Contents

Programmentwurf
[Bezeichung]
Name: [Name, Vorname]
Martrikelnummer: [MNR]
Abgabedatum: [DATUM]

1

Allgemeine Anmerkungen:

- es darf nicht auf andere Kapitel als Leistungsnachweis verwiesen werden (z.B. in der Form "XY wurde schon in Kapitel 2 behandelt, daher hier keine Ausführung")
- alles muss in UTF-8 codiert sein (Text und Code)
- sollten mündliche Aussagen den schriftlichen Aufgaben widersprechen, gelten die schriftlichen Aufgaben (ggf. an Anpassung der schriftlichen Aufgaben erinnern!)
- alles muss ins Repository (Code, Ausarbeitung und alles was damit zusammenhängt)
- die Beispiele sollten wenn möglich vom aktuellen Stand genommen werden
  - finden sich dort keine entsprechenden Beispiele, dürfen auch ältere Commits unter Verweis auf den Commit verwendet werden
  - Ausnahme: beim Kapitel "Refactoring" darf von vorne herein aus allen Ständen frei gewählt werden (mit Verweis auf den entsprechenden Commit)
- falls verlangte Negativ-Beispiele nicht vorhanden sind, müssen entsprechend mehr Positiv-Beispiele gebracht werden

  - Achtung: werden im Code entsprechende Negativ-Beispiele gefunden, gibt es keine Punkte für die zusätzlichen Positiv-Beispiele
  - Beispiele
    * "Nennen Sie jeweils eine Klasse, die das SRP einhält bzw. verletzt."
      · Antwort: Es gibt keine Klasse, die SRP verletzt, daher hier 2 Klassen, die SRP einhalten: [Klasse 1], [Klasse 2]
      · Bewertung: falls im Code tatsächlich keine Klasse das SRP verletzt: volle Punktzahl ODER falls im Code mind. eine Klasse SRP verletzt: halbe Punktzahl

- verlangte Positiv-Beispiele müssen gebracht werden
- Code-Beispiel = Code in das Dokument kopieren

2 Kapitel 1: Einführung

2.0.1 Übersicht über die Applikation

[Was macht die Applikation? Wie funktioniert sie? Welches Problem löst sie/welchen Zweck hat sie?]

Die Anwendung ist zur organisierten Abspeicherung von Links bzw. URLs gedacht. Diese können durch die Anwendung abgespeichert werden. Zur besseren Organisation ist es außerdem möglich, Kategorien anzulegen und die Links diesen zuzuordnen, wie beispielsweise 'Libary', 'Selfhostable', 'Dienst' usw. Zusätzlich kann die Anwendung auch Tags zu Links hinzufügen können, wenn die Implementation Webseite beispielsweise bereits kennt (z.B. 'Github'). Eigene Regeln für Tags können auch angelegt werden, sie werden durch einen Regulären Ausdruck beschrieben. Der User, welcher einen Eintrag angelegt hat wird auch gespeichert.

TODO: Die Anwendung soll Persistenz enthalten sowie verschiedene Methoden zum Durchsuchen (nach Kategorie, User) und Exportieren der Daten.

2.0.2 Wie startet man die Applikation?

[Wie startet man die Applikation? Welche Voraussetzungen werden benötigt? Schritt-für-Schritt-Anleitung]

1. Voraussetzungen:
   - Java 17

- Maven

2. Compilieren

   ```
   mvn clean install
   ```

3. Ausführen

   ```
   java -jar 0-Plugin/target/0-Plugin-1.0-SNAPSHOT-jar-with-dependencies.jar [PARAMET
   ```

2.0.3 Wie testet man die Applikation?

[Wie testet man die Applikation? Welche Voraussetzungen werden benötigt? Schritt-für-Schritt-Anleitung]

```
mvn clean test
```

3 Kapitel 2: Clean Architecture

3.0.1 Was ist Clean Architecture?

[allgemeine Beschreibung der Clean Architecture in eigenen Worten]

Die Clean Architecture ist eine Software Architektur, die es ermöglichen soll langlebige Systeme zu entwickeln. Dazu wird der eigentliche Zweck einer Anwendung von möglichst vielen technischen Details getrennt. Auf diese Weise soll ein Kern der Anwendung entstehen, welcher beispielsweise die Businessregeln enthält und bis auf die Wahl der Programmiersprache (für Langlebigkeit von Sprachen siehe bspw. Java) keine Abhängigkeiten zu technischen Entscheidungen hat. Konkretere technische Details, wie beispielsweise die Wahl einer Datenbank oder ob für die Benutzerschnittstelle ein CLI oder ein Webserver genutzt wird, werden an den Rand der Anwendung gedrängt und nur durch Zwischenschichten mit dem Kern verbunden.

