Programmentwurf - LinkDitch

valentin.theodor@web.de (Klarname und Matrikelnummer siehe Mail)

Abgabedatum 29.05.2022

Contents

1 Kapitel 1: Einführung

1.0.1 Übersicht über die Applikation

Der Name der Anwendung ergibt sich aus einem Wortspiel und ist inspiriert von dem Programm LinkSnitch.

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.

Die Anwendung enthält Persistenz in Form von CSV-Dateien sowie verschiedene Methoden zum Durchsuchen (nach Kategorie, User, Tag und gruppiert nach Hostname) der Daten.

1.0.2 Wie startet man die Applikation?

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
   ```

   oder alternativ mit dem beigelegten Shell-Script

   ```
   ./link-ditch [PARAMETER]
   ```

   Die Verfügbaren Commands der Anwendung sollten recht selbsterklärend sein und werden aufgelistet, wenn man die Anwendung ohne Parameter startet.

   ```
   $ ./link-ditch
   All available subcommands:

   Usage:
   link add http://example.org yourUsername [category1 category2 .. categoryN]
   link get
   link category aCategoryName
   link tag aTagName
   link user aUserName
   link hosts

   Usage:
   tag add tagName tagRegexExpression
   ```

```
Usage:
category add categoryName
category get
```

Die Speicherung der Daten erfolgt in CSV-Dateien, die im Workingdirectory abgelegt werden.

Eine beispielhafte Interaktion ist hier gegeben. Die Daten sollten auch im Repo vorhanden sein:

```
./link-ditch category add browser
./link-ditch category add privacy
./link-ditch link add https://github.com/libredirect/libredirect valentin privacy
./link-ditch link add https://github.com/nickspaargaren/no-google valentin androic
./link-ditch tag add emacswiki emacswiki.org
./link-ditch category add texteditor
./link-ditch link add https://www.emacswiki.org/emacs/WriteOrDieMode valentin text
./link-ditch link add http://www.vimgolf.com/  valentin texteditor
./link-ditch category add latex
./link-ditch link add https://github.com/JabRef/jabref  valentin latex
./link-ditch link add https://geschicktgendern.de/  valentin latex
./link-ditch link add https://latexeditor.lagrida.com/  valentin latex

./link-ditch link get
./link-ditch link hosts # gruppiert nach gleichen Hosts
./link-ditch link user valentin
./link-ditch link category privacy
./link-ditch link tag github


./link-ditch tag get
./link-ditch catgegory get
```

1.0.3 Wie testet man die Applikation?

