Programmentwurf IssueTracker

Im Rahmen der Vorlesung
Advanced Software Engineering

Verfasser:	Denis Thiessen
Kurs:	TINF19B4
Abgabedatum:	16. Mai 2022
Matrikelnummer:	6993015

Inhaltsverzeichnis

Abbildungsverzeichnis IV

1 Domain-Driven-Design 1
1.1 Ubiquitous-Language . . . 1
1.2 Analyse und Begründung der verwendeten Muster . . . 2
1.2.1 Value-Objects . . . 2
1.2.2 Entities . . . 3
1.2.3 Aggregates . . . 3
1.2.4 Repositories . . . 4

2 Clean-Architecture 5
2.1 Domain-Schicht . . . 5
2.2 Application-Schicht . . . 6
2.3 Adapter-Schicht . . . 6
2.4 Plugin-Schicht . . . 6

3 Programming Principles 7
3.1 SOLID . . . 7
3.1.1 Single-Reposibility Principle . . . 7
3.1.2 Open/Close Principle . . . 8
3.1.3 Liskov-Substitution-Principle . . . 8
3.1.4 Interface-Segregation Principle . . . 8
3.1.5 Dependency-Inversion Principle . . . 9
3.2 GRASP . . . 9
3.2.1 Controller . . . 9
3.2.2 Kohäsion . . . 9
3.2.3 Kopplung . . . 10
3.2.4 Information Expert . . . 10
3.2.5 Polymorphie . . . 10
3.2.6 Pure Fabrication . . . 11
3.2.7 Delegation/Indirection . . . 11
3.2.8 Protected Variations . . . 11
3.2.9 Creator . . . 12

3.3 DRY 12

4 Refactoring **13**

4.1 Code-Refactoring 1 | Extract Method 13

4.2 Code-Refactoring 2 | Duplicated Code 14

5 Unit-Tests **16**

5.0.1 ATRIP-Regeln 16

5.0.2 Verwendung von Mocks 17

6 Entwurfsmuster **18**

6.1 Analyse des verwendeten Entwurfsmusters 18

6.1.1 Observer-Pattern 18

6.1.2 Anwendung im Projekt 19

Literatur **V**

Abbildungsverzeichnis

1 Ausschnitt der Initialisierungsmethode der ProjectView 13
2 ProjectView Initialisierungsmethode nach Refactoring 14
3 Ausschnitt des duplizierten Codes 15
4 Referenzierung mittels Hilfsklasse 15
5 Beispiel UML-Diagramm zum Observer-Pattern [1] 19
6 UML-Diagramm des Observer-Musters (grün: vorher) 20

Kapitel 1

Domain-Driven-Design

1.1 Ubiquitous-Language

Ubiquitous-Language bezeichnet innerhalb des Kontextes der Softwareentwicklung die Sprache mit ihren Fachbegriffen, welche sowohl von den Softwareentwicklern als auch von den Domänenexperten gesprochen wird, um sich über bestimmte Konzepte innerhalb der Domäne zu verständigen.

Innerhalb des Issuetrackers existieren nicht besonders viele solcher Begriffe, da die Domäne in diesem Gebiet keine hohe Komplexität vorweist.

Die Hauptkomponente stellt hierbei die sogenannte *Issue* (dt. Problem) dar. Eine Issue stellt ein Problem jeglicher Art eines *Nutzers* dar. Dabei kann dieses Problem grundsätzlich ein Titel, eine Kurzbeschreibung und den *Ausführungsstatus* beziehungsweise die *Ausführungsdringlichkeit* während der Lösung des Problems enthalten.

Ein Ausführungsstatus und eine Ausführungsdringlichkeit beschreiben jeweils Issueeigenschaften, welche aufzeigen, in welcher Phase der Problemlösung sich das Issue befindet und wie dringend die Lösung dieses Problems ist.

Diese Issues können schließlich zu sogenannten *Projekten* gruppiert werden, welche einen ähnlichen Kontext oder sonstige Abhängigkeiten besitzen.