Die konkreten Schichten sind (von langlebig nach kurzlebig und von wenigen (keinen) nach vielen Abhängigkeiten sortiert):

- Abstraction Code
- Domain Code
- Application Code
- Adapters

- Plugins

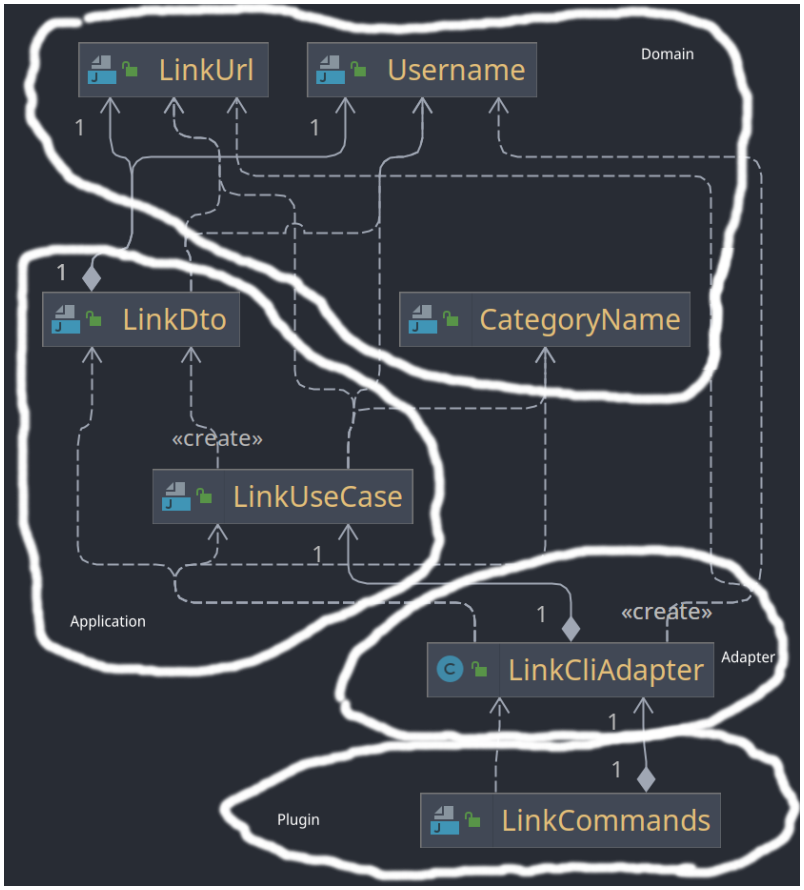
Durch die Dependency Rule wird sichergestellt, dass Abhängigkeiten immer von außen nach innen sind, und somit eine äußere Schicht ausgetauscht werden könnte, ohne, dass die inneren Schichten angepasst werden müssten.