```
mvn clean test
```

2 Kapitel 2: Clean Architecture

2.0.1 Was ist Clean Architecture?

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):

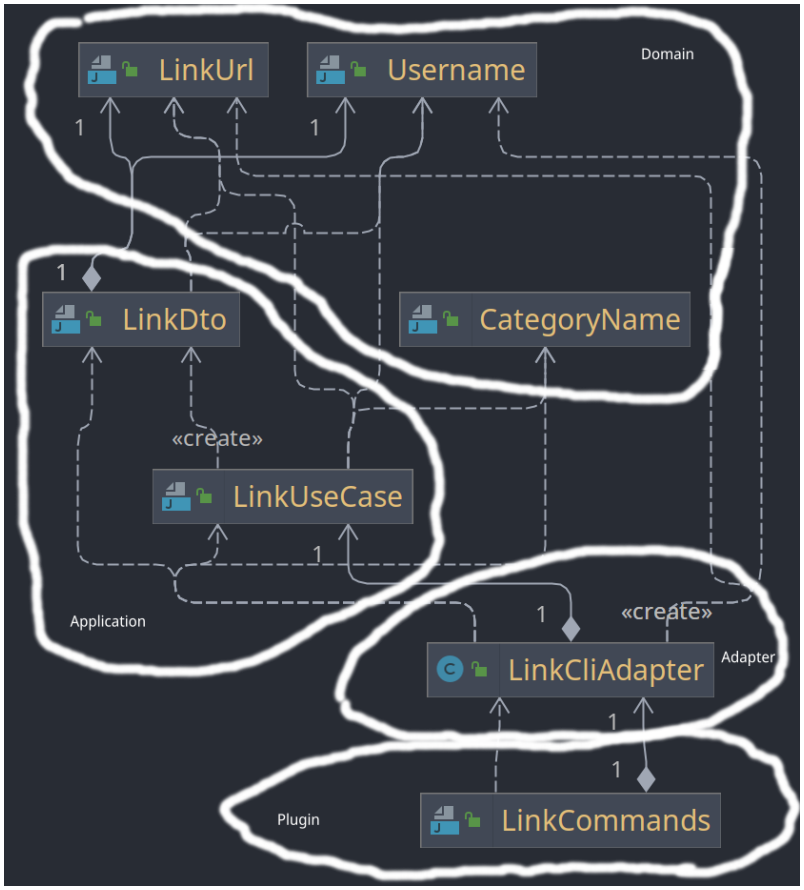
- Abstraction Code
- Domain Code
- Application Code
- Adapters
- Plugins

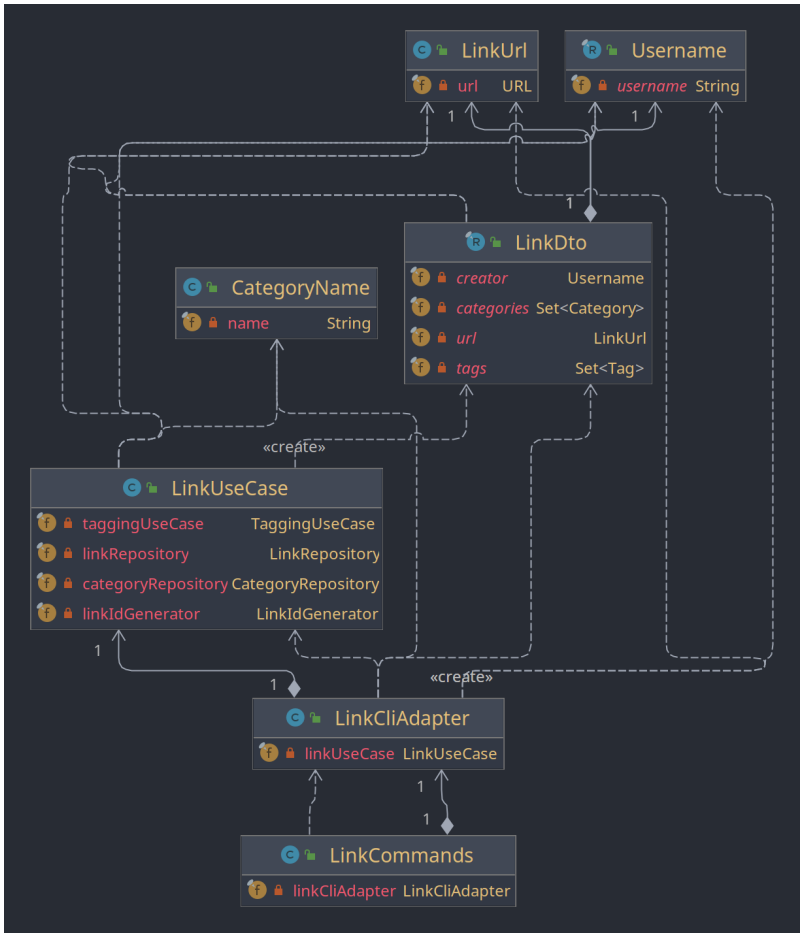
Durch die Dependency Rule wird sichergestellt, dass Abhängigkeiten immer von außen nach innen sind, und somit eine äußere Schicht ausgetauscht werden könnte, ohne, dass die inneren Schichten angepasst werden müssten.

2.0.2 Analyse der Dependency Rule

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

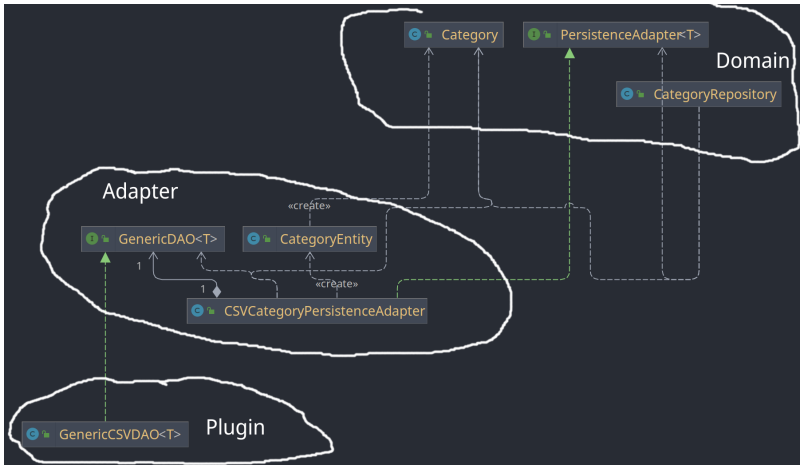
1. Positiv-Beispiel: Dependency Rule

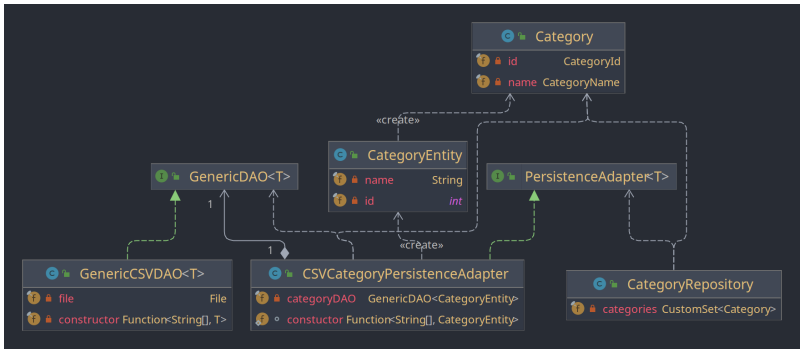

Domain
LinkUrl
Username
1
1
1
LinkDto
CategoryName
«create»
LinkUseCase
1
Application
1
«create»
Adapter
LinkCliAdapter
1
1
Plugin
LinkCommands



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



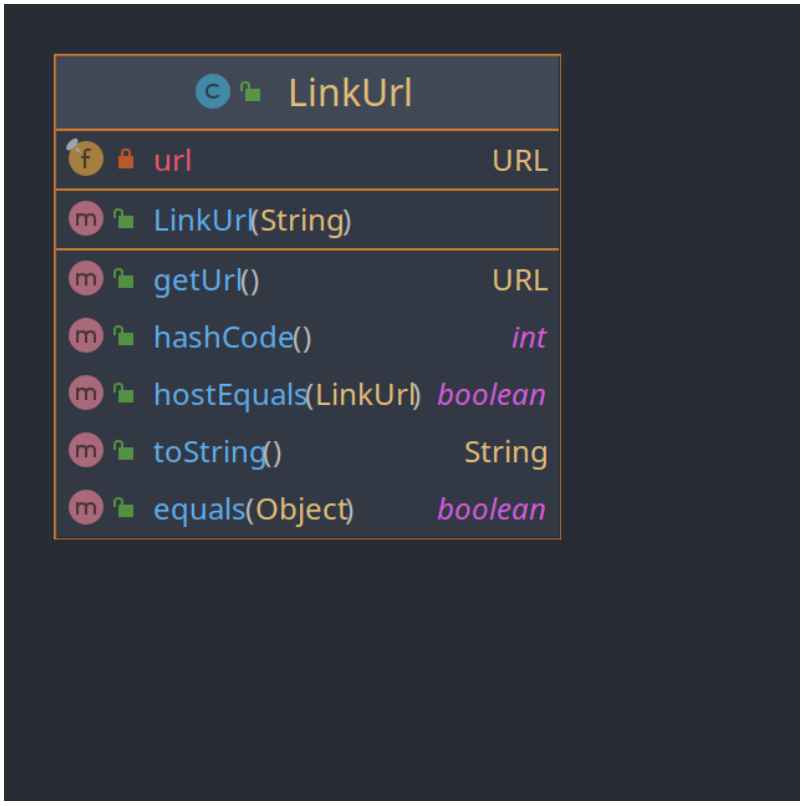


Die Klasse CSVCategoryPersistenceAdapter ist abhängig von dem Domänen Entity Category und Implementiert das Interface der Domäne PersistenceAdapter<Category>. Außerdem ist es abhängig vom Persitenz Entity CategorEntity, das in der Adapter Schicht definiert ist und dem Interface GenericDAO<CategorEntity> 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.

2.0.3 Analyse der Schichten

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.

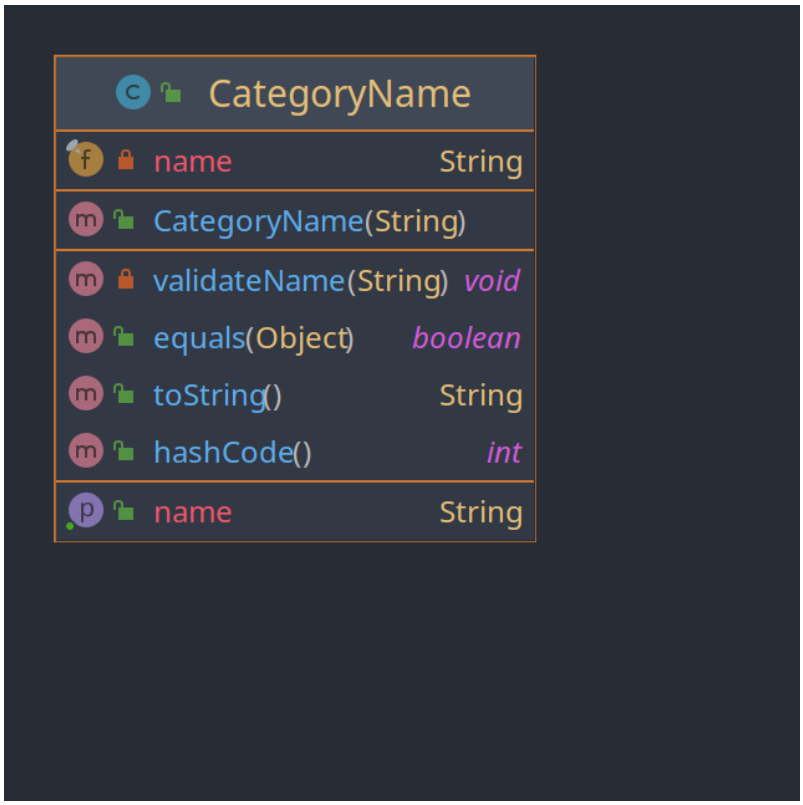
2. Schicht: Plugin

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.

3 Kapitel 3: SOLID

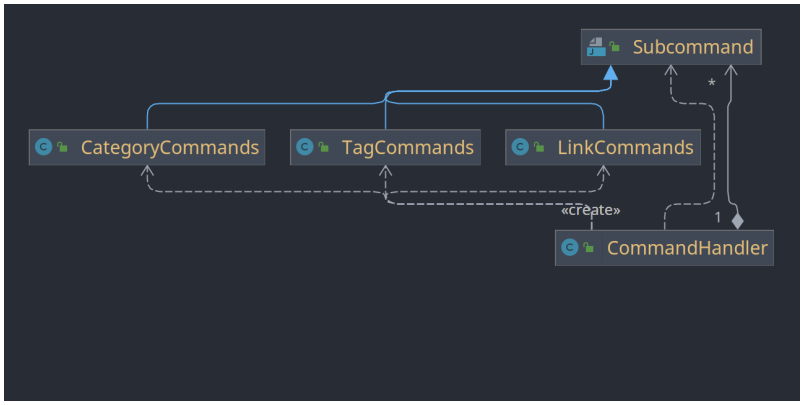
3.0.1 Analyse Single-Responsibility-Principle (SRP)

1. Positiv-Beispiel



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

2. Negativ-Beispiel

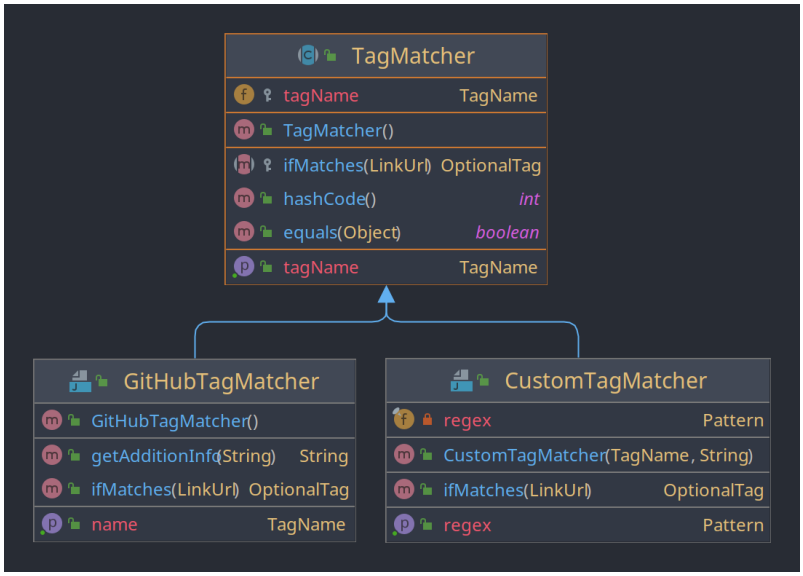


Die Klasse CommandHandler wird aufgerufen und leitet die CLI Parameter an die einzelnen SubCommand-Klassen weiter. Da die SubCommand-Klassen jedoch für ihre Konstruktoren, die Adapter benötigen, die Adapter wiederum die Usecases benötigen usw., wird der gesamte Baum an benötigten Klassen im Konstruktor 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.)

3.0.2 Analyse Open-Closed-Principle (OCP)

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:

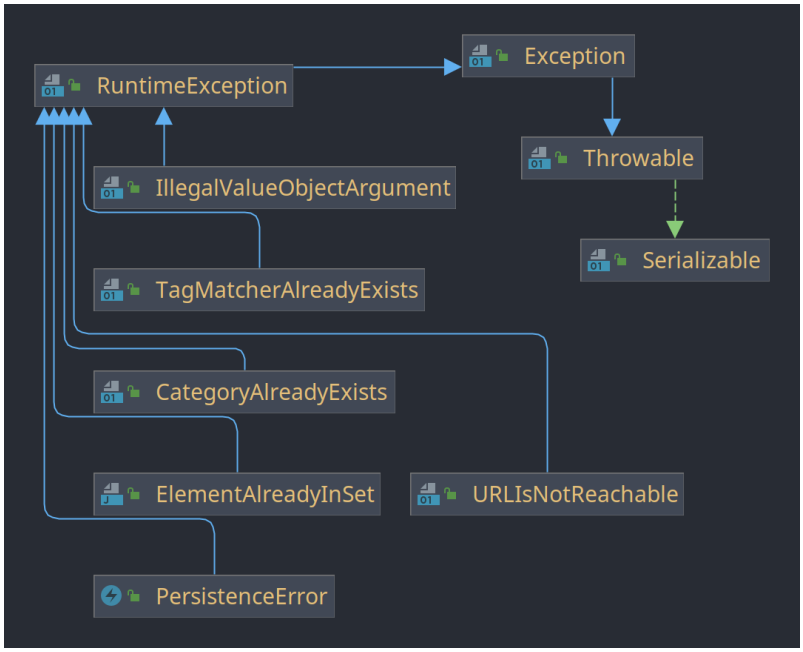
```
switch(url):
    case githubRegexPatter.matches(url):
      tags.add(new GitHubTag())
    case someOtherMatcher.matches(url):
      tags.add(new GitHubTag())
```

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());
}
```