Nutzer innerhalb dieser Issue-Tracker Plattform können nach Belieben neue Issues erstellen und entsprechend kategorisieren. Innerhalb von Projekten kann eine Liste von Nutzern verwaltet werden, durch Projektmodifikation und *Projekteinladungen*, welche auf das jeweilige Projekt zugreifen kann. Einladungen werden primär dazu verwendet, um neue projektfremde Nutzer zu einem Projekt hinzuzufügen und ihnen die Möglichkeit zu liefern, einem neuen Projekt beizutreten. Projektnutzer sind Teil eines Projektes und bearbeiten schließlich zusammen als Gruppe die Menge an Issues innerhalb eines Projektes.

1.2 Analyse und Begründung der verwendeten Muster

1.2.1 Value-Objects

Unter einem *Value-Object* versteht man unveränderliches Objekt, welches einen gewissen Wert innerhalb der Domäne repräsentiert. Innerhalb dieses Projektes werden diese Value-Objects dafür verwendet, um für Entitäten jeglicher Art gewisse Werteinheiten festzulegen, welche nach Belieben ausgetauscht werden können.

Value-Objects innerhalb des Anwendungskontextes des Issue-Trackers sind dabei folgende Objekte.

- Issuestatus
- Issuedringlichkeiten
- Projekteinladungen

All diese Objekte haben die Gemeinsamkeit, dass sie nach der Initialisierung einen festen Wert innerhalb der Domäne darstellen. Bei den Issuestatus sind das eine Menge von Status (In Vorbereitung in Durchführung erledigt usw.) sein, welche eine Issue annehmen kann und bei Änderung immer wieder ersetzt werden. Bei den Issuedringlichkeiten ist es ähnlich zu den Issuestatus, da auch hier eine Menge von Dringlichkeiten existiert, welche eine Issue annehmen kann und immer wieder schließlich ersetzt werden muss, allerdings die eigentlichen Dringlichkeiten nie ändern.

Bei den Projekteinladungen handelt es sich um ein Objekt, worin schlichtweg festgelegt wird, welcher Nutzer eine Einladung zu einem gewissen Objekt erhalten hat. Dabei sollen die Einladungen nicht weiter modifizierbar sein und bei neuen Projekteinladungen für einen Nutzer wird in aller Regel eine neue Projekteinladung erstellt.

1.2.2 Entities

Entities besitzen im Gegensatz zu Value-Objects den Unterschied, dass diese einen eindeutigen Lebenszyklus besitzen und sich somit im Gegensatz zu Value-Objects darin auch verändern können.

Innerhalb der Applikation existieren folgende Entity-Objekte.

- Nutzer
- Issuedetails
- Projektdetails

Innerhalb eines Lebenszyklus können sich Nutzerdetails wie Nutzernamen ändern, genauso wie sich in der Realität auch Namen ändern können. Deswegen lassen sich die Nutzerobjekte auch als Entities einordnen. Sowohl Issue als auch Projektdetails enthalten Informationen zu den jeweiligen Objekten wie ein Titel, eine Beschreibung oder auch Referenzen zu Nutzern, welche auch während der Laufzeit verändert werden müssen. Deswegen lassen sich auch diese Objekte jeweils als Entities einordnen.

1.2.3 Aggregates

Aggregates haben die Aufgabe, eine Menge an Value-Objects und Entities miteinander zu gruppieren, zu einer gemeinsam verwalteten Einheit, um so die Komplexität dieser gesamten Verwaltungseinheit zu verringern.

Dabei stellen folgende Objekte innerhalb des Issue-Trackers Aggegates dar.

- Issue
- Project

Der Grund hierbei ist, dass diese Objekte beiderlei Value-Objects (Issue-State, Issue-Urgency) und Entities (Project-Details, Issue-Details) enthalten und diese wie bereits angesprochen über diesen Aggregate-Mechanismus verwalten.

