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

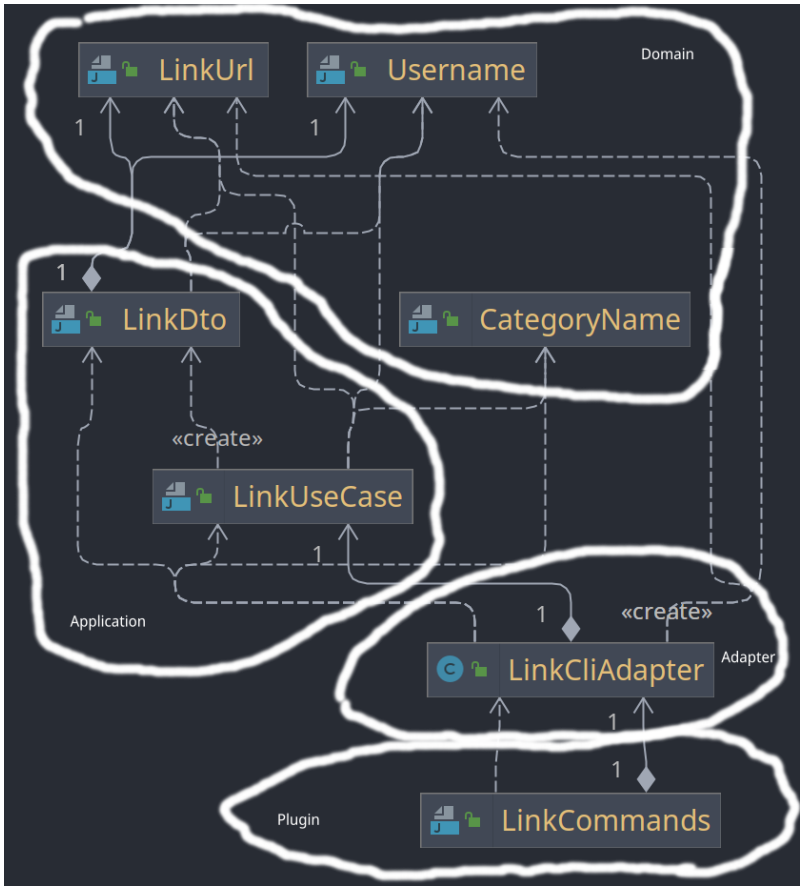
Durch die Dependency Rule wird sichergestellt, dass Abhängigkeiten immer von außen nach innen sind, und somit eine äußere Schicht ausgetauscht werden könnte, ohne, dass die inneren Schichten angepasst werden müssten.

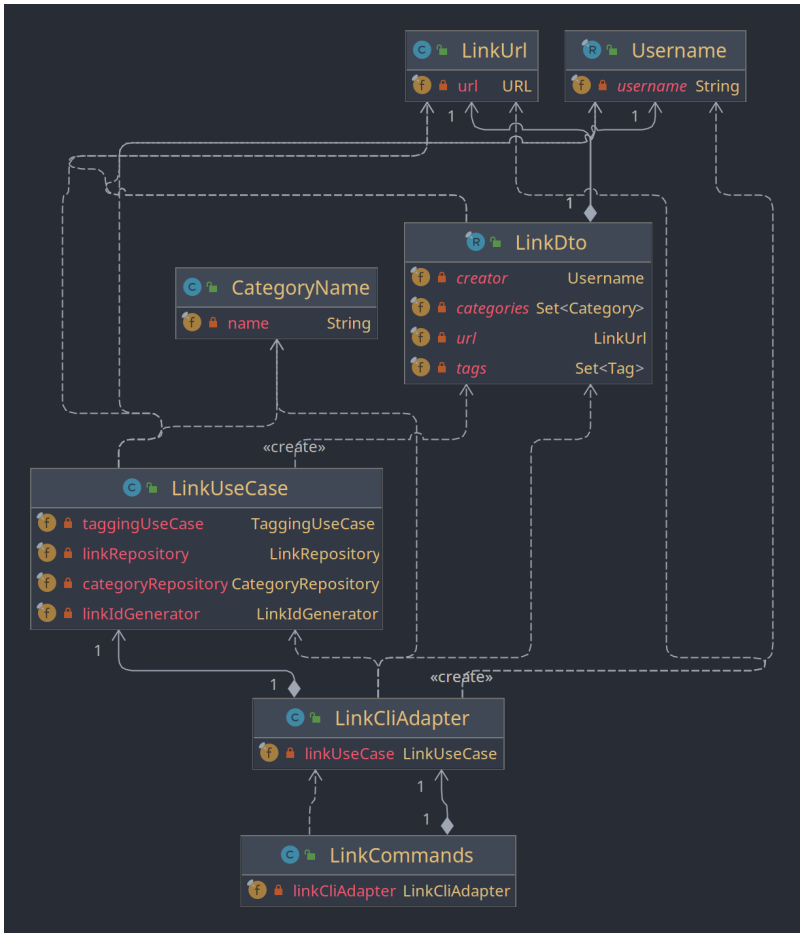
3.0.2 Analyse der Dependency Rule

[(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.