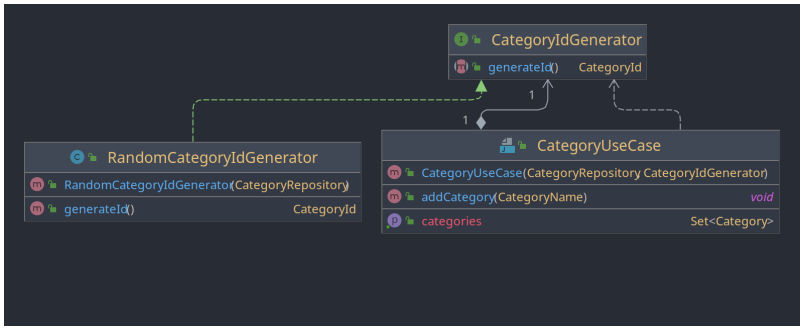
Fügt man eine neue Exeption hinzu muss man zwar keinen Catch-Block hinzufügen um die Lauffähigkeit zu erhalten, es wäre jedoch für die Benutzerfreundlichkeit deutlich besser (vgl. extra Nachricht bei PersistenceError) wenn man es täte. Damit hierfür dann nicht für jede Exeption ein Catch-Block hinzugefügt 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.

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

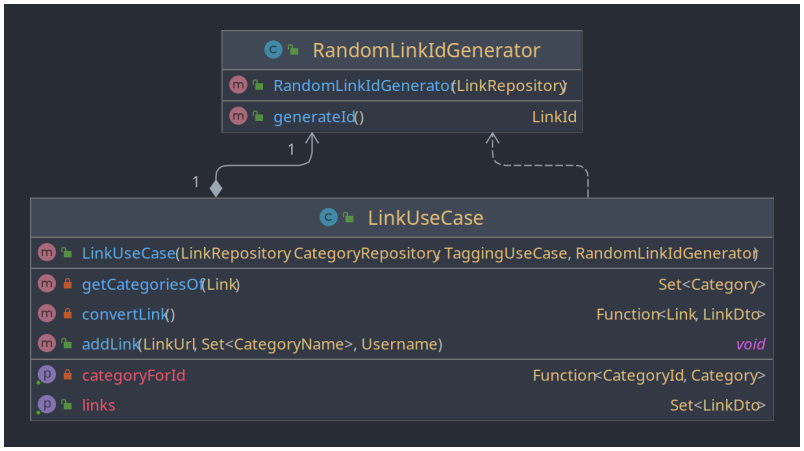
Dependency-Inversion-Principle

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

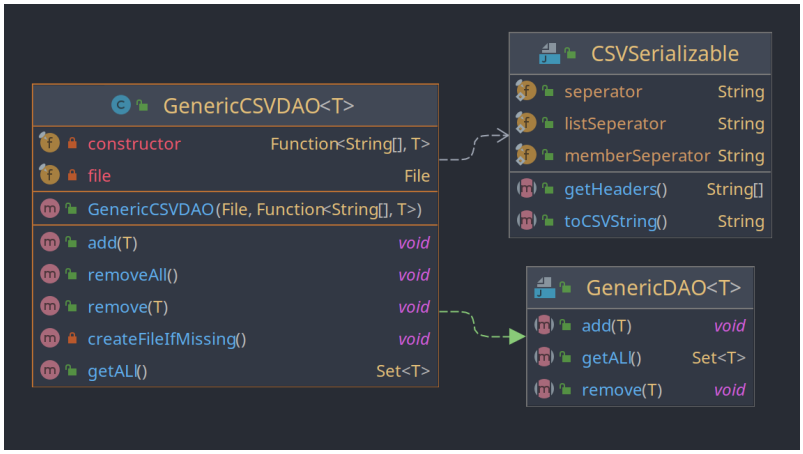


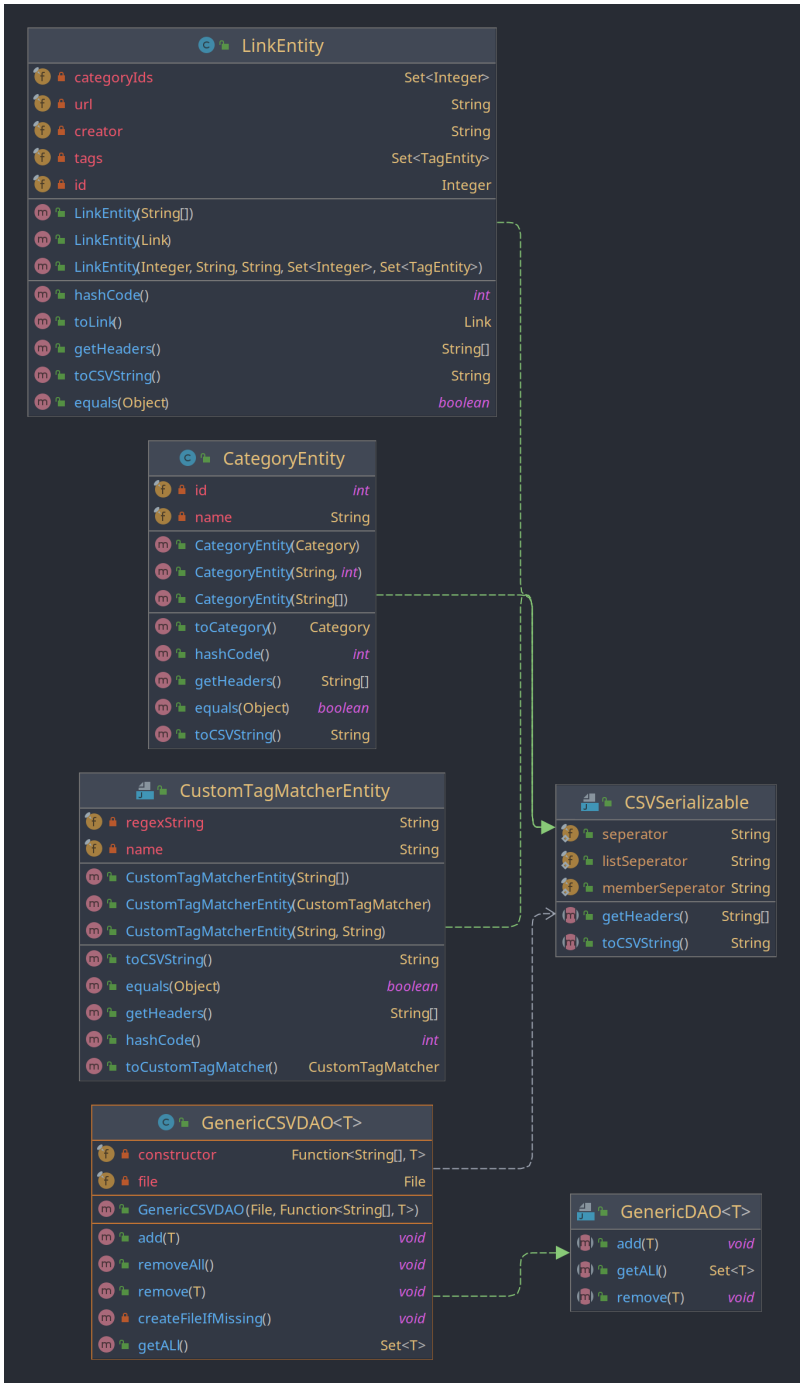
Beim der Klasse LinkUseCase wird das DIP nicht erfüllt. Die Klasse LinkUseCase ist anhängig von konkreten Implementation eines IdGenerators, dem RandomLinkIdGenerator.

4 Kapitel 4: Weitere Prinzipien

4.0.1 Analyse GRASP: Geringe Kopplung

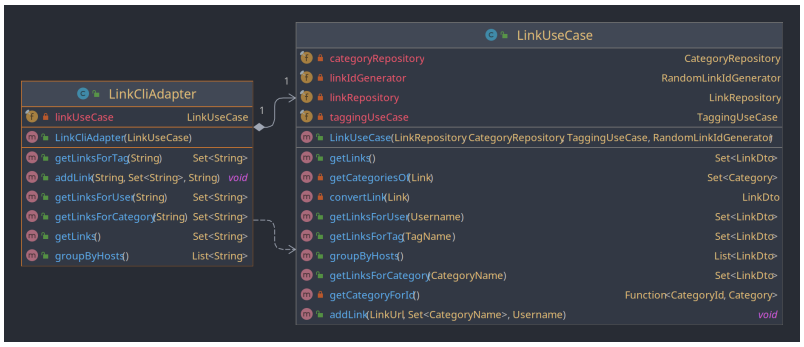
1. Positiv-Beispiel


GenericCSVDAO<T>
constructor Function<String[], T>
file File
GenericCSVDAO(File, Function<String[], T>)
add(T) void
removeAll() void
remove(T) void
createFileIfMissing() void
getALl() Set<T>
CSVSerializable
seperator String
listSeperator String
memberSeperator String
getHeaders() String[]
toCSVString() String
GenericDAO<T>
add(T) void
getALl() Set<T>
remove(T) void