1.2.4 Repositories

Repositories stellen hierbei ein Verbindungsglied zwischen der Domäne und dem Datenmodell dar und bieten Methoden bereit, um konkrete Aggregate aus dem Datenspeicher zu lesen oder auch zu verarbeiten, wobei diese Repositories vor der konkreten Implementierung verborgen bleiben.

Im Folgenden lassen sich drei verschiedene Repositories identifizieren.

- Nutzerrepository
- Issuerepository
- Projektrepository

Diese Repositories bestehen aus Interfaces, welche Methodenhüllen zur Verarbeitung der jeweiligen Objekte enthalten sowie einige zusätzliche Hilfsmethoden enthalten, wie zum Beispiel eine Methode zum Generieren einer Liste von Projekten, worin ein Nutzer ein Mitglied ist. Mithilfe von Hibernate wird somit eine Persistierungsschicht geschaffen, womit die entsprechenden Daten persistiert werden können, welche schließlich mit diesen Methoden aufgerufen und die dahinterstehenden Operationen durchgeführt werden.

Kapitel 2

Clean-Architecture

Die Clean-Architecture innerhalb des Issue-Tracker Projekts ist in vier unterschiedliche Ebenen in folgender Reihenfolge gegliedert.

- Die Plugin-Schicht
- Die Adapter-Schicht
- Die Application-Schicht
- Die Domain-Schicht

2.1 Domain-Schicht

Die Domain-Schicht enthält alle fachlichen Regeln, welche innerhalb des gesamten Projektes beziehungsweise der Organisation gelten sollen.

In dem Issue-Tracker Projekt sind das alle Objekte, welche bereits im Abschnitt Analyse und Begründung der verwendeten Muster angesprochen wurden, im Form von entsprechenden Klassen enthalten und passend mit Hibernate-Annotationen versehen. An dieser Stelle wäre nach Clean-Architecture Regeln hierbei eine Trennung zwischen diesen Klassen und den Hibernate-Annotationen notwendig, da solche Frameworks in die Plugin-Schicht ausgelagert werden müssten.

Da es sich allerdings hier um Annotationen, welche keinen Einfluss auf die eigentliche Business-Logik hinter der Domain-Schicht haben und ein solcher Schritt eine höhere Komplexität für wenig Mehrwert bieten würde, wurde an dieser Stelle davon abgesehen. Eine Möglichkeit, um dieses Problem zu lösen, wäre es an dieser Stelle Interfaces oder abstrakte Klassen zu definieren, welche schließlich in der Pluginschicht mit den jeweiligen Annotationen vervollständigt werden, um eine solche Trennung herbeizuführen.

2.2 Application-Schicht

Die Application-Schicht enthält Anwendungsfälle, welche direkt aus den Projektanforderungen entstehen. Dadurch entsteht schließlich die anwendungsspezifische Geschäftslogik, womit die Steuerung der Domain-Elemente bewerkstelligt wird.

Innerhalb des Projekts wurden so einfache *CRUD*-Operationen zu den einzelnen Objekten umgesetzt sowie Hilfsoperationen, wie zum Beispiel eine Funktion zum Finden von allen Projekten, worin ein bestimmter Nutzer Mitglied ist.

2.3 Adapter-Schicht

Die Adapter-Schicht vermittelt Daten zwischen den inneren Applikations und Domänenschichten sowie der Pluginschicht. Hierbei finden oftmals Formatkonvertierungen zwischen externen und internen Formaten/Objekten statt.

In diesem Projekt finden hauptsächlich Formatkonvertierungen zwischen sogenannten *DTO* oder *Date-Transfer-Objects* und den internen Objekten statt. Diese Formatkonvertierungen finden in eigenen objektspezifischen Klassen, welche jeweils entweder das interne Objekt in das DTO umwandeln oder umgekehrt.