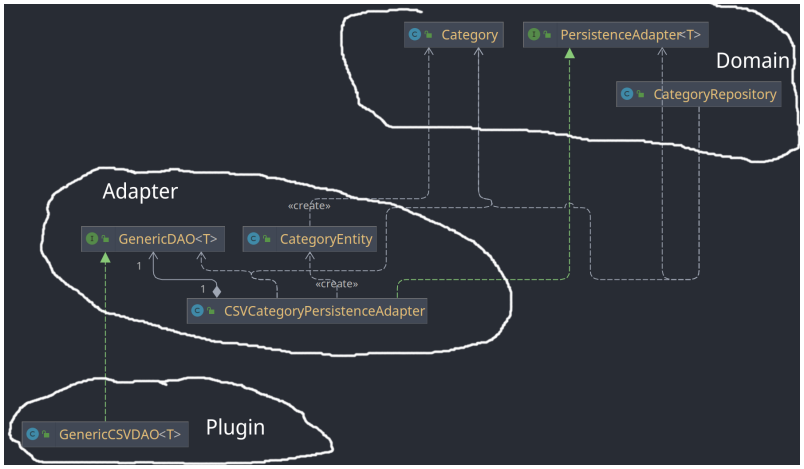
1. Positiv-Beispiel: Dependency Rule

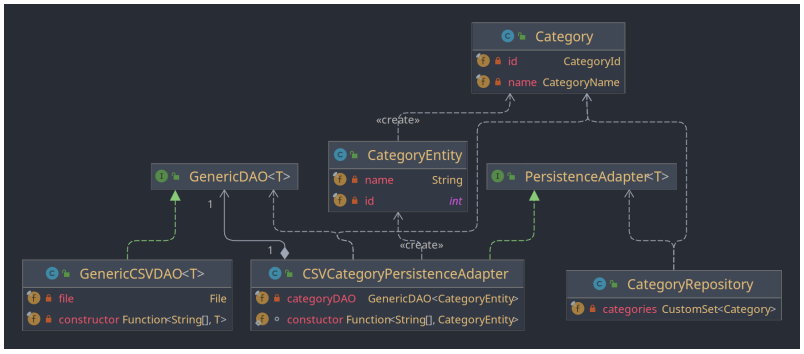




Die Klasse LinkCliAdapter ist selbst abhängig von einigen Value Object der Domäne (LinkUrl, UserName, CategoryName) und dem LinkUseCase sowie seinem speziellen Format LinkDto aus der Application Schicht. Abhängig von der Klasse LinkCliAdapter ist die Klasse LinkCommands aus der Plugin Schicht.

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

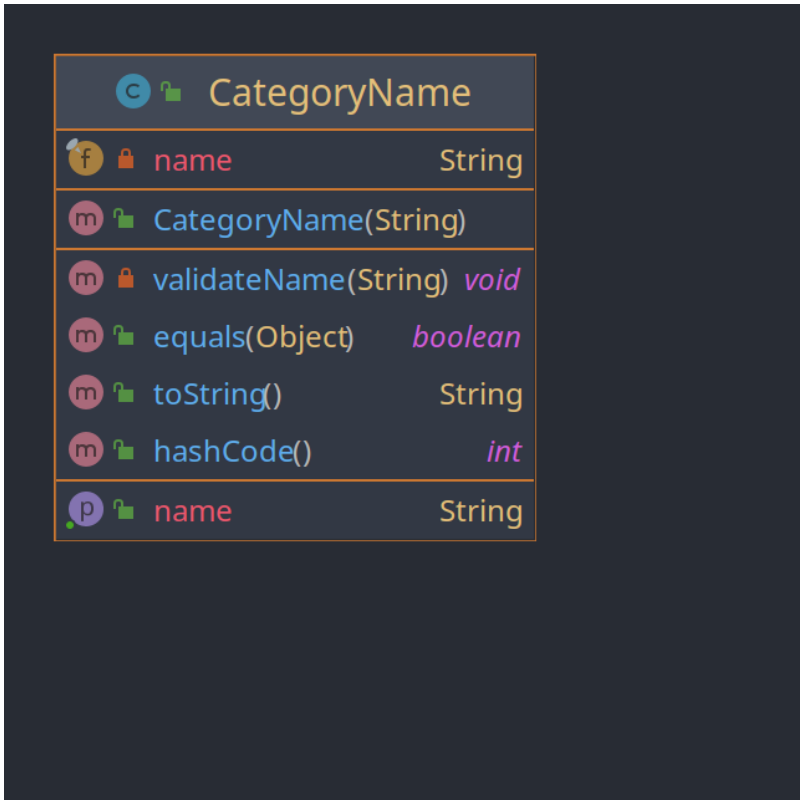
Die Klasse GitHubTagMatcher ist eine Implementation der des Interface TagMatcher und dafür verantwortlich festzustellen, ob ein Link auf eine GitHub Url verweist und im positiv Fall zu versuchen über die GitHub Repository-Api zusätzliche Informationen über das verlinkte Repository zu erhalten. Die Klasse 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)

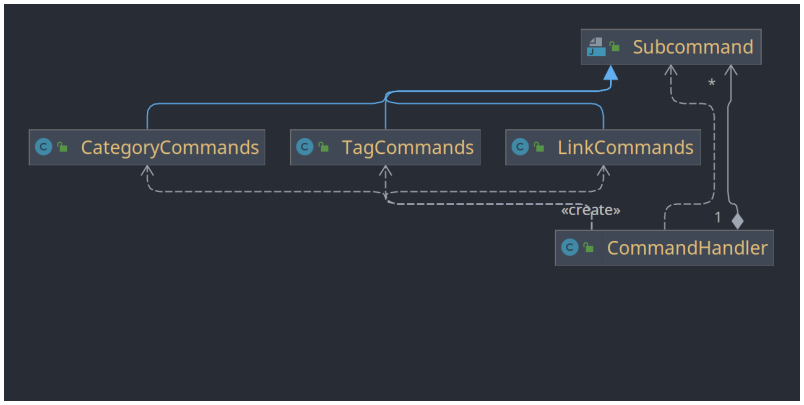
[jeweils eine Klasse als positives und negatives Beispiel für SRP; jeweils UML der Klasse und Beschreibung der Aufgabe bzw. der Aufgaben und möglicher Lösungsweg des Negativ-Beispiels (inkl. UML)]

1. Positiv-Beispiel



   Die Klasse CategoryName repräsentiert den Namen einer Kategorie und legt dabei fest, welche Werte dieser annehmen kann.

