifyObserver(Object)` herangezogen werden, die außerdem Gefahr läuft immer weiter anzuwachsen. Diese hat 33 Zeilen also mehr als dreimal so viel wie im optimalen Fall.

Code Smells dieser Art lassen sich durch das extrahieren von Methoden lösen. Dabei werden die Codezeilen, die zusammen eine bestimmte Aufgabe erfüllen, in eine separate Methode ausgelagert. Dadurch entstehen viele kleine, leicht wartbare und verständliche Methoden und die Größe der ursprünglichen Methode wird deutlich verkleinert. (vgl. bloaters, long-method und ListViewController)

Code Smell 4: Verletzung des Informationsexperten-Prinzips

Neue Features sollten immer in der Klasse hinzugefügt werden, die bereits die meisten Informationen für dieses Feature beinhaltet. Dadurch wird dafür gesorgt, dass zusammengehörige Informationen zusammen bleiben und unnötige Hilfsklassen werden verhindert. Eine Verletzung dieses Prinzips (*Smell*) fand in der Methode `TaskComponent.formatDate(DateInFuture)` statt. Zwar wurde eine unnötige Hilfsklasse in diesem Fall ausgespart - dennoch ist das Formatieren eines Datums naheliegenderweise die Aufgabe des Datums selbst und nicht die einer bestimmten UI-Komponente. Durch das anschließende Verlagern wird nicht nur das Informationsexperten-Prinzip eingehalten sondern auch dafür gesorgt, dass die allgemeine und evtl. häufiger benötigte Aufgabe der Datumsformatierung nun für andere Komponenten die das `DateInFuture`-Objekt benutzen verwendbar ist. (vgl. vorher, nachher)

Begründung durchgeführter Refactorings

Refactoring 1: Extract Method

Das Refactoring *Extract Method* gehört zur Kategorie *Composing Methods*. Diese fasst Refactorings zusammen, die dafür sorgen den Einsatz von Methoden zu verbessern, indem die Methodenlänge reduziert und der Codeduplizierung entgegengewirkt wird. *Extract Method* bezieht sich im speziellen auf das Aufteilen einer Methode in viele kleine Methoden. Dabei werden Code-Fragmente die inhaltlich zusammengehören in eine separate Methode *extrahiert*. Dadurch wird der Code lesbarer und leichter verständlich, da die ursprüngliche Länge der Methode verringert wird und die neuen Methoden (zumindest sollte dies der Fall sein) mit sprechenden Namen versehen werden aus denen ihre Funktion hervorgeht.

Ein Beispiel für eine Methode bei der das Refactoring *Extract Method* eingesetzt werden kann ist `onAddTaskButtonClicked(Event)` in der Klasse `dev.fg.dhbw.ase.tasktracker.domain.controller.ListViewController`. Diese ist zum einen sehr lang und zum anderen erfüllt sie mehrere Aufgaben in einem, die stattdessen in separate Methoden ausgelagert werden könnten. Diese Aufgaben lassen sich wie folgt zusammenfassen:

1. Prüfe ob aktuell eine Liste ausgewählt ist und ob es sich dabei um eine intern verwaltete Liste handelt (d.h. um die Liste zur Speicherung abgeschlossener Aufgaben oder die Überschrift, die direkt nach dem Start der Applikation angezeigt wird)
2. Öffne das Fenster mit dem Formular zum Hinzufügen einer Aufgabe zu einer Liste

Die beiden Aufgaben wurden in die Methoden `selectedListNameIsInvalid()` und `openAddTaskWindow()` extrahiert. (vgl. composing-methods, extract-method, vorher und nachher)

Refactoring 2: Extract Class und Extract Superclass

Bei dem *Extract Class* Refactoring geht es um das Extrahieren von Funktionalität (Variablen und Methoden) einer Klasse in eine andere Klasse, d.h. aus einer Klasse werden zwei Klassen. Dies trägt zu einer erhöhten Modularität, Übersichtlichkeit und der Einhaltung des SRP bei. Das Refactoring kann dann sinnvoll eingesetzt werden, wenn eine Klasse mehrere Aufgaben übernimmt, die eigentlich besser von separaten Klassen übernommen werden könnten. Das Refactoring *Extract Superclass* trägt dazu bei Redundanzen aus dem Code zu entfernen indem zwei Klassen, die ähnliche Felder und Methoden benutzen durch eine gemeinsame Elternklasse ergänzt werden, die die Funktionalität, welche von den beiden Unterklassen genutzt wird zu implementieren.