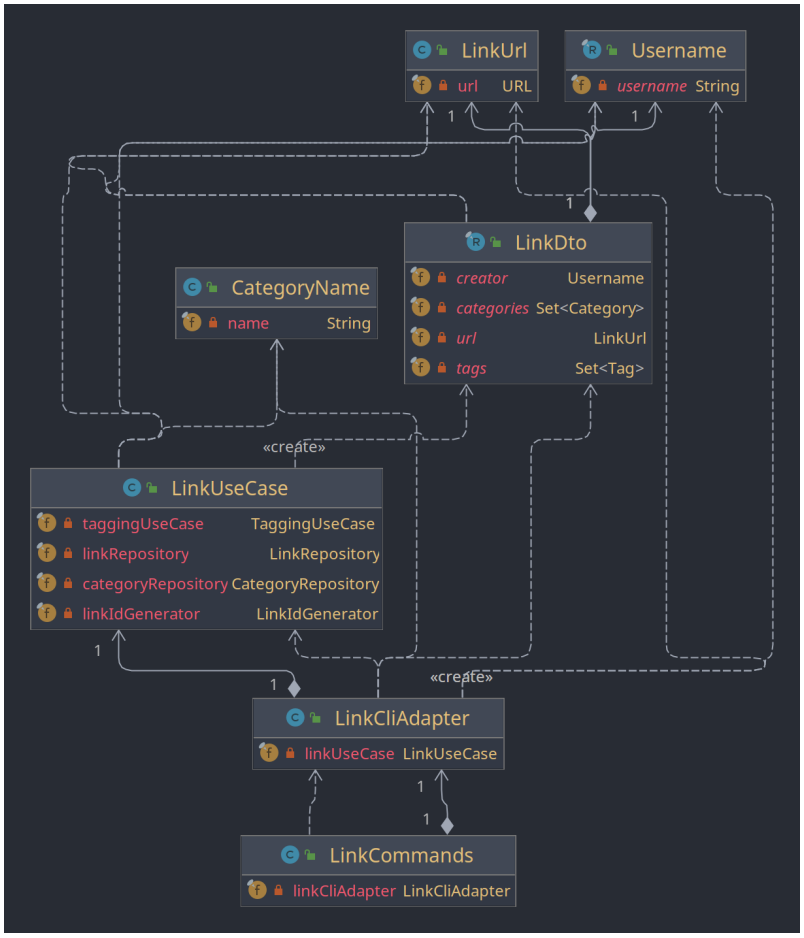
3.0.2 Analyse der Dependency Rule

[(1 Klasse, die die Dependency Rule einhält und eine Klasse, die die Dependency Rule verletzt); jeweils UML der Klasse und Analyse der Abhängigkeiten in beide Richtungen (d.h., von wem hängt die Klasse ab und wer hängt von der Klasse ab) in Bezug auf die Dependency Rule]

Da die Abhängigkeiten zwischen den einzelnen Schichten durch Maven restriktiv kontrolliert werden gibt es kein negativ Beispiel.

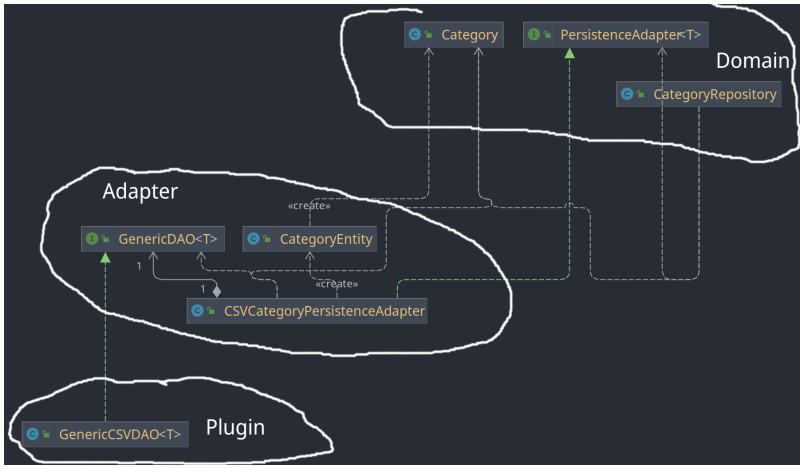
1. Positiv-Beispiel: Dependency Rule

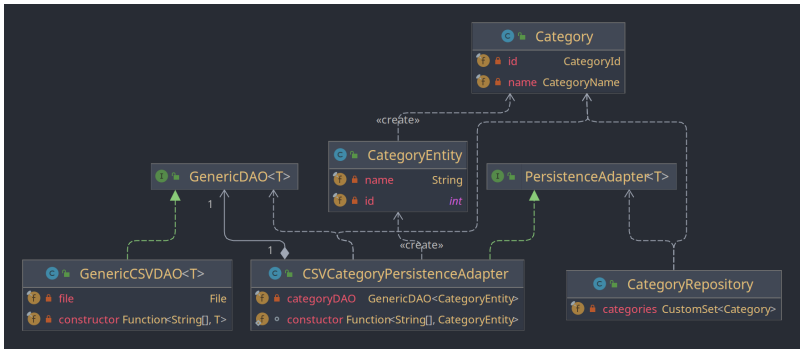




Die Klasse LinkCliAdapter ist selbst abhängig von einigen Value Object der Domäne (LinkUrl, UserName, CategoryName) und dem LinkUseCase sowie seinem speziellen Format LinkDto aus der Application Schicht. Abhängig von der Klasse LinkCliAdapter ist die Klasse LinkCommands aus der Plugin Schicht.

2. 2. Positiv-Beispiel: Dependency Rule





Die Klasse CSVCategoryPersistenceAdapter ist abhängig von dem Domänen Entity Category und Implementiert das Interface der Domäne PersistenceAdapter<Category>. Außerdem ist es abhängig vom Persitenz Entity CategorEntity, das in der Adapter Schicht definiert ist und dem Interface GenericDAO<CategorEntity> aus der Adapter Schicht. In der Plugin Schicht ist mit dem GenericCSVDAO<T> eine Klasse gegeben, die dieses Interface implementiert.