2. Negativ-Beispiel



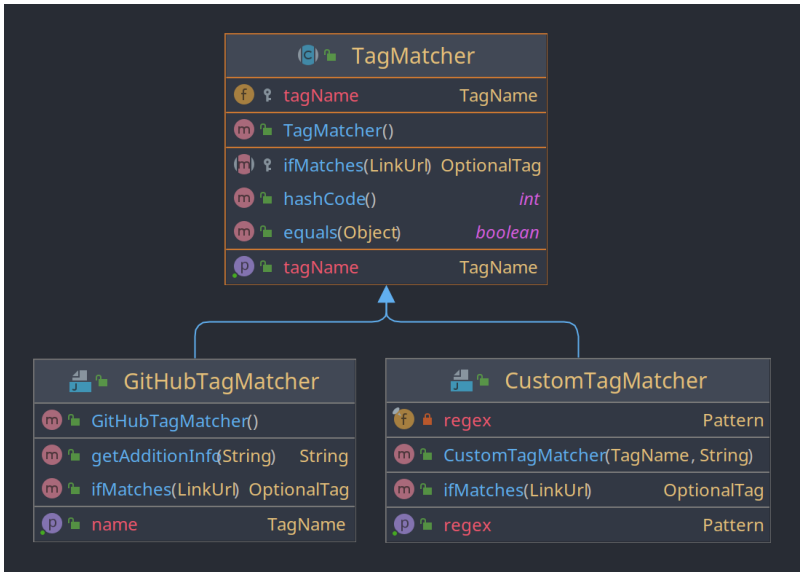
Die Klasse CommandHandler wird aufgerufen und leitet die CLI Parameter an die einzelnen SubCommand-Klassen weiter. Da die SubCommand-Klassen jedoch für ihre Konstruktoren, die Adapter benötigen, die Adapter wiederum die Usecases benötigen usw., wird der gesamte Baum an benötigten Klassen im Konstruktor der CommandHandler Klasse aufgebaut. Dies ist jedoch nicht ihre Responsibility. **Lösung**: Der

CommandHandler bekommt die SubCommand als Konstruktorparameter eingreicht und das Erstellen der restlichen Klassen wird von einer dedizierten Klasse durchgeführt. (UML nicht skizziert, weil es schneller ist den Code zu fixen und dann das UML zu generieren, als das UML per Hand zu machen.)

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



Die Klasse TagMatcher bietet das Interface (in diesem Fall als abstrakte Klasse) für alle möglichen Test, ob einer URL ein gewisser Tag zugeordnet werden kann.

Statt eines Switchstatements wie hier gezeigt:

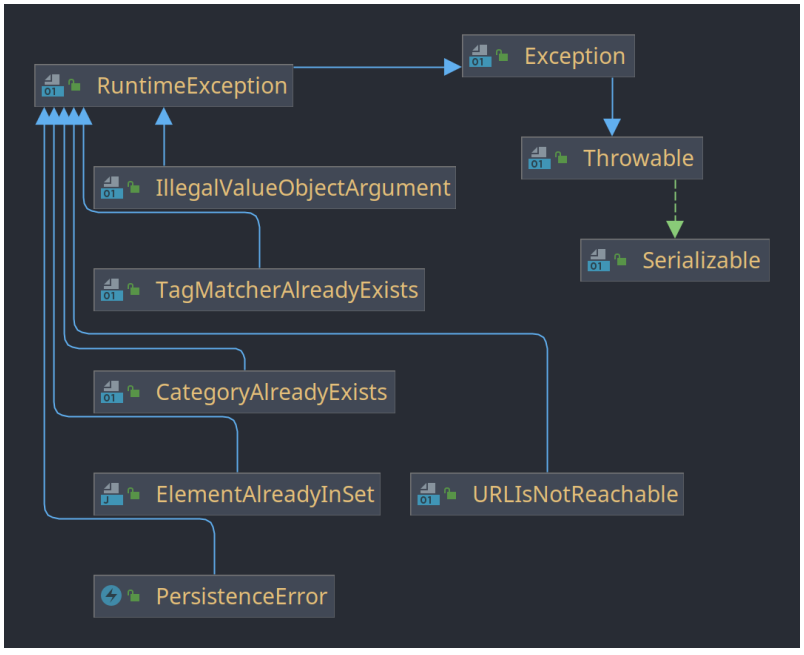
```
switch(url):
    case githubRegexPatter.matches(url):
      tags.add(new GitHubTag())
```

```
case someOtherMatcher.matches(url):
  tags.add(new GitHubTag())
```

Wird für einen Link von allen TagMatchern zur Laufzeit geprüft, ob dieser matcht. So können beispielsweise benutzerdefinierte Matcher verwendet werden (siehe Klasse CustomTagMatcher) und wenn ein neuer TagMatcher mit besonderer Implementation hinzufügt wird muss er nur die abstrakte Klasse erweitern und zur Liste die während der Dependecy-Injection-Phase gebaut wird hinzugefügt werden. In den Domain, Application und Adapter Schichten muss hierfür kein Code angepasst werden.

```
public Set<Tag> getTagsFor(LinkUrl url) {
                Set<Tag> result = new HashSet<>();
                tagMatcherRepository.getTagMatchers().forEach(
                     tagMatcher -> tagMatcher.ifMatches(url).
                        addTo(result));
                return result;
        }
```

2. Negativ-Beispiel



Die Exeptions der Anwendung erfüllen nicht wirklich das OCP. So gibt es einen Try-Catch-Block um die gesamte Anwendung.

```
try {
        commandHandler.executeCommand(args);
}
catch (PersistenceError persistenceError) {
        System.out.println("There was a Error with loading
                        or saving the persistence data.");
        System.out.println(persistenceError.getMessage());
}
catch (RuntimeException runtimeException) {
        System.out.println(runtimeException.getMessage());
}
```

Fügt man eine neue Exeption hinzu muss man zwar keinen Catch-Block hinzufügen um die Lauffähigkeit zu erhalten, es wäre jedoch für die Benutzerfreundlichkeit deutlich besser (vgl. extra Nachricht bei PersistenceError) wenn man es täte. Damit hierfür dann nicht für jede Exeption ein Catch-Block hinzugefügt werden muss sollten die Exeptions semantisch gruppiert werden und gemeinsame Elternklassen haben. So könnte man die Elternklasse Domainerror einfügen, für Exeptions, die innerhalb der Domäne liegen und keinen Programmfehler sondern eine falsche Nutzerhandlung bedeuten. Darunter würden Exeption wie CategoryAlreadyExists fallen. Sind diese definiert kann man leichter neue Exeptions hinzufügen ohne die Catchblöcke anpassen zu müssen oder dem Nutzer schlechte/inkonsistente Ausgaben zu geben.