LinkEntity
categoryIds Set<Integer>
url String
creator String
tags Set<TagEntity>
id Integer
LinkEntity(String[])
LinkEntity(Link)
LinkEntity(Integer, String, String, Set<Integer>, Set<TagEntity>)
hashCode() int
toLink() Link
getHeaders() String[]
toCSVString() String
equals(Object) boolean
CategoryEntity
id int
name String
CategoryEntity(Category)
CategoryEntity(String, int)
CategoryEntity(String[])
toCategory() Category
hashCode() int
getHeaders() String[]
equals(Object) boolean
toCSVString() String
CustomTagMatcherEntity
regexString String
name String
CustomTagMatcherEntity(String[])
CustomTagMatcherEntity(CustomTagMatcher)
CustomTagMatcherEntity(String, String)
toCSVString() String
equals(Object) boolean
getHeaders() String[]
hashCode() int
toCustomTagMatcher() CustomTagMatcher
CSVSerializable
seperator String
listSeperator String
memberSeperator String
getHeaders() String[]
toCSVString() String
GenericCSVDAO<T>
constructor Function<String[], T>
file File
GenericCSVDAO(File, Function<String[], T>)
add(T) void
removeAll() void
remove(T) void
createFileIfMissing() void
getALl() Set<T>
GenericDAO<T>
add(T) void
getALl() Set<T>
remove(T) void

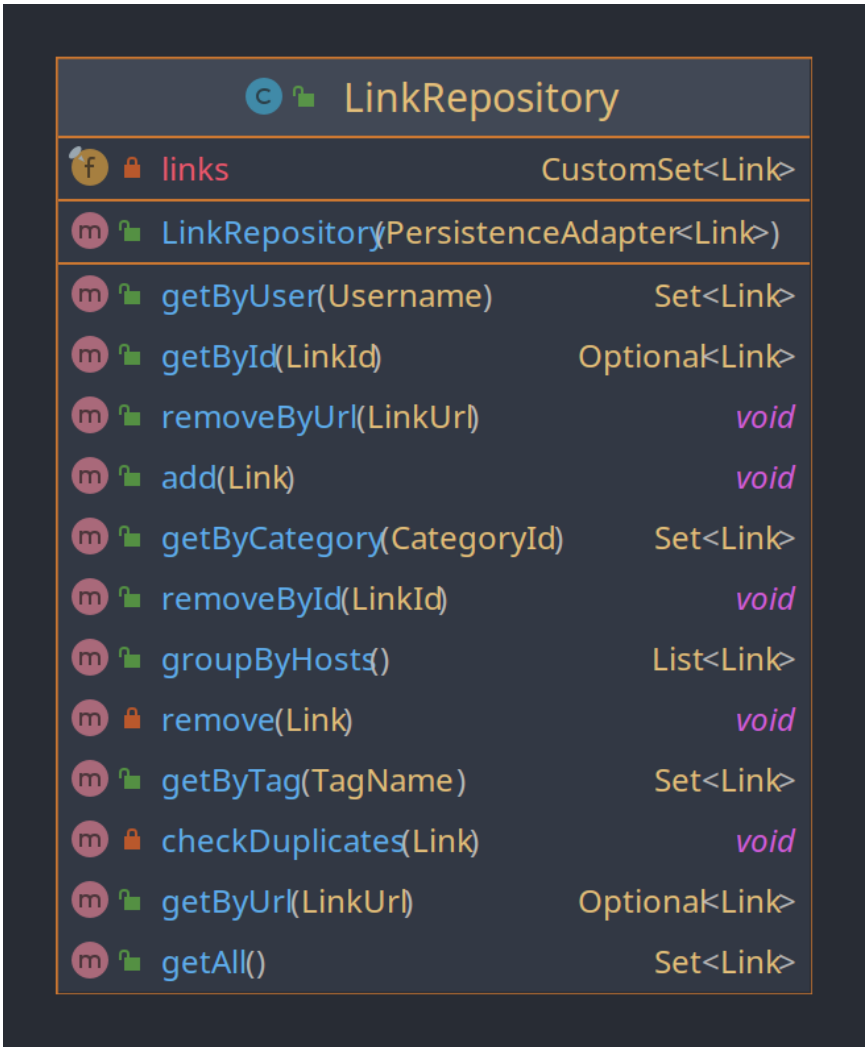
Die Klasse GenericCSVDAO ist für das persitieren von Entities gedacht. Sie besitzt aber nur eine geringe Kopplung zu den Entities, die sie persitieren soll. Mit ihr kann quasi jede Klasse persitiert werden, die das Interface CSVSerializable implementiert (aktuell sind das LinkEntity, CategorEntity, und CustomTagMatcherEntity). Dadurch besteht keine Abhängig der Klasse GenericCSVDAO, zu den Eigenschaften der Entites, sondern nur zur dem Interface. Dies sorgt für eine gute Wiederverwendbarkeit des Codes, da alle Interaktionen mit dem Dateisystem und das Aufteilen der Datei zu einzelnen Objekten generisch für alle zu persistierenden Datentypen implementiert ist. Das Interface GenericDAO sorgt außerdem dafür, dass Klassen die GenericCSVDAO nutzen nicht direkt an diese Implementation gebunden sind, sondern an das Interface. So könnte man beispielsweise die CSV-Implementation schnell und einfach durch eine Datenbank austauschen.

2. Negativ-Beispiel



Die Klasse LinkCliAdapter bildet die Schnittstelle für die Nutzereingaben zu dem LinkUseCase und wandelt dabei die Daten von einfachen Strings zu den Domänenobjekten (ValueObjects) um. Sie ist direkt an den LinkUseCase gebunden und hat somit eine hohe Kopplung. Die Kopplung könnte durch das Einführen eines Interfaces, das die Funktionalität des LinkUseCases beschreibt verringert werden. Dies wurde jedoch nicht gemacht, da die Kopplung als wenig schlimm angesehen wird. Der Code des Usecases sollte langlebiger sein, als der des Adapters. Es ist also deutlich Wahrscheinlicher, dass sich der Adapter ändert. Auch entspricht diese technische Kopplung der fachlichen: werden neue Funktionen im Usecase hinzugefügt sollten sie auch im Adapter ergänzt werden, damit der Anwender sie benutzen kann.

4.0.2 Analyse GRASP: Hohe Kohäsion



Die Klasse LinkRepository hat eine hohe Kohäsion, da alle vorhanden Funktionen mit dem einzigen Attribut "links" arbeiten, somit ist der semantische Zusammenhang zwischen den Methoden und den Daten hoch.

4.0.3 Don't Repeat Yourself (DRY)

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.

5 Kapitel 5: Unit Tests

5.0.1 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.

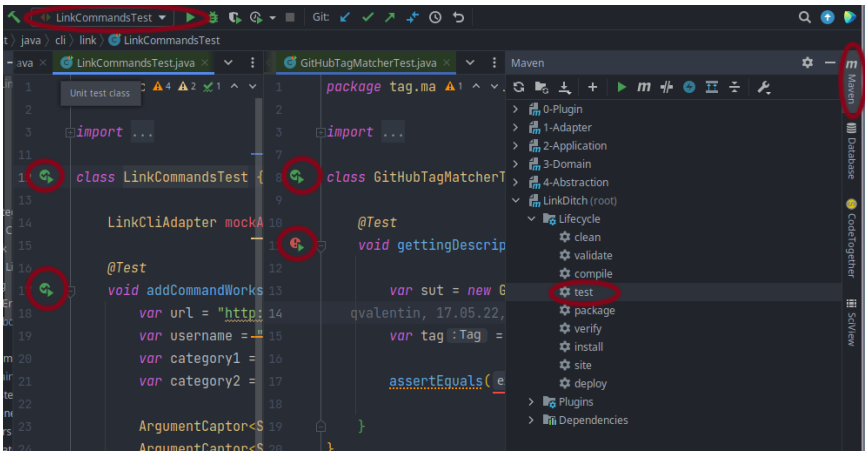
10. GitHubTagMatcherTest#gettingDescriptionWorks Überprüft die Interaktion mit der GitHub Repository API indem für ein konkretes Repository der Wert abgefragt wird.

5.0.2 ATRIP: Automatic

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

```
mvn clean test
```

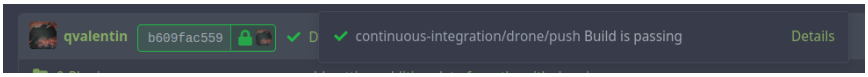
Alternativ genügen auch wenige Clicks bzw. Shortcuts in der IDE um die Test auszuführen und detailreiches Feedback über ihren Erfolg zu erhalten.



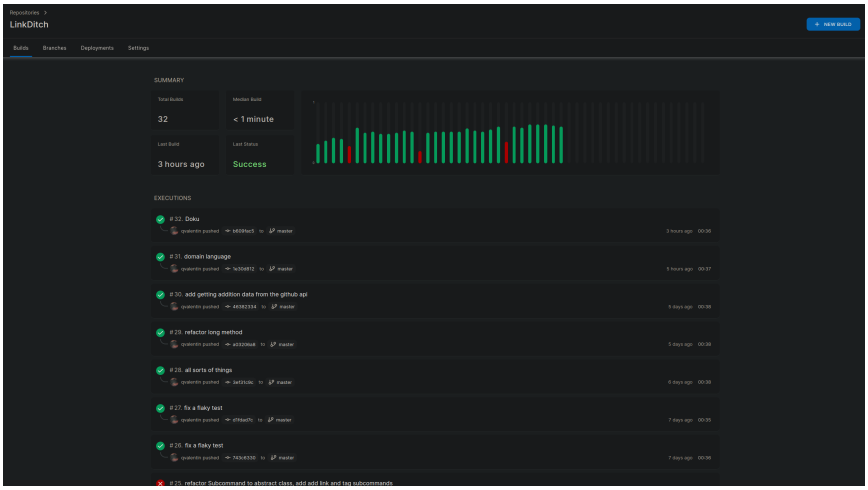
Die Test laufen ohne Eingaben und liefern immer ein Ergebnis, dass eindeutig Erfolg oder Misserfolg bezeugt.