2.4 Plugin-Schicht

Die Plugin-Schicht greift schließlich zuletzt auf die jeweiligen Adapter und dessen Interfaces zu, um mit den vorhandenen Frameworks, welche in dieser Schicht beheimatet sind, die konkreten Operationen innerhalb des Projektes zu implementieren. Das Verwenden von Frameworks soll dabei schließlich den Programmieraufwand mindern und durch eine solche Abkapslung können verwendete Frameworks bei benötigten Änderungen schnell ausgetauscht werden.

Innerhalb des Projekts werden hierbei folgende Frameworks verwendet.

- Die H2-Database Engine, zum Speichern der Objektdaten.
- Das Hibernate Persistenz-Framework, zum Persistieren der Domänenobjekte mit den Hibernate-Annotationen in die Datenbank.
- Das Vaadin-Framework zum Gestalten des Web-Frontends mithilfe von Java-Programmcode.

Das Frontend der Anwendung ist hierbei komplett in der Plugin-Schicht vorzufinden, da in jedem darin enthaltenen Objekt Vaadin-spezifische Komponenten enthalten sind und somit von spezifischen Frameworks abhängig sind.

Kapitel 3

Programming Principles

3.1 SOLID

Das SOLID-Prinzip stellt in diesem Kontext ein Akronym aus folgenden Prinzipien dar.

- Single-Reposibility Principle
- Open/Close Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

3.1.1 Single-Reposibility Principle

Dieses Prinzip bezeichnet, dass Objekte beziehungsweise Klassen eine einzige Verantwortung oder Aufgabe besitzen. Innerhalb des Projektes zeigt es sich gut innerhalb der Mapper-Klassen, welche als einzige Aufgabe haben, ein interneres oder ein DTO-Objekt in das jeweils andere Objekt zu konvertieren.

3.1.2 Open/Close Principle

Das Open-Close Prinzip bezeichnet das Prinzip, Objekte und Klassen so zu gestalten, dass diese offen für Veränderungen sind, jedoch geschlossen für Modifikationen. Der bestehende Code soll möglichst nicht verändert werden, jedoch kann dieser um passende Funktionalitäten erweitert werden, ohne das Fehler in der Applikation entstehen.

Besonders in einer großen Anwendung ist das Prinzip nicht immer einzuhalten, jedoch geht es hierbei prinzipiell darum, eine erhöhte Klassenstabilität herbeizurufen.

Anwendet wird dieses Prinzip beispielsweise innerhalb der Issue View und den Labels, welche automatisiert sich aktualisieren. Da diese Labels zur Aktualisierungsmechanik mittels eines Observers aktualisiert werden, welcher eine Aktualisierung automatisch innerhalb einer gewissen Zeitspanne auslöst, ist es trivial, eine weitere Komponente diesen Observers hinzuzufügen, um eine Komponente mit dieser Aktualisierungsmechanik zu erweitern.

3.1.3 Liskov-Substitution-Principle

Das Liskov-Substitution-Principle bezeichnet, dass eine abgeleitete Klasse jederzeit durch ihre Basisklasse ersetzt werden kann, ohne das es zu unerwünschten Nebeneffekten kommt.

Innerhalb des Vaadin-basierten Frontends des Projekts findet dieses Prinzip Anwendung. Dabei wurden, um Programmcode von UI-Komponenten auszulagern, eigene Klassen für diese gebildet, welche von den Basisklassen abgeleitet werden. Da diese neuen Klassen nur bestimmte Parameter der jeweiligen Komponente definieren und keine weitere Funktionalität im Vergleich zu der Basisklasse einbinden, ist es somit möglich, diese mit den Basisklassen zu ersetzen, ohne unerwünschte Nebeneffekte zu verursachen.

3.1.4 Interface-Segregation Principle

Das Interface-Segregation Principle bezeichnet, dass viele kleinere Interfaces gebildet werden sollen, um innerhalb einer Klasse mehrere Funktionalitäten darzustellen. Dies hat den Vorteil, dass Funktionen variabler, modularer und klarer verteilt werden innerhalb eines Projektes.