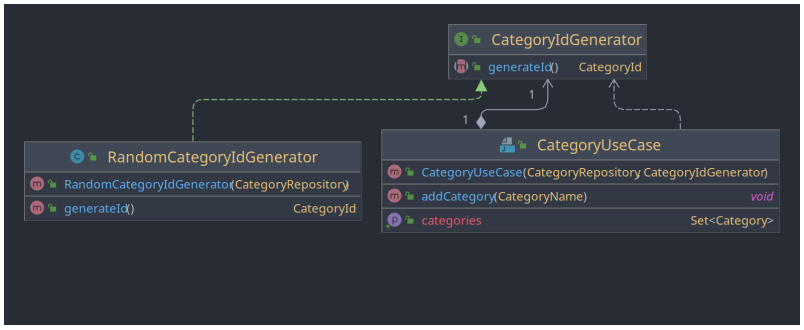
UML nicht vorhanden, da es schneller wäre den Fix einzubauen und das UML zu generieren als das UML von Hand zu bauen.

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]

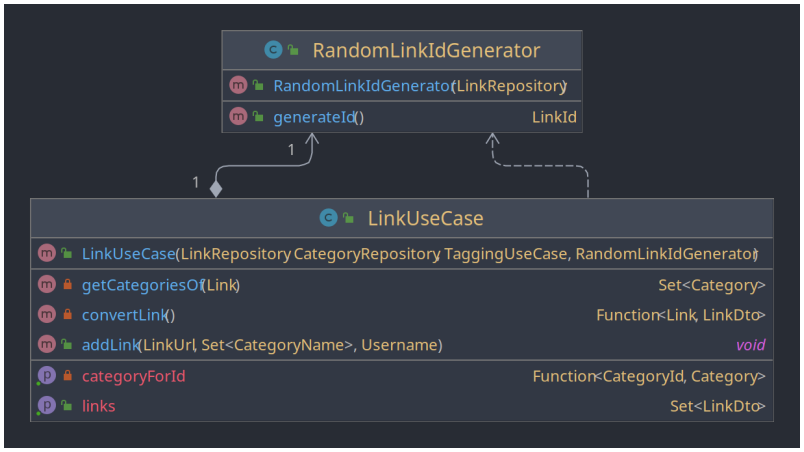
Dependency-Inversion-Principle

1. Positiv-Beispiel



Beim der Klasse CategoryIdGenerator wird das DIP erfüllt. Die Klasse CategoryUseCase ist nicht anhängig von einer konkreten Implementation eines IdGenerators wie dem RandomCategoryIdGenerator sondern von dem Interface CategoryIdGenerator. Dies ist auch besonders für Test praktisch, da man dann nicht mit Zufallszahlen umgehen muss.

2. Negativ-Beispiel



Beim der Klasse LinkUseCase wird das DIP nicht erfüllt. Die Klasse LinkUseCase ist anhängig von konkreten Implementation eines IdGenerators, dem RandomLinkIdGenerator.

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 zusammenspielen-

den 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]

Das Interface SubCommand definiert einen CLI SubCommand, der wiederum einzelne Funktionen hat. Diese einzelnen Funktionen werden in einer Map gespeichert, welche den Namen auf die Methode mappt. Die Logik hierfür war zunächst in jeder Implementation von SubCommand gleich implementiert.

```
public class CategoryCommands extends Subcommand {

        final private CategoryCliAdapter categoryCliAdapter;
        final private HashMap<String, Function<String[], String>> commands =
        new HashMap<>();

        @Override
        public String executeSubcommand(String[] args) {
                return commands.get(args[0]).apply(args);
        }
}
```

Indem das Interface SubCommand zu einer abstrakten Klasse umgebaut wurde, wurde die Logik an eine zentrale Stelle verschoben und zusätzlich gleich das benötigte Errorhandling eingebaut.

```
abstract public class Subcommand {

        public String executeSubcommand(String[] args);
```

```
        final public HashMap<String, Function<String[], String>> commands =
        new HashMap<>();

        abstract public String getSubcommand();

        abstract public String getUsage();

        public String executeSubcommand(String[] args) {
                try {
                        commandExsits(args[0]);
                        return commands.get(args[0]).apply(args);
                }
                catch (IndexOutOfBoundsException e) {
                        throw new CliError("Missing a value! " +
                                getUsage());
                }
        }

        private void commandExsits(String command) {
                if (commands.get(command) == null) {
                        throw new CliError("Subcommand does not exist! " +
                                getUsage());
                }
        }
}
```

Die angebenen Änderungen sind im Commit 78730bc69f sichtbar. Da die neuen Implementation des Interfaces im selben Commit hinzugefügt wurden ist, wurden diese direkt ohne den duplizierten Code committet.

6 Kapitel 5: Unit Tests

6.0.1 10 Unit Tests

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

1. CategoryIdTest#ConstructorWorks Stellt sicher, dass eine CategoryId mit einem int erstellt werden kann. Da ich vorher noch nie Java-Records verwendet hatte, war es ganz gut mit einem kleinen Test deren Funktionalität zu überprüfen. Gerne hätte ich auch einen Test zur

Unveränderbarkeit von Records gemacht, beim Versuch einen solchen zu schreiben wurde allerdings klar, dass es keine Methoden gibt die Veränderungen bewirken und Records somit die Anforderungen erfüllen (auch wenn es nicht testbar ist).