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.



5.0.3 ATRIP: Thorough

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, constructorThrowsEmpty, constructorThrowsTooShort)

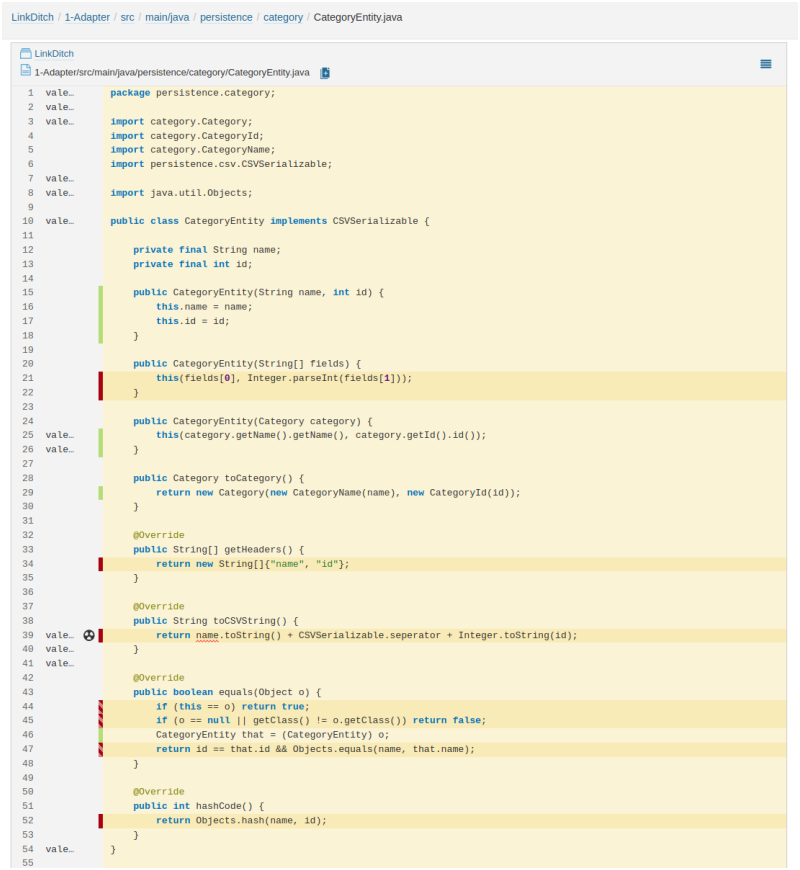
Coverage **53.3%**

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).



5.0.4 ATRIP: Professional

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.

```java
@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.class);
        ArgumentCaptor<Set<String>> captureCategories =
        ArgumentCaptor.forClass(Set.class);

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

        var sut = new LinkCommands(mockAdapter);
        var returnValue = sut.executeSubcommand(new String[]{"add",
                                    url, username, category1, category2});

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

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

5.0.5 Zusatz: ATRIP: Repeatable

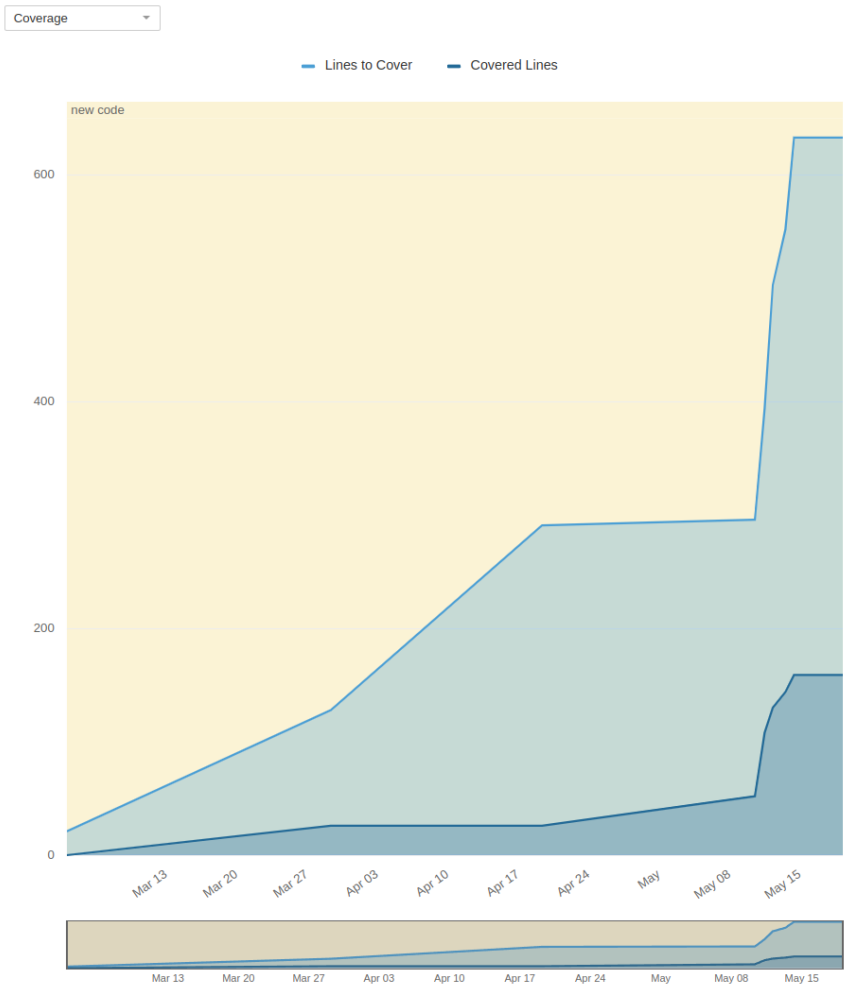
Der Commit d1fdad7cf9 zeigt den Fix für einen Test der nicht repeatable war, weil bei Sets die Reihenfolge der Elemente nicht eindeutig ist und somit zufällig.

5.0.6 Code Coverage

Die Coverage ist mit etwa 22 % deutlich zu niedrig. Da es sich bei dem Projekt jedoch um Code handelt, der niemals wirklich produktiv eingesetzt wer-

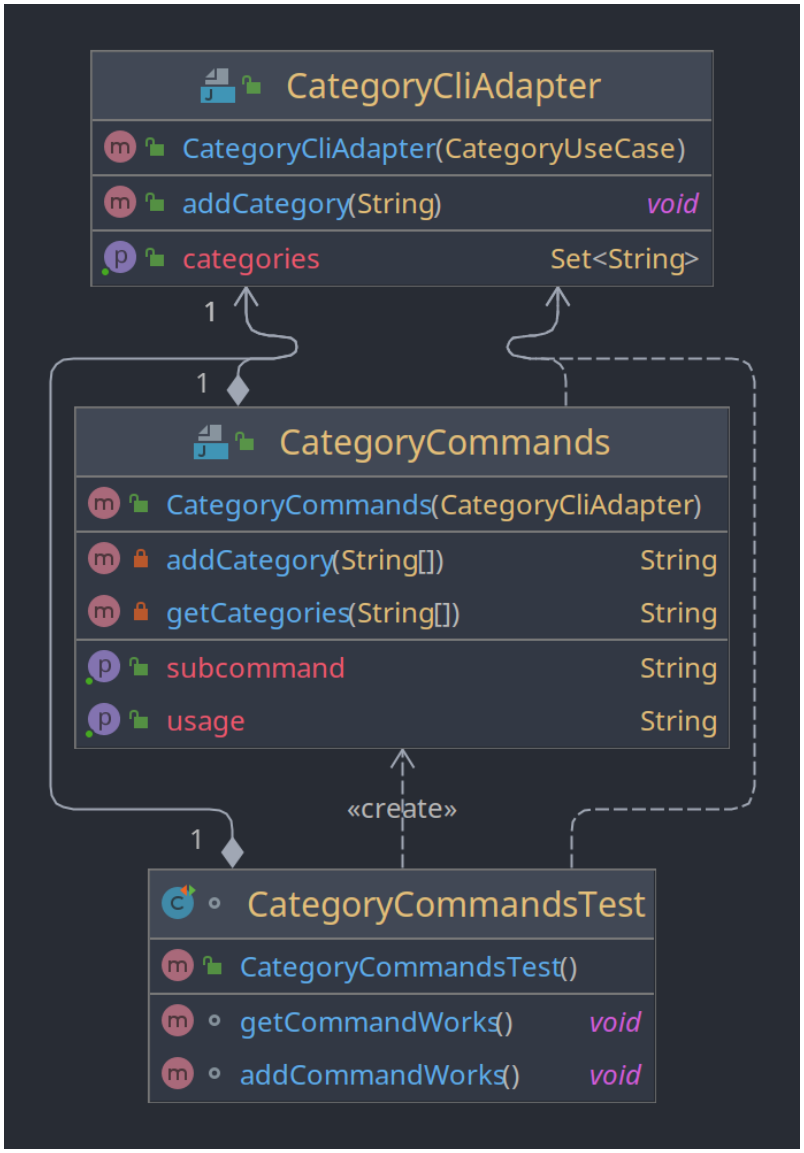
den wird und der nicht langfristig weiterentwickelt wird, ist dies verkraftbar. Es wurden hauptsächlich die komplizierteren Stellen getestet, wie beispielsweise die CSV-Persistierung und die Interaktion mit der Github-APi. Bei diesen Stellen wurden grade genug Test geschrieben, um sicherzustellen, dass 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.

Zur Anaylse des Codes wird Sonarqube genutz: `https://sonar.filefighter.de/dashboard?id=de.qvalentin%3ALinkDitch`



5.0.7 Fakes und Mocks

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.

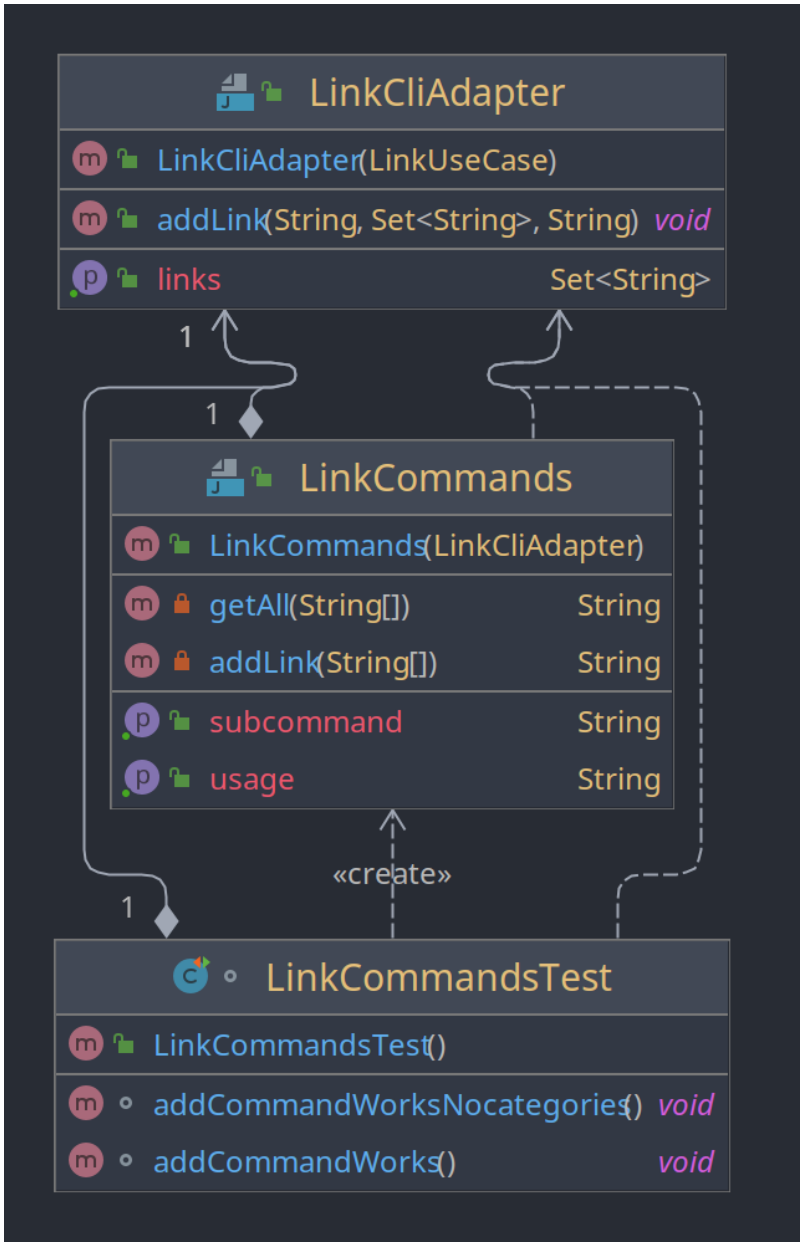
```
class CategoryCommandsTest {