Da jedoch nahezu nie der Fall aufkam, dass mehrere Funktionen auf diese Weise vereint werden, mussten (höchstens nur über ein Interface), da die Klassen entsprechend klein aufgebaut sind, konnte dieses Prinzip nur bedingt angewendet werden.

3.1.5 Dependency-Inversion Principle

Das Dependency-Inversion Principle bezeichnet das Prinzip, das Module von Abstraktionen abhängig sein sollen und Abstraktionen nie von Modulen. (High-level modules should not depend on low-level modules.)

In der Praxis ist dieses Prinzip nur schwer aufgrund von Drittcode oder anderen Modulen worin Abhängigkeiten existieren, umzusetzen. Im weitesten Sinn ist dieses Prinzip jedoch bis auf solche Ausnahmen durch die klare Schichtenstruktur implementiert, welche sicherstellt, dass Komponenten aus höheren Schichten nur von Komponenten aus niedrigeren Schichten abhängen und nie umgekehrt.

3.2 GRASP

3.2.1 Controller

Innerhalb des Projektes werden Spring-Controller eingesetzt, um die REST-API Funktionalität bereitzustellen. Diese finden sich in der Plugin-Schicht des Projekts wieder und definieren verschiedene REST-Endpoints, welche teilweise ihren Ursprung in den Repository-Operationen finden.

3.2.2 Kohäsion

Kohäsion bezeichnet ein Maß für den inneren Zusammenhang von Elementen und wie eng diese miteinander zusammenhängen. Ein Beispiel für dieses hohe Kohäsionsmaß innerhalb des Issue-Trackers ist die Verwendung von den JPA-Repositories, welche nur die Kenntnis über die Anbindung an die Datenbasis besitzen und sonst keine weiteren Verbindungen benötigt.

3.2.3 Kopplung

Kopplung bezeichnet den Grad von Abhängigkeiten zwischen zwei oder mehr Objekten. Ein niedriges Maß an Kopplung ist dabei vorteilhaft, da somit Komponenten leichter angepasst und wiederverwendet würde, da hierfür weniger Abhängigkeiten existieren.

Eine niedrige Kopplung ist allerdings nicht immer komplett für ein Projekt umzusetzen. Beispiele aus dem Projekt hierbei sind die verschiedenen Controller für verschiedene Objekte. Manche Controller benötigen keinerlei andere Repositories außer das des jeweiligen Objektes, was auf eine niedrige Kopplung hinweist. Andere Controller benötigen aufgrund von Umwandlungen darin enthaltenen Objekten Zugriff auf weitere Repositories, was auf eine höhere Kopplung hinweist und nicht unbedingt vorteilhaft ist, da mehr Abhängigkeiten herrschen, diese aber notwendig sind für die gegebene Funktionalität.

3.2.4 Information Expert

Das Information Expert Prinzip beschreibt das die Klasse mit den meisten Informationen zu einer Aufgabe auch diese übernimmt. Ein Vorteil für diese Arbeitsweise ist es, dass weniger bis gar keine Informationen an andere Objekte übertragen werden müssen.

Ein Beispiel eines solchen Information Experts innerhalb des Issue-Trackers ist innerhalb einiger UI-Hauptkomponenten vorzufinden. Gerade innerhalb von formularbasierten Komponenten werden in der Hauptkomponente die Werte von Textfeldern oder sonstigen Feldern benötigt, um die jeweiligen Werte an den passenden Service weiterzuleiten. Ein Beispiel hierfür ist die Login-Komponente. Dabei muss der Wert des Nutzernamens aus dem Textfeld der Loginkomponente extrahiert werden und an den entsprechenden Service weitergeleitet werden, um einen Login zu bewerkstelligen.

3.2.5 Polymorphie

Polymorphie bezeichnet in der objektorientierten Programmierung das Konzept verschiedene Variationen und Verhaltenstypen eines Objekts eines gemeinsamen Interfaces repräsentieren zu können. Durch Polymorphie können unvorteilhafte Konstrukte wie längere if-else oder switch-Statements vermieden und Fehler bei Objekterweiterungen vorgebeugt werden.