2. CategoryId#equalsWorks Stellt sicher, dass zwei CategoryIds die mit dem selben int erstellt wurden durch die equals Methode als gleich angesehen werden. Erneut eine Überprüfung, dass Records sich wie erwartet verhalten.

3. CategoryNameTest#getNameWorks Stellt sicher, dass der Getter für Name den erwarteten Wert zurück liefert.

4. CategoryNameTest#constructorThrowsNull,constructorThrowsBlank,constructorThrowsEmpty,c
Stellen sicher, dass die Regeln die für den Namen einer Category definiert sind auch korrekt überprüft werden und im Fehlerfall eine entsprechende Exeption geschmissen wird.

5. CategoryEntityTest#categoryConversionWorks Stellt sicher, dass bei der Konvertierung zwischen Category und CategorEntity durch die Funktionen toCategory und den Konstruktor. Die Komposition der beiden Funktionen sollte die Identitätsfunktion ergeben.

6. LinkEntityTest#toCSVString Stellt sicher, dass das serialisieren eines Objektes zu einem CSV String das erwartete Ergebnis liefert.

7. LinkEntityTest#fromCSVString Stellt sicher, dass das de-serialisieren eines CSV Strings das erwartete Ergebnis liefert.

8. CategoryCommandsTest#addCommandWorks Stellt sicher, dass beim Aufruf der Methode executeSubcommand mit dem String “add” und einem CategoryNamen die korrekte Methode des Adapters aufgerufen wird und die Parameter korrekt weitergereicht werden und eine passende Erfolgsmeldung geliefert wird.

9. GenericCSVDAOTest#addWorks Stellt sicher, dass nach dem Hinzufügen eines CategorEntitys dieses auch wieder gefunden werden kann.

10. GitHubTagMatcherTest#gettingDescriptionWorks Überprüft die Interaktion mit der GitHub Repository API indem für ein konkretes Repository der Wert abgefragt wird.

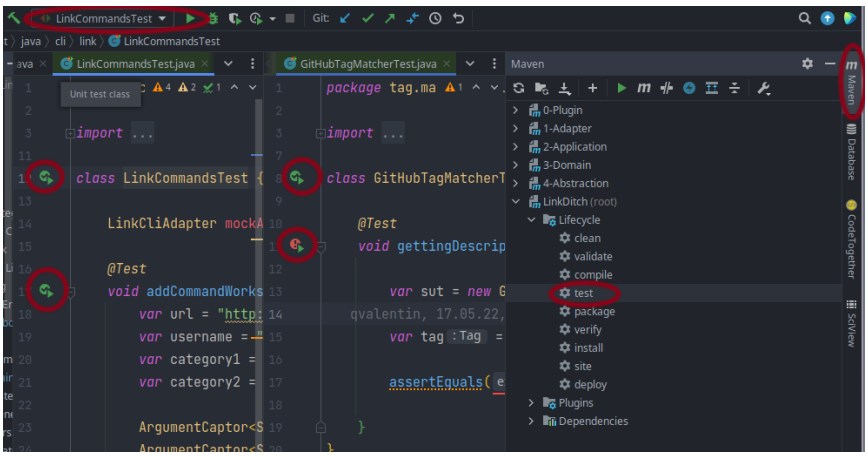
6.0.2 ATRIP: Automatic

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

Automatic wurde durch die einfache Ausführbarkeit realisiert. So muss nur ein Befehl ausgeführt werden um die Test zu starten.

```
mvn clean test
```

Alternativ genügen auch wenige Clicks bzw. Shortcuts in der IDE um die Test auszuführen und detailreiches Feedback über ihren Erfolg zu erhalten.

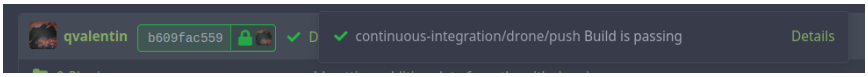


Die Test laufen ohne Eingaben und liefern immer ein Ergebnis, dass eindeutig Erfolg oder Misserfolg bezeugt.

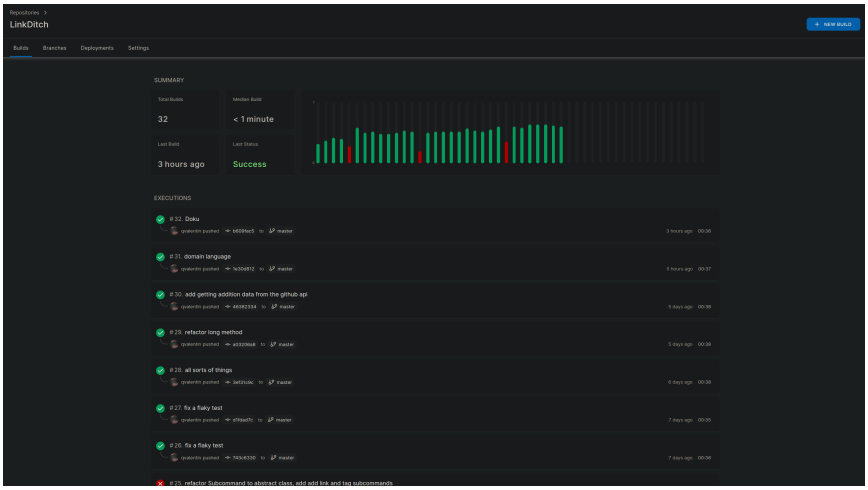
Damit man die Test nicht immer nur lokal ausführen muss werden sie auch bei jedem einchecken des Codes in das entfernte Versionskontrollrepository auf dem Server ausgeführt. Dies ist mit Drone CI realisiert und liefert in der Weboberfläche der Versionskontrolle schnelles Feedback.

```
---
kind: pipeline
type: docker
name: Tests Coverage

steps:
  - name: Run Tests With Coverage
    image: maven:3.8-openjdk-17-slim
    environment:
      SONAR_LOGIN:
```

```
      from_secret: SONAR_TOKEN
    commands:
      - mvn clean verify sonar:sonar -s ./settings.xml
trigger:
  branch:
    include:
      - master
  trigger:
    event:
      - push
```



Mit Drone CI bekommt man dann gleich auch einen guten Überblick, wie oft die Test fehlschlagen und wie lange das Testen braucht.