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

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

        var returnValue = sut.executeSubcommand(
                                    new String[]{"add", categoryName});

        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", "workStuff"));
        var returnValue = sut.executeSubcommand(new String[]{"get"});

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

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

Das Mock ist hier besonders nützlich, da wir uns an der Grenze zwischen zwei Schichten befinden. Für das Erstellen des CategoryCliAdapters 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 Funk-

tionalität 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.

1. 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.

6 Kapitel 6: Domain Driven Design

6.0.1 Ubiquitous Language

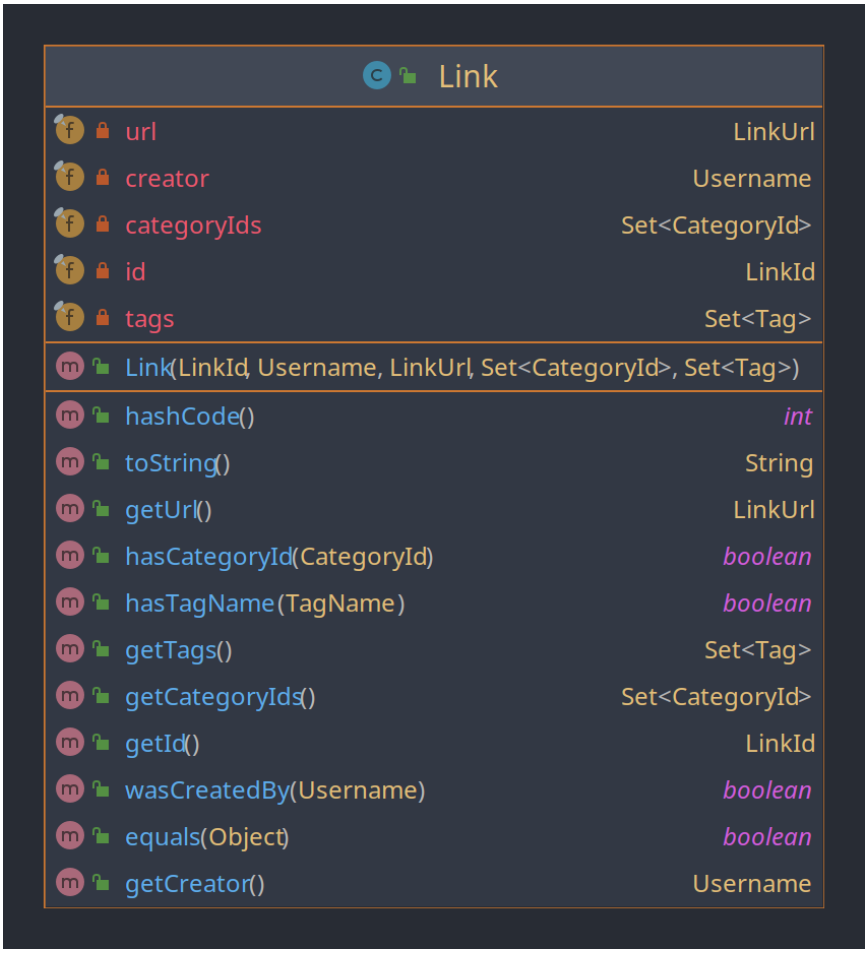
1. Link **Bedeutung**: Steht für eine eindeutige URL, die von der Anwendung gespeichert werden soll **Begründung**: Gehört zur Ubiquitous-Laguage, weil Link außerhalb der Domäne der Anwendung auch anders verwendet werden kann

2. Category **Bedeutung**: Steht für eine Kategorie, die einem Link zugeordnet werden kann **Begründung**: Gehört zur Ubiquitous-Laguage, weil nur im Kontext der Anwendung klar ist, wofür die Kategorien verwendet werden

3. Tag **Bedeutung**: Steht für einen Tag der automatisch bestimmten Link-Typen zugeordnet wird **Begründung**: Gehört zur Ubiquitous-Laguage, weil nur im Kontext der Anwendung klar ist, was genau getaggt wird und wie dies geschieht

4. *TagMatcher **Bedeutung**: Steht für eine spezielle Implementation des Interfaces TagMatcher (z.B. GitHubTagMatcher), das dafür sorgt, dass einem Link ein Tag zugewiesen werden kann **Begründung**: Gehört zur Ubiquitous-Laguage, weil nur im Kontext der Anwendung klar ist, auf welche Tag sich die Funktion bezieht und was deren Bedeutung ist.