In diesem Projekt wurde das oben beschriebene Refactoring bei der Klasse `dev.fg.dhbw.ase.tasktracker.domain.components.TaskComponent` eingesetzt. Zunächst erüllte die Klasse die Anforderungen von sowohl abgeschlossenen als auch offenen Aufgaben. Dies führte allerdings dazu, dass die Klasse an immer mehr Komplexität gewann und If-Statements verwendet werden mussten. Deshalb wurde die Klasse in zwei separate Klassen `FinishedTaskComponent` und `OpenTaskComponent` aufgeteilt. Die beiden entstandenen Klassen teilen einen Großteil der Funktionalität weshalb zusätzlich die gemeinsame Oberklasse `TaskComponent` eingeführt wurde (*Extract Superclass*). (vgl. vorher, nachher, extract-class und extract-superclass)

Entwurfsmuster

Observer

Das Observer-Pattern gehört zur Klasse der Benachrichtigungsmuster. Es ermöglicht den Austausch von Nachrichten zwischen Softwarekomponenten ohne, dass eine starke Kopplung zwischen den Komponenten entsteht. Beim Observer-Pattern existieren zwei Arten von Entitäten: Observer und Observables. Ein Observable ist eine Komponente, die von einem Observer *beobachtet* werden kann. Ein Observer ist eine Komponente, die sich bei einem Observable registrieren und anschließend Nachrichten von dem Observable erhalten kann (s. *dev.fg.dhbw.ase.tasktracker.domain.observer.Observer* und *dev.fg.dhbw.ase.tasktracker.domain.observer.Observable*).

Einsatz begründen

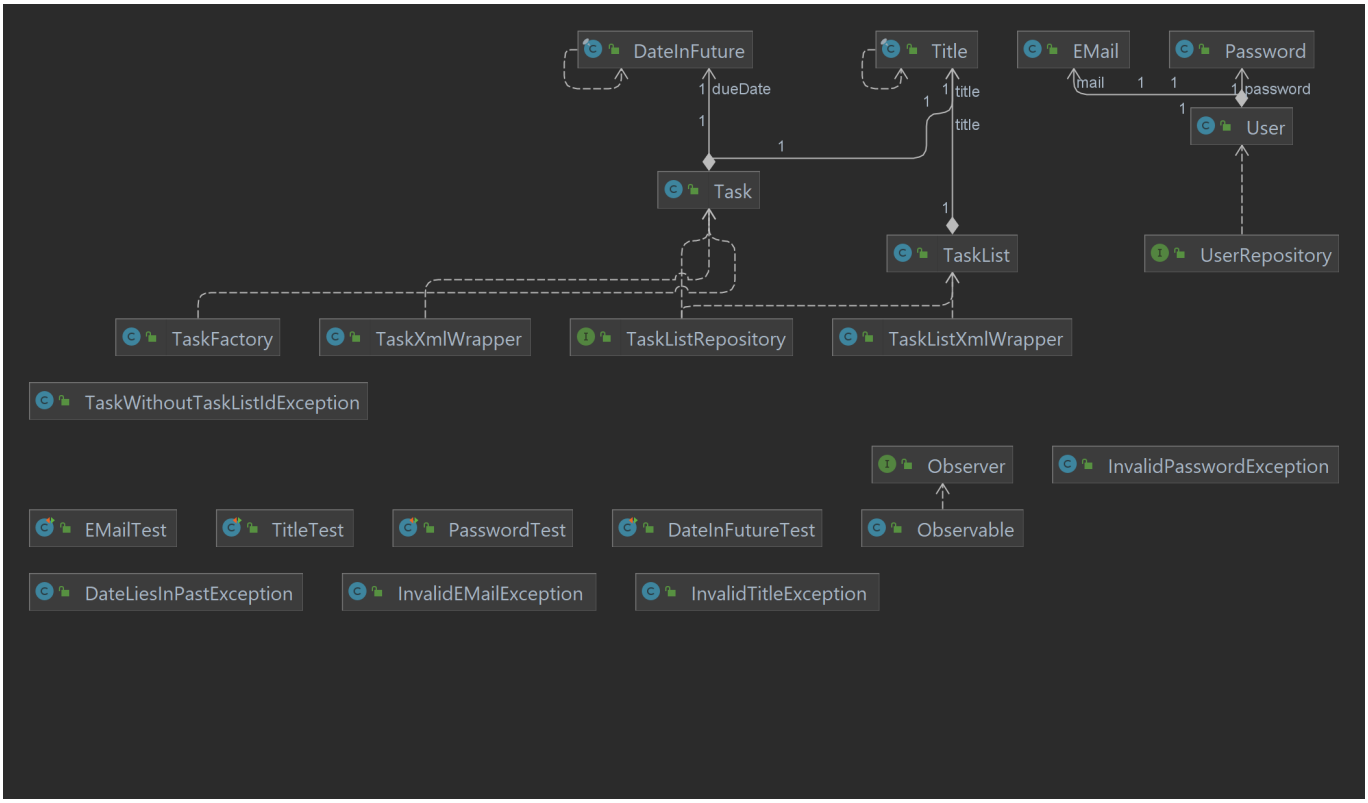
Das Observer-Pattern wird in diesem Projekt genutzt um einen Großteil der Kommunikation zwischen den einzelnen Komponenten des User Interfaces zu gestalten. So implementiert bspw. die Klasse *dev.fg.dhbw.ase.tasktracker.domain.components.TaskComponent* das Observable-Interface (später zu Klasse geändert) um registierte Observer bei dem Löschen einer Aufgabe oder dem Markieren einer Aufgabe als abgeschlossen zu benachrichtigen. Die Klasse repräsentiert die UI-Komponente einer Aufgabe innerhalb einer Liste. Die Klasse *dev.fg.dhbw.ase.tasktracker.domain.controller.ListViewController* implementiert das Observer-