6.0.3 ATRIP: Thorough

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

1. Positives Beispiel Beim CategoryNameTest werden (bis auf generierten Code) sämtliche Methoden getestet und sämtliche Sonderfälle für die Eingaben in einzelnen Test geprüft (constructorThrowsNull,constructorThrowsBlank,constructorT

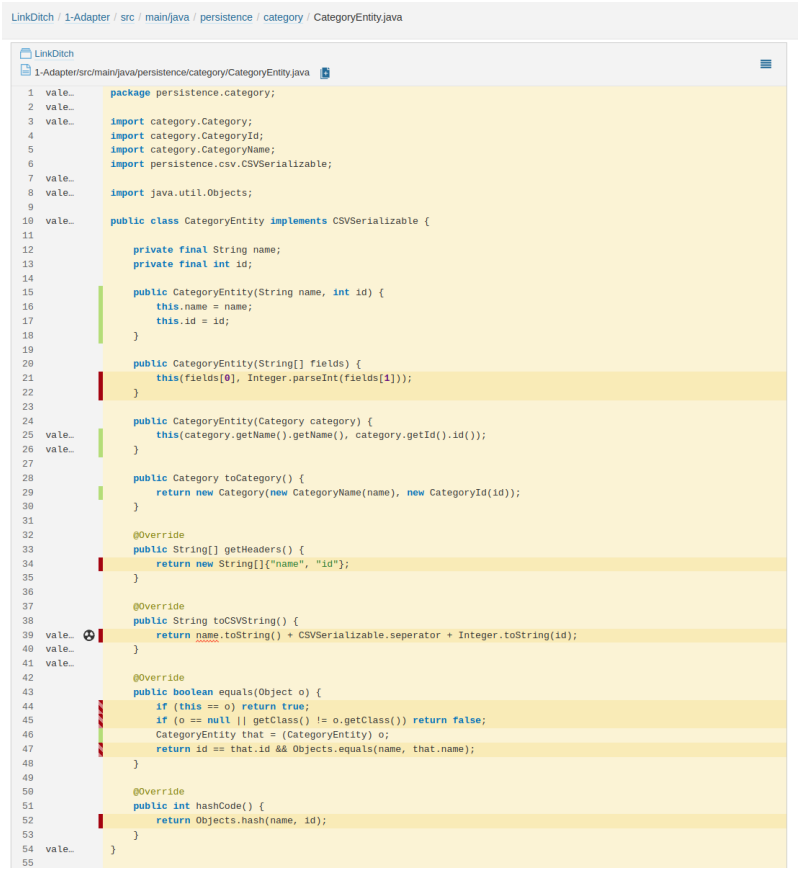
Coverage **53.3%**

LinkDitch

3-Domain/src/main/java/category/CategoryName.java

```
1  vale…  package category;
2  vale…
3  vale…  import exeptions.IllegalValueObjectArgument;
4  vale…
5         import java.util.Objects;
6
7         public class CategoryName {
8  vale…
9  vale…      private final String name;
10
11            public String getName() {
12                return name;
13            }
14
15 vale…      public CategoryName(final String name) {
16 vale…          validateName(name);
17                this.name = name;
18            }
19
20 vale…      private void validateName(final String name) {
21 vale…          if (name == null) {
22                    throw new IllegalValueObjectArgument("A Category name can not be null.");
23                }
24
25 vale…          if (name.isBlank() || name.length() < 3) {
26 vale…              throw new IllegalValueObjectArgument("A Category name must be a valid string of at least 3 characters.");
27                }
28            }
29
30            @Override
31            public boolean equals(Object o) {
32                if (this == o) return true;
33                if (o == null || getClass() != o.getClass()) return false;
34
35                CategoryName that = (CategoryName) o;
36
37                return Objects.equals(name, that.name);
38            }
39
40            @Override
41            public int hashCode() {
42                return name != null ? name.hashCode() : 0;
43            }
44 vale…
45            @Override
46            public String toString() {
47                        return name;
48            }
49 vale…  }
50
```

2. Negatives Beispiel Beim CategorEntityTest werden bis auf die beiden Koversationsmethoden Richtung Category keine Methoden getestet, obwohl beispielsweise die Konvertierung nach CSV leicht einen Fehler enthalten könnte, der Probleme verursachen würde (z.B. vergessenes toString).



6.0.4 ATRIP: Professional

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

1. Positives Beispiel Der Test GenericCSVDAOTest verwendet eine Datei. Damit dies sauber abläuft wird eine temporäre Datei verwendet.

   ```
   File file = File.createTempFile("test","link-ditch");
   ```

   Vor und nach jedem Test wird die Datei gesäubert, damit die Test alle im gleichen Zustand starten.

   ```
   @BeforeEach
   public void beforeEach() throws IOException {
   ```

```
        if (file.exists()) {
                file.delete();
        }
        file.createNewFile();

        this.sut = new GenericCSVDAO<>(file, CategoryEntity::new);
}

@AfterEach
void afterEach() {
        file.delete();
}
```

Damit die Tests lesbarer sind wird das recht aufwendige erzeugen eines CategoryEntitys und hinzufügen dessen in eine private Hilfsfunktion ausgelagert.

```
private CategoryEntity addDummyEntity(String categoryName, int id) {
        var entityToAdd = new CategoryEntity(categoryName, id);
        sut.add(entityToAdd);
        return entityToAdd;
}
```

Somit liest sich der removeAllWorks Test deutlich besser und duplizierter Code wird vermieden, was wiederum Fehler vermeidet.

```
@Test
public void removeAllWorks() throws IOException {
        addDummyEntity("categoryName1", 101);
        addDummyEntity("categoryName2", 102);
        assertEquals(2, sut.getALl().size());
        sut.removeAll();

        assertEquals(0, sut.getALl().size());
}
```

2. Negatives Beispiel

   Der Test addCommandWorks ist nicht sehr professionell. Es werden schlechte Variablennamen wie category1 und category2 verwendet. Die Verwendung des ArgumentCaptors macht den Code schlecht lesbar.