Das Domänen Repository CategoryRepository ist abhängig von einem PersistenceAdapter (Interface), welche das CSVCategoryPersistenceAdapter implementiert. Somit ist das Repository der Domäne zur Compile Zeit nicht abhängig von dem CSVCategoryPersistenceAdapter des Adapter sondern nur zur Runtime, da im Adapter eine Implementation des benötigten Interfaces liegt.

3.0.3 Analyse der Schichten

[jeweils 1 Klasse zu 2 unterschiedlichen Schichten der Clean-Architecture: jeweils UML der Klasse (ggf. auch zusammenspielenden Klassen), Beschreibung der Aufgabe, Einordnung mit Begründung in die Clean-Architecture]

1. Schicht: Domain

Die Klasse LinkUrl ist ein Klasse, welche einen zentralen Bestandteil der zu speichernden Daten repräsentiert: Die URL eines Links. Damit ist sie Teil der Domäne, da es sich direkt um die Businessregeln der zu verarbeiten Daten handelt. So stellt sie beispielsweise durch die Verwendung der Java Klasse URL sicher, dass die Url ein valides Format hat und somit die Domänenregeln erfüllt.

2. Schicht: Plugin

Die Klasse GitHubTagMatcher ist eine Implementation der des Interface TagMatcher und dafür verantwortlich festzustellen, ob ein Link auf eine GitHub Url verweist und im positiv Fall zu versuchen über die GitHub Repository-Api zusätzliche Informationen über das verlinkte Repository zu erhalten. Die Klasse ist Teil der Plugin Schicht, da die Interaktion mit der GitHub Repository-Rest-Api eindeutig eine Abhängigkeiten zu einem fremden Bestandteil darstellt und solche Abhängigkeiten an den Rand der Anwendung gedrängt werden sollten.

4 Kapitel 3: SOLID

4.0.1 Analyse Single-Responsibility-Principle (SRP)

[jeweils eine Klasse als positives und negatives Beispiel für SRP; jeweils UML der Klasse und Beschreibung der Aufgabe bzw. der Aufgaben und möglicher Lösungsweg des Negativ-Beispiels (inkl. UML)]

1. Positiv-Beispiel
2. Negativ-Beispiel

4.0.2 Analyse Open-Closed-Principle (OCP)

[jeweils eine Klasse als positives und negatives Beispiel für OCP; jeweils UML der Klasse und Analyse mit Begründung, warum das OCP erfüllt/nicht erfüllt wurde – falls erfüllt: warum hier sinnvoll/welches Problem gab es? Falls nicht erfüllt: wie könnte man es lösen (inkl. UML)?]

1. Positiv-Beispiel
2. Negativ-Beispiel

4.0.3 Analyse Liskov-Substitution- (LSP), Interface-Segreggation- (ISP), Dependency-Inversion-Principle (DIP)

[jeweils eine Klasse als positives und negatives Beispiel für entweder LSP oder ISP oder DIP); jeweils UML der Klasse und Begründung, warum man hier das Prinzip erfüllt/nicht erfüllt wird]

[Anm.: es darf nur ein Prinzip ausgewählt werden; es darf NICHT z.B. ein positives Beispiel für LSP und ein negatives Beispiel für ISP genommen werden]

1. Positiv-Beispiel
2. Negativ-Beispiel

5 Kapitel 4: Weitere Prinzipien

5.0.1 Analyse GRASP: Geringe Kopplung

[jeweils eine bis jetzt noch nicht behandelte Klasse als positives und negatives Beispiel geringer Kopplung; jeweils UML Diagramm mit zusammenspielenden Klassen, Aufgabenbeschreibung und Begründung für die Umsetzung der

geringen Kopplung bzw. Beschreibung, wie die Kopplung aufgelöst werden kann]

1. Positiv-Beispiel
2. Negativ-Beispiel

5.0.2 Analyse GRASP: Hohe Kohäsion

[eine Klasse als positives Beispiel hoher Kohäsion; UML Diagramm und Begründung, warum die Kohäsion hoch ist]

5.0.3 Don't Repeat Yourself (DRY)

[ein Commit angeben, bei dem duplizierter Code/duplizierte Logik aufgelöst wurde; Code-Beispiele (vorher/nachher); begründen und Auswirkung beschreiben]