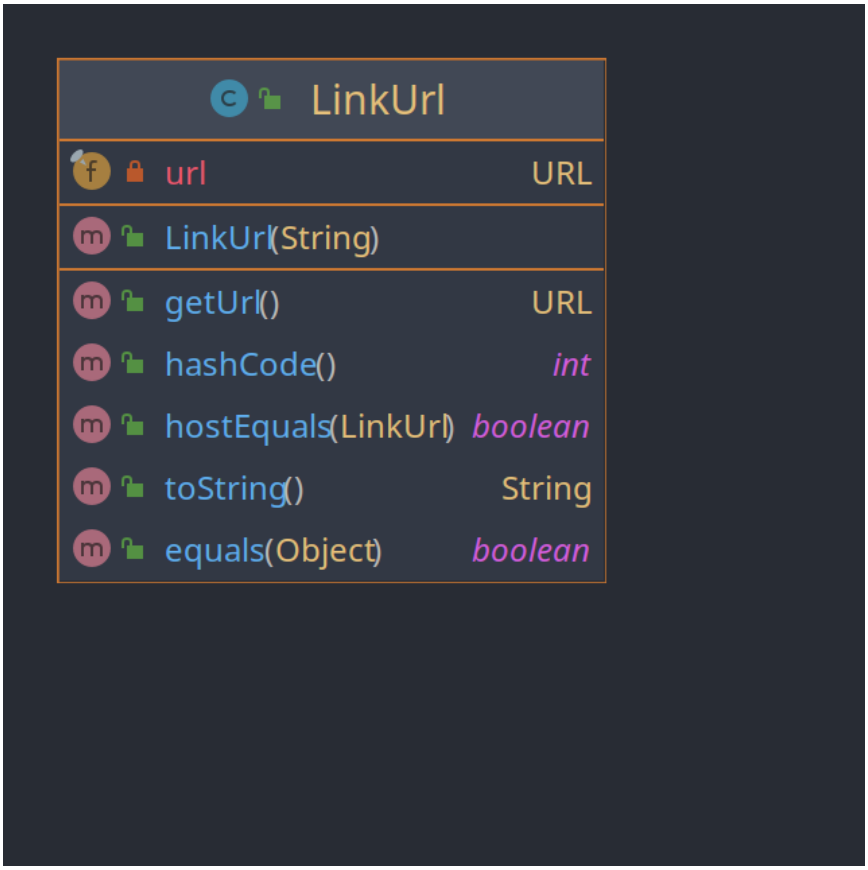
6.0.2 Entities



Die Klasse Link ist ein Entity, da sie eindeutig über ihre Id LinkId identifizierbar ist und andere Eingenschaften hat, die nicht identifizierend sind. Man hätte für die Id letztlich auch die URL verwenden können, wenn man verhindern wollte, dass zwei Links mit gleicher URL gespeichert werden, es wurde sich aber gegen diesen "natürlichen" Schlüssel entschieden. Stattdessen werden die Schlüssel zufällig generiert. Links werden genutzt um URL und dazugehörige Informationen für die Organisation, wie beispielsweise die Verknüpfung zu Kategorien und Tags zu speichern. Auch wenn es bisher nicht implementiert ist, könnten Links verändert werden (indem beispielsweise eine neue Kategorie hinzugefügt wird), weshalb ein Entity geeignet ist. Mit den Funktionen wasCreatedBy,hasTagName,hasCategoryId

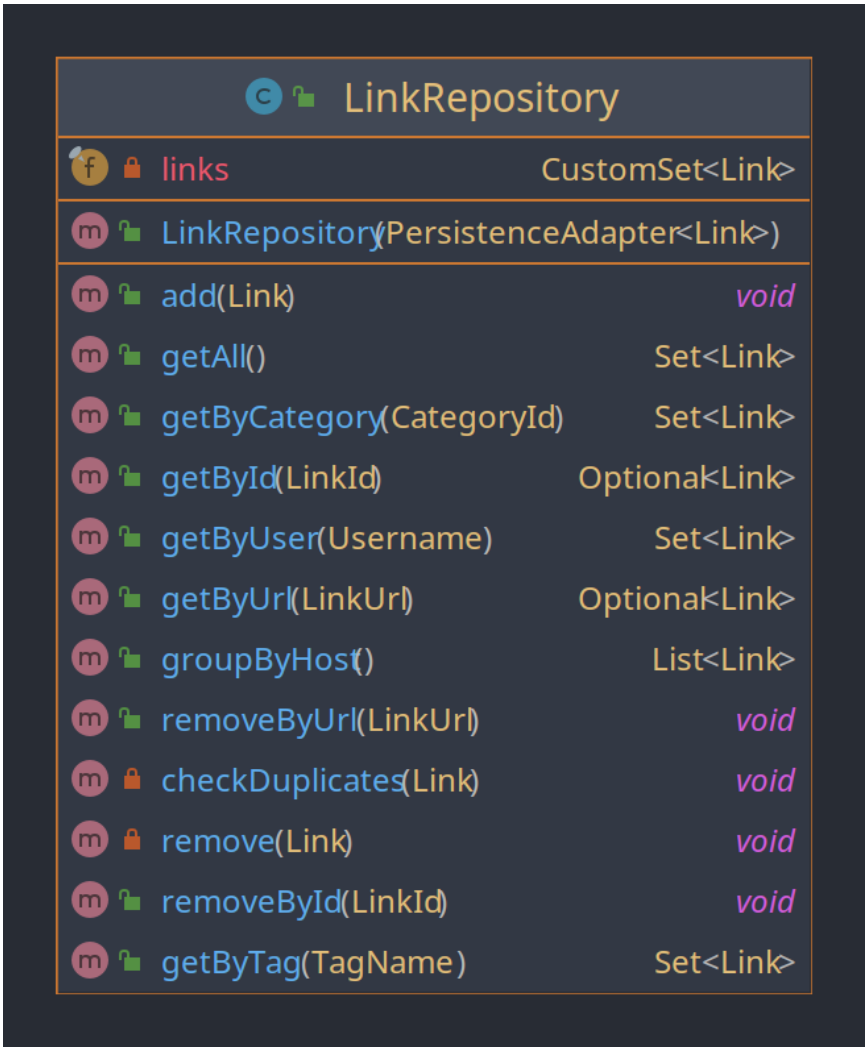
wird Verhalten direkt im Entity beschrieben. Die Getter sind hauptsächlich für die Konvertierung zum Ausgabe- bzw. Persistenzformat.

6.0.3 Value Objects



Die Klasse LinkUrl ist ein ValueObject, dass den Wert einer URl repräsentiert. Um die Domänenregeln zu überprüfen wird die Java Klasse URL verwendet. Die Gleichheit zweier LinkUrls ergibt sich, daraus, ob beide dem selben String entsprechen. Wenn die Domäne dies so will könnte man beispielsweise bei Vergleich auch das Protokoll vernachlässigen. Mit der Methode hostEquals können beispielsweise zwei Urls verglichen werden, ob sie den gleichen Host haben, dies wird dazu genutzt, die Links nach Host gruppieren zu können.

6.0.4 Repositories



Die Klasse LinkRepository ist ein Repository. Sie bietet Zugriff auf die zur Laufzeit des Programmes im Arbeitsspeicher gehaltenen Link Objekte und ist gleichzeitig die Schnittstelle (ein Adapter ist noch dazwischen) zum persistenten CSV-Datei Repository (Klasse: GenericCSVDAO). Alle Domänenspezifischen Abfragen an die Daten (getById, getByUser, getByUrl ...) und Veränderungen werden in diesem Repository durchgeführt. Das CSV-Datei Repository hat dagegen nur reine Create, Read, (Update) und Delete Funktionalität anhand des Schlüssels einer Entity. Dass das Programm

sämtliche Daten im Arbeitsspeicher hält ist eine diskutable Designentscheidung. Aufgrund der eher geringen zu erwartenden Datenmengen sollte es keine Probleme mit benötigtem Arbeitsspeicher geben. Die Vorteile sind, dass in der Domäne direkt auf den Objekten gearbeitet werden kann und nicht erst eine Anfrage an die Pluginschicht gestellt werden muss. Auch kann die Domänenspezifische Abfragelogik in der Domäne implementiert werden und die Persitenz-Repositories haben nur eine reine CRUD Funktionalität.

6.0.5 Aggregates

Bei der Implementierung gibt es keine Klasse, die ein Aggregate widerspiegelt, da jedes Aggregat aus genau einem Entity besteht und somit eine reine Wrapper-Klasse unnötig ist. So verwaltet das CategoryRepository genau das Entity Category und das LinkRepository das Entity Link. Die Assoziationen von Links zu Categorys erfolgt indirekt über die IDs der Catgegorys und nicht über direkte Objektreferenzen, was quasi dem Ansatz von Aggregates entspricht. Das TagMatcherRepository verwaltet mit den CustomTags auch genau ein Entity und hält zusätzlich noch zur Laufzeit unveränderte Statische TagMatcher.

Grundsätzlich könnten Aggregates eingesetzt werden, aufgrund der geringen Komplexität der Daten würden sie die Implementierung aber vermutlich nur unnötig verkomplizieren.

7 Kapitel 7: Refactoring

7.0.1 Code Smells

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.

```java
// 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.

2. Long Method Die Funktion im LinkUseCase, welche alle Links aus dem Repository ausliest und mit den richtigen Kategorienamen anreichert war recht lang und nicht gerade einfach zu verstehen. Der Name der Methode vermittelte auch nicht wirklich, dass die Daten noch angereichert werden. Hier der alte Code:

```
public Set<LinkDto> getLinks() {
        return linkRepository
                        .getAll()
                        .stream()
                        .map(link ->
                new LinkDto(link.getCreator(),
                        link.getUrl(),
                        link
                        .getCategoryIds()
```

```
                        .stream()
                        .map(categoryRepository::getById)
                        .map(optional -> optional.orElseThrow(() ->
                                        new CategroyDoesNotExist(
                            "A Category for a certain id does not exits.
                            You must create it first.")))
                        .collect(Collectors.toSet()),
                        link.getTags()))
                        .collect(Collectors.toSet());
}
```

Das Aufteilen in kleiner Funktionen machte den Code besser lesbar und die zusätzlich eingefügten Funktionsnamen machen deutlicher, was genau der Code macht. Es wurde Extract-Method genutzt.