Immerhin werden Variablen verwendet und nicht die Strings an allen Stellen hardgecoded.

```
@Test
void addCommandWorks() {
        var url = "http://tea.filefighter.de";
        var username = "mario";
        var category1 = "funStuff";
        var category2 = "workStuff";

        ArgumentCaptor<String> captureUrl = ArgumentCaptor.forClass(String.class);
        ArgumentCaptor<String> captureUsername = ArgumentCaptor.forClass(String.cl
        ArgumentCaptor<Set<String>> captureCategories = ArgumentCaptor.forClass(Se

        doNothing()
                        .when(mockAdapter)
                        .addLink(captureUrl.capture(), captureCategories.capture()

        var sut = new LinkCommands(mockAdapter);
        var returnValue = sut.executeSubcommand(new String[]{"add", url, username,

        assertEquals("Added the new Link", returnValue);

        assertEquals(url, captureUrl.getValue());
        assertEquals(username, captureUsername.getValue());
        assertEquals(Set.of(category1, category2), captureCategories.getValue());
}
```

6.0.5 Zusatz: ATRIP: Repeatable

Der Commit d1fdad7cf9 zeigt den Fix für einen Test der nicht repeatable war, weil bei Sets die Reihenfolge der Elemente nicht eindeutig ist und somit zufällig.

6.0.6 Code Coverage

[Code Coverage im Projekt analysieren und begründen]

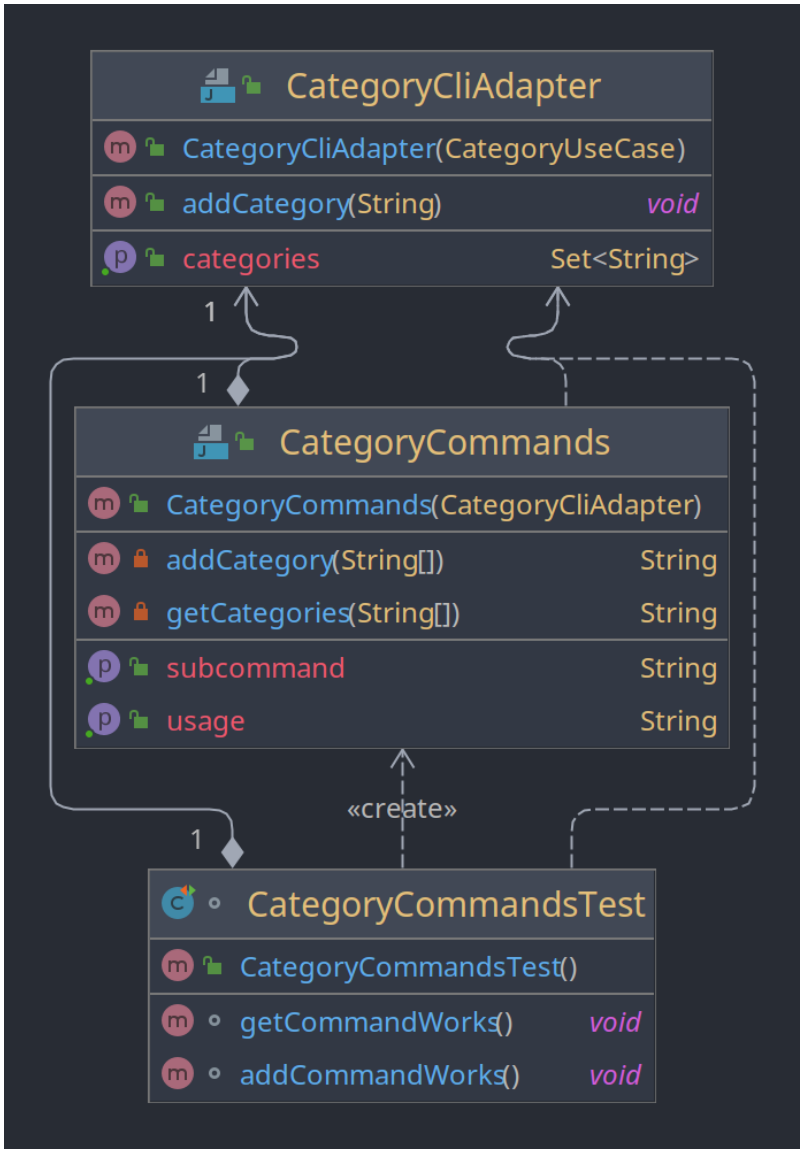
Die Coverage ist mit etwa 25 % deutlich zu niedrig. Da es sich bei dem Projekt jedoch um Code handelt, der niemals wirklich produktiv eingesetzt werden wird und der nicht langfristig weiterentwickelt wird, ist dies verkraftbar. Es wurden hauptsächlich die komplizierteren Stellen getestet,

wie beispielsweise die CSV-Persistierung und die Interaktion mit der Github-APi. Bei diesen Stellen wurden grade genug Test geschrieben, um sicherzustellen, dass die Grundfunktion korrekt ist. Teilweise wurde während des Entwicklungsprozesses gemerkt, dass es besser gewesen wäre, manche Stellen zu testen. Für Fehler die beim manuellen Testen der Anwendung aufgefallen waren wurden teilweise extra Tests geschrieben.

6.0.7 Fakes und Mocks

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

1. CategoryCommandsTest: Beim CategoryCommandsTest wurde die Klasse CategoryCommands erstellt, doch anstatt ein Objekt der Klasse CategoryCliAdapter beim Erstellen zu übergeben wurde ein Mock übergeben.