Interface und registriert sich bei einem *TaskComponent*, um über diese Events benachrichtigt zu werden und das UI zu aktualisieren. Eine ähnliche Vorgehensweise wurde auch bei den anderen UI-Komponenten gewählt. Der Aufzählungstyp *dev.fg.dhbw.ase.tasktracker.domain.components.ComponentEvent* definiert zudem die Arten der Events die auftreten können. Die Observables definieren dort die Events die sie an die Observer verteilen. Dadurch kann der Observer auf die unterschiedlichen Events verschieden reagieren.

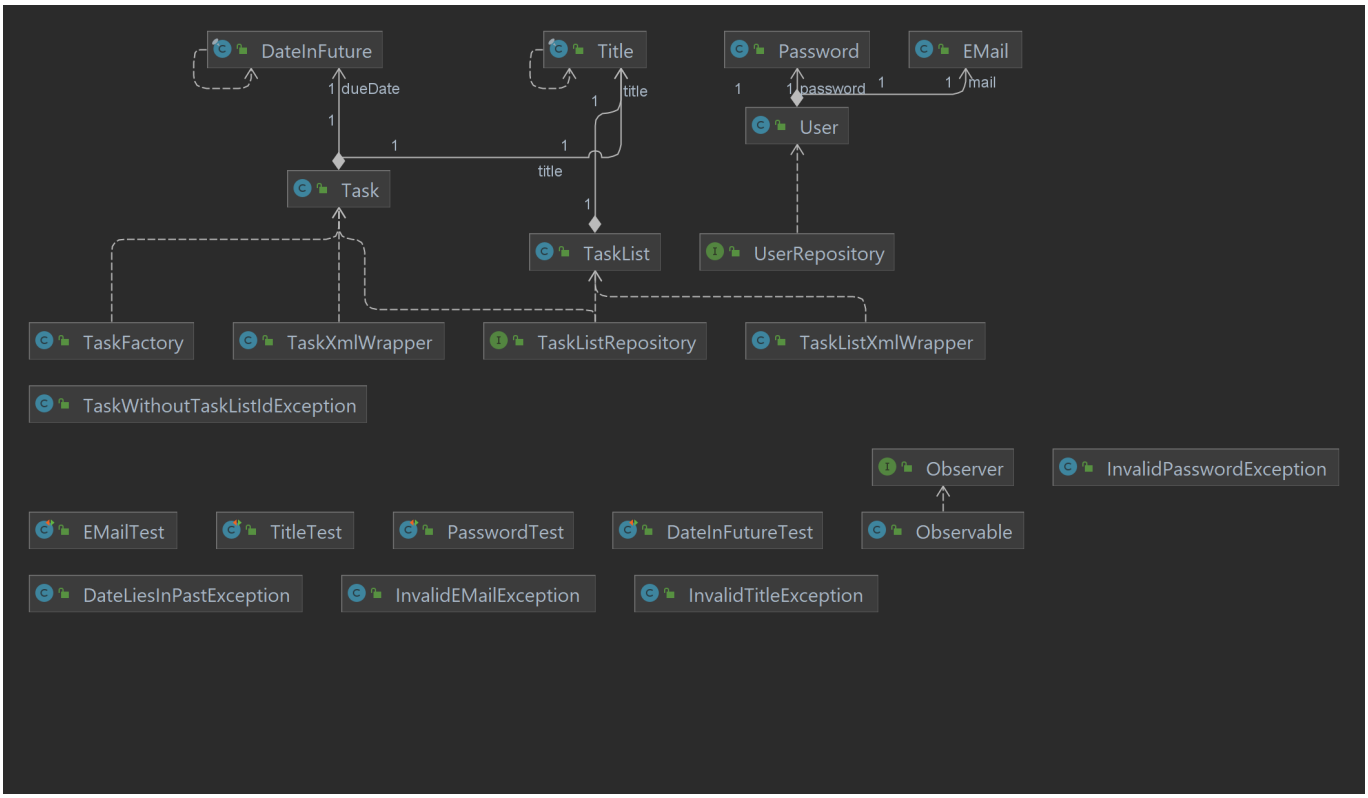
(der obige Abschnitt bezieht sich auf das noch nicht modular aufgebaute Projekt)

UML

Klassen der Abstraktions- und Domänenschicht



Klassen der Domänen- und Applikationsschicht



Klassen der Applikations- und Pluginschicht