```
public Set<LinkDto> getLinks() {
        return linkRepository.getAll()
        .stream().map(convertLink())
        .collect(Collectors.toSet());
}

private Function<Link, LinkDto> convertLink() {
        return link -> new LinkDto(
                        link.getCreator(),
                        link.getUrl(),
                        getCategoriesOf(link),
                        link.getTags());
}

private Set<Category> getCategoriesOf(Link link) {
        return link.getCategoryIds()
        .stream().map(getCategoryForId())
        .collect(Collectors.toSet());
}

private Function<CategoryId, Category> getCategoryForId() {
        return id -> categoryRepository
                        .getById(id)
                        .orElseThrow(() ->
                        new CategroyDoesNotExist(
```

```
                                                    "A Category for a certain id does not exit
                        You must create it first."));
}
```

Die Änderung wurde mit Commit a03206a8e1 durchgeführt.

7.0.2 2 Refactorings

1. Replace Error Code with Exception Der Code zum überprüfen, ob eine URL auf eine aktuell erreichbare Ressource zeigt gab im Falle einer IO Exception den ErrorCode 500 (anhand des HTTP Status Codes) zurück. Dieser wurde dann von der aufrufenden Methode interpretiert und ein Fehler geworfen. Somit konnte ein Internal Server Error der angefragten Ressource nicht von einem lokalen Fehler, wie fehlender Netzwerkverbindung oder einem DNS-Fehler unterschieden werden. Deutlich besser ist es, direkt eine passende Exeption zu werfen und dabei die Informationen an den Nutzer weiterzuleiten, sowie die möglichen IOExceptions zu unterscheiden und einen Sonderfall für einen DNS Error, der recht wahrscheinlich ist einzuführen.

   Alter Code:

```
private static int getResponseCode(LinkUrl url) {
        try {
                HttpURLConnection http = (HttpURLConnection) url.getUrl()
            .openConnection();
                http.setRequestMethod("HEAD");
                http.disconnect();

                return http.getResponseCode();
        }
        catch (IOException e) {
        return 500; //TODO: seems smelly
        }
}
```

   Alter Code:

```
private static int getResponseCode(LinkUrl url) {
        try {
                HttpURLConnection http = (HttpURLConnection) url.getUrl()
            .openConnection();
```

```
                http.setRequestMethod("HEAD");
                http.disconnect();

                return http.getResponseCode();
            }
        catch (UnknownHostException unknownHostException){
                        throw new URLIsNotReachable("The host of the url " + url +
            }
            catch (IOException e) {
                throw new URLIsNotReachable(
                 "Something went wrong when trying to check if the url " +
                 url +
                 " is reachable. Make sure your internet connection is working: "
                 + e.getMessage());
            }
    }
```

Das Refactoring wurde mit Commit 7fbc3f722c durchgeführt.

2. Rename Method

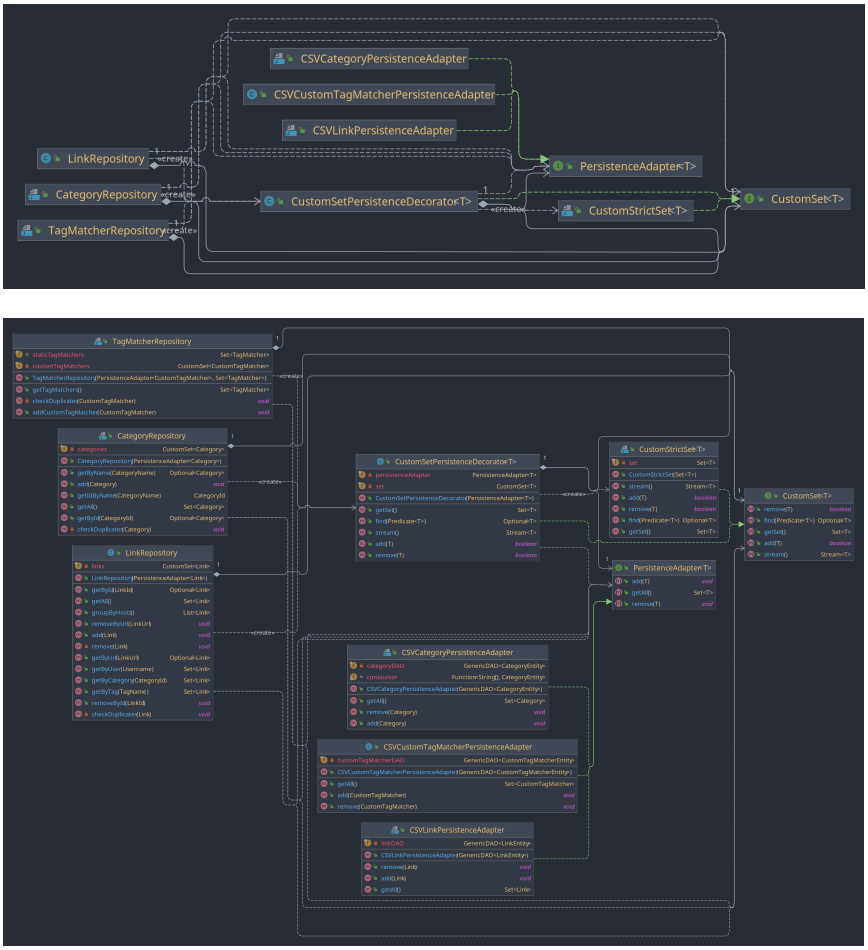
   In den Klassen LinkCliAdapter, LinkCommands und LinkUseCase gibt es jeweils eine Funktion groupByHosts, die die nächste Schicht aufruft, bis im Repository die Daten entsprechend angeordnet werden. Nur in der Klasse LinkRepository hieß diese Methode groupByHost ohne "s" am Ende. Diese Inkonsistenz ist schlecht und verwirrt beim Lesen des Codes.

   Das Refactoring wurde mit Commit ba6a889d35 durchgeführt.

8 Kapitel 8: Entwurfsmuster

8.0.1 Entwurfsmuster: Dekorator

Für die Synchronisation der Datenhaltung im Arbeitsspeicher und der Persistenz in Form von CSV-Dateien wird das Dekorator Entwurfsmuster verwendet.



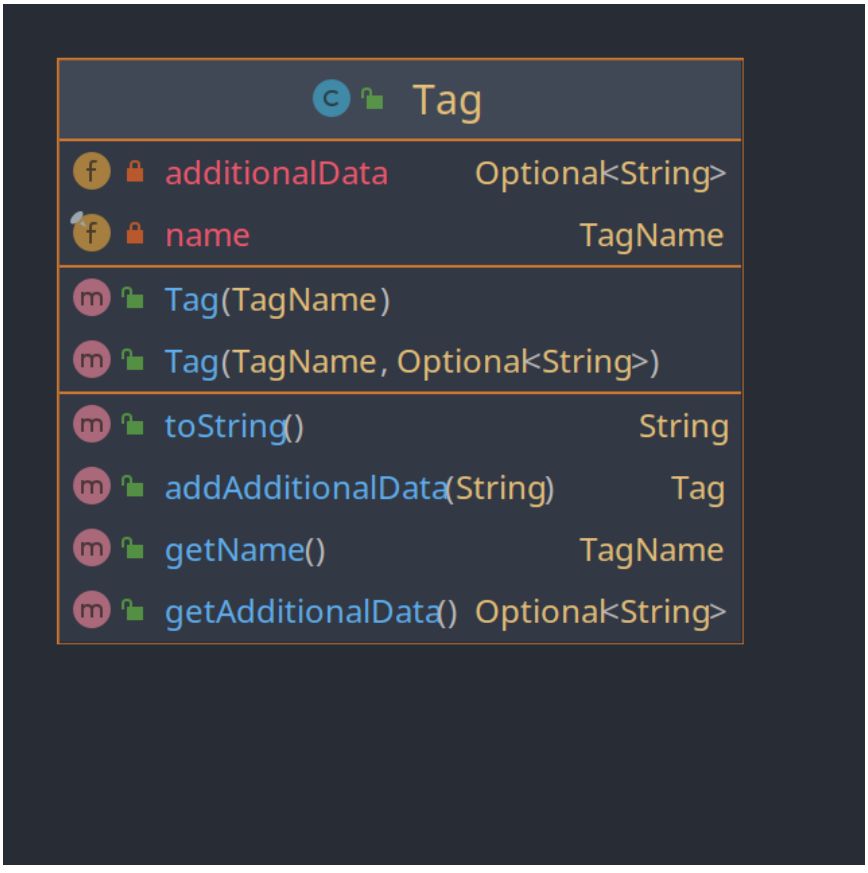
Zum Vergrößern des Bildes siehe Datei: ./uml/CustomSetPersistenceDecoratorClasses.png

Jedes der Repositories (LinkRepository, CategoryRepository und TagMatcherRepository) hält die Daten in einem Set. Doch statt hierfür ein normales Set zu verwenden wird eine Implementation des Interfaces CustomSet verwendet. Für dieses Interface gibt es zwei Implementationen. Das CustomStrictSet entspricht quasi einem normalen Set, nur dass es Exeptions wirft, wenn ein Element hinzugefügt wird, das bereits existiert. Die andere Implementation CustomSetPersistenceDecorator bildet den Dekorator für das CustomSet. Dieser schaltet sich vor ein CustomStrictSet und fängt sämtliche Aktionen ab, welche den Zustand der Daten verändern. Diese Aktionen werden dann sowohl in der Datenstruktur im Arbeitsspeicher durchgeführt als auch über das PersistenceAdapter an die Persistenzschicht weitergeleitet und

dort verarbeitet. Hierfür erhält der Dekorator ein PersistenceAdapter<T>, wobei T dem Datentyp entspricht, der durch das Repository verwaltet wird.

Durch den Einsatz des Entwurfsmusters wird sichergestellt, dass eine Veränderung der Daten im Arbeitsspeicher auch immer in die CSV-Dateien persistiert wird. Sämtliche Repository-Methoden, die den Zustand ändern müssen nicht mehr das PersistenceAdapter aufrufen sondern dies geschieht automatisch. Dadurch wird der Code einfacher lesbar und Fehler können vermieden werden. Diese Ansatz orientiert sich an Aspectorientierter Programmierung, hier wird des Aspect der Persistenz vom Repository losgelöst und durch den Dekorator geregelt.

8.0.2 Entwurfsmuster: Builder (Erbauer)



Das Builder Entwurfsmuster wird bei der Klasse Tag für den Optionalen Wert additionalData genutzt. Die Funktion addAdditionalData fügt den

optionalen Wert hinzu und gibt das veränderte Objekt zurück.