6

7 Kapitel 5: Unit Tests

7.0.1 10 Unit Tests

[Nennung von 10 Unit-Tests und Beschreibung, was getestet wird]

Unit Test Klasse#Methode	Beschreibung

7.0.2 ATRIP: Automatic

[Begründung/Erläuterung, wie 'Automatic' realisiert wurde]

7.0.3 ATRIP: Thorough

[jeweils 1 positives und negatives Beispiel zu 'Thorough'; jeweils Code-Beispiel, Analyse und Begründung, was professionell/nicht professionell ist]

7.0.4 ATRIP: Professional

[jeweils 1 positives und negatives Beispiel zu 'Professional'; jeweils Code-Beispiel, Analyse und Begründung, was professionell/nicht professionell ist]

7.0.5 Code Coverage

[Code Coverage im Projekt analysieren und begründen]

7.0.6 Fakes und Mocks

[Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten; zusätzlich jeweils UML Diagramm der Klasse]

8 Kapitel 6: Domain Driven Design

8.0.1 Ubiquitous Language

[4 Beispiele für die Ubiquitous Language; jeweils Bezeichung, Bedeutung und kurze Begründung, warum es zur Ubiquitous Language gehört]

Bezeichung	Bedeutung
Link	Steht für eine eindeutige URL, die von der Anwendung gespeichert werden soll
Category	Steht für eine Kategorie, die einem Link zugeordnet werden kann
Tag	Steht für einen Tag der automatisch bestimmten Link-Typen zugeordnet wird
*TagMatcher	Steht für eine spezielle Implementation des Interfaces TagMatcher (z.B. GitHubTagM

8.0.2 Entities

[UML, Beschreibung und Begründung des Einsatzes einer Entity; falls keine Entity vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]

8.0.3 Value Objects

[UML, Beschreibung und Begründung des Einsatzes eines Value Objects; falls kein Value Object vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]

Klasse: Category Name

8.0.4 Repositories

[UML, Beschreibung und Begründung des Einsatzes eines Repositories; falls kein Repository vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]

8.0.5 Aggregates

[UML, Beschreibung und Begründung des Einsatzes eines Aggregates; falls kein Aggregate vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]

9 Kapitel 7: Refactoring

9.0.1 Code Smells

[jeweils 1 Code-Beispiel zu 2 Code Smells aus der Vorlesung; jeweils Code-Beispiel und einen möglichen Lösungsweg bzw. den genommen Lösungsweg beschreiben (inkl. (Pseudo-)Code)]

1. Duplicated Code Da es bei Java keine Funktion zum durchsuchen eines Sets gibt wurde an mehreren Stellen ein Konstrukt, wie unten sichtbar verwendet. Dies macht den Code unleserlich und schwerer zu warten.

   ```
   // LinkRepository.java
   public Optional<Link> getById(LinkId id) {
       return links.stream().filter(link -> link.getId().equals(id)).findFirst();
   }

   public Optional<Link> getByUrl(LinkUrl url) {
       return links.stream().filter(link -> link.getUrl().equals(url)).findFirst();
   }
   ```

   Durch die Einführung des Dekorator-Entwurfsmuster für Set wurde jedoch auch eine eigene Implementation eines Sets eingeführt. Dadurch

konnte diese Set Implementation auch einfach durch eine find Methode ergänzt werden, wie dargestellt.

```
// CustomStrictSet.java
@Override
public Optional<T> find(Predicate<T> predicate) {
    return set.stream().filter(predicate).findFirst();
}
```

So wurde die Codezeile an vier Stellen ersetzt. Wenn das Refactoring nicht recht früh durchgeführt worden wäre, wären es eventuell sogar mehr Stellen geworden.

```
// LinkRepository.java
public Optional<Link> getById(LinkId id) {
    return links.find(link -> link.getId().equals(id));
}

public Optional<Link> getByUrl(LinkUrl url) {
    return links.find(link -> link.getUrl().equals(url));
}
```

Das Refactoring wurde mit Commit e4f1670742 durchgeführt.

9.0.2 2 Refactorings

[2 unterschiedliche Refactorings aus der Vorlesung anwenden, begründen, sowie UML vorher/nachher liefern; jeweils auf die Commits verweisen]

10 Kapitel 8: Entwurfsmuster

[2 unterschiedliche Entwurfsmuster aus der Vorlesung (oder nach Absprache auch andere) jeweils sinnvoll einsetzen, begründen und UML-Diagramm]

10.0.1 Entwurfsmuster: [Name]

10.0.2 Entwurfsmuster: [Name]