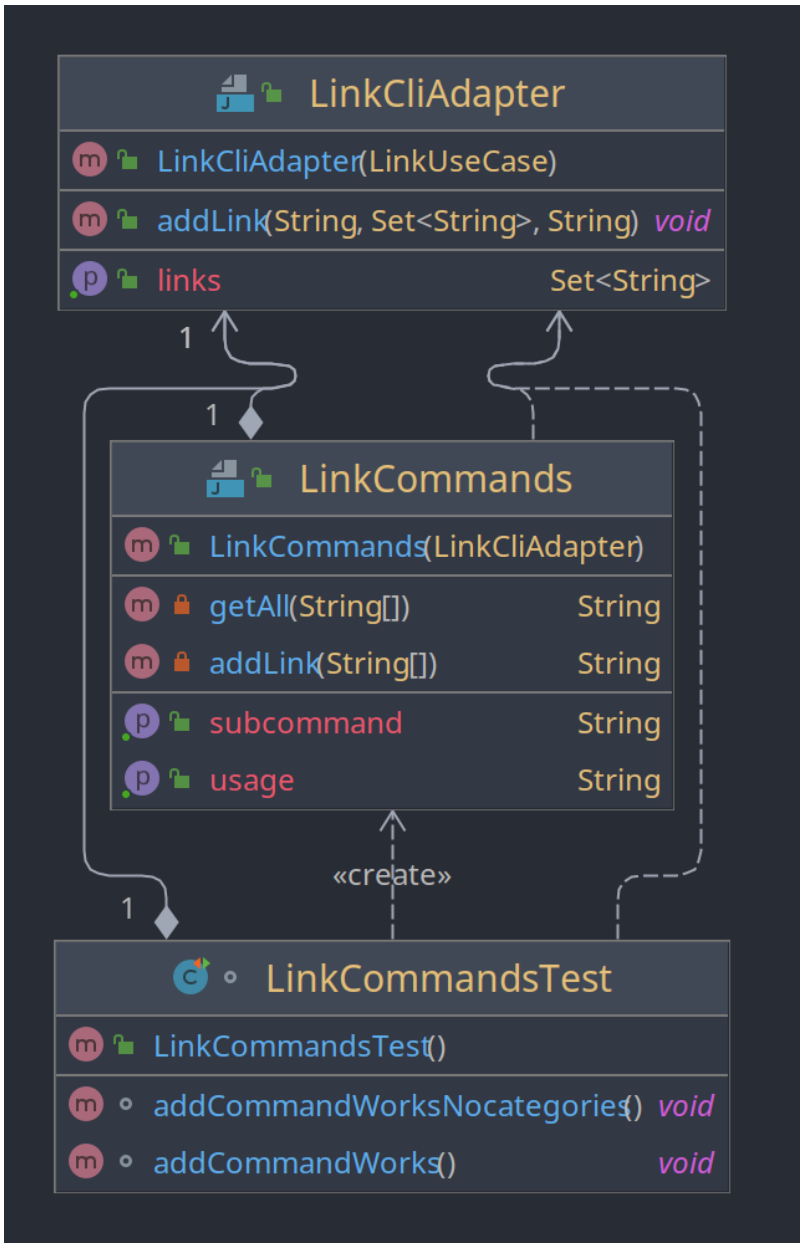
Dieses Mock wird dann aufgerufen und die Parameter des Aufrufs werden überprüft. Zusätzlich wird definiert, welche Rückgabewerte das Mock liefern soll.

```
class CategoryCommandsTest {

    CategoryCliAdapter mockAdapter = mock(CategoryCliAdapter.class);
```

```
    @Test
    void addCommandWorks() {
        var categoryName = "funStuff";
        ArgumentCaptor<String> valueCapture = ArgumentCaptor.forClass(String.c
        doNothing().when(mockAdapter).addCategory(valueCapture.capture());
        var sut = new CategoryCommands(mockAdapter);

        var returnValue = sut.executeSubcommand(new String[]{"add", categoryNa

        assertEquals(categoryName, valueCapture.getValue());
        assertEquals("Added the new category", returnValue);
    }

    @Test
    void getCommandWorks() {
        var sut = new CategoryCommands(mockAdapter);
        when(mockAdapter.getCategories()).thenReturn(Set.of("funStuff", "workS
        var returnValue = sut.executeSubcommand(new String[]{"get"});

        var expected =
                "Available Categories:" + System.lineSeparator() + "funStuff"

        var expectedDifferentOrder =
                "Available Categories:" + System.lineSeparator() + "workStuff"
        assertTrue(expected.equals(returnValue) || expectedDifferentOrder.equa
    }
}
```

Das Mock ist hier besonders nützlich, da wir uns an der Grenze zwischen zwei Schichten befinden. Für das Erstellen des CategoryCli-Adapters wird eine Usecase benötigt, welcher wiederum Instanzen aus der Domäne benötigt, welche wiederum bestimmte Instanzen benötigen. Indem wir stattdessen ein Mock erstellen werden quasi alle anderen Schichten weg abstrahiert. Dies ist auch empfehlenswert, da wir beim aktuellen Unittest ja nur die Funktionalität der aktuellen Klasse testen wollen. Deshalb definieren wir durch das Mock, wie sich der Rest der Anwendung verhalten sollte und können uns auf unsere aktuell Klasse konzentrieren und sind unabhängig von eventuellen Bugs in anderen Bereichen oder fehlenden Implementationen.

2. LinkCommandsTest:



Beim LinkCommandsTest wurde die Klasse LinkCommands erstellt, doch anstatt ein Objekt der Klasse LinkCliAdapter beim Erstellen zu übergeben wird ein Mock übergeben.

Auch hier befinden wir uns an der Grenze von zwei Schichten.

7 Kapitel 6: Domain Driven Design

7.0.1 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

7.0.2 Entities

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

7.0.3 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

7.0.4 Repositories

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

7.0.5 Aggregates

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

8 Kapitel 7: Refactoring

8.0.1 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.

8.0.2 2 Refactorings

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

9 Kapitel 8: Entwurfsmuster

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

9.0.1 Entwurfsmuster: Dekorator

9.0.2 Entwurfsmuster: [Name]