3.2.6 Pure Fabrication

Pure Fabrication bezeichnet das Konzept von Hilfsklassen, welche so nicht direkt in der jeweiligen Projektdomäne existieren, allerdings dennoch benötigt werden, um das jeweilige Projekt durchzuführen. Ein Vorteil von der Verwendung solcher Hilfsklassen ist es das solche nicht domänenrelevanten Implementierungen getrennt von den domänenrelevanten Teilen und somit auch entsprechend leichter austauschbar sind.

Innerhalb des Projektes werden solche Hilfsklassen hauptsächlich in der Plugin-Ebene verwendet, da an dieser Stelle viel domänenunabhängiger Code existiert. Ein Beispiel hierfür ist in dem Frontend-Code des Logins vorzufinden, worin zuerst eine Überprüfung stattfindet, ob Nutzer mit speziellen Nutzernamen bereits existieren und entsprechende Fehlermeldungen anzeigen. Diese Funktionalität hat prinzipiell nichts mit der jeweiligen Domäne des Issue-Trackers zu tun, wird jedoch benötigt, um keine Fehler bei dem Nutzerlogin des Issue-Trackers zu verursachen.

3.2.7 Delegation/Indirection

Delegation bezeichnet das Prinzip, dass zwei Einheiten über einen Vermittler kommunizieren, anstatt miteinander. Das hat den Vorteil, dass die Interaktionsschnittstelle zwischen diesen beiden Einheiten einheitlich und flexibler ist und nicht direkt abhängig von den jeweiligen Komponenten. Verwendet wird dieses Prinzip, um die grundsätzliche Schichtenarchitektur dieses Projekts umzusetzen, da zwischen den Schichten, gerade in der Adapterschicht Zwischenkomponenten umgesetzt sind, um eine Kommunikation zwischen Objekten aus verschiedenen Schichten umzusetzen.

3.2.8 Protected Variations

Protected Variations bezeichnet das Prinzip innerhalb von Interfaces versteckte konkrete Implementierungen zu verwenden. Somit ist es möglich für bestimmte Objekte Basis-Interfaces zu verwenden, welche eine konkrete Implementierung verbergen, welche einfach austauschbar ist, da das jeweilige Objekt nicht von einer konkreten Implementierung abhängig ist.

Innerhalb des Projekts wird dieses Prinzip hauptsächlich innerhalb des Frontends verwendet, worin häufig gewisse UI-Komponenten in verschiedenen Klassen unterteilt sind, jedoch innerhalb der Hauptkomponente als einfaches Layout oder als Komponente referenziert werden.

3.2.9 Creator

Das Creator beziehungsweise *Erzeuger* Prinzip gibt vor, dass eine Klasse A eine neue Klasse B unter folgenden Regeln nur erzeugen darf.

- A eine Aggregation von B ist oder Objekte von B enthält
- A Objekte von B verarbeitet
- A von B abhängt (starke Kopplung)
- A der Informationsexperte für die Erzeugung von B ist z.B. hält A die Intialisierungsdaten oder ist eine Factory.

Dieses Prinzip findet jedoch keine Anwendung innerhalb des Projektes, da keine konkreten Objekte vorhanden sind, welche aktiv neue, andere Objekte erzeugen.

3.3 DRY

DRY als Prinzip steht hier für *Don't repeat yourself* und bezeichnet das Prinzip, Redundanzen zu vermeiden und möglichst keinen Programmcode zu wiederholen.

Anwendung hat dieses Prinzip dahingehend gefunden, dass an Stellen, worin potenziell mehrfach Code vorkommen könnte, Helferklassen oder andere Objekte erstellt wurden, um diesen Code auszulagern. Ein Beispiel hierfür sind die Mapper-Klassen, welche die internen Objekte in DTO-Objekte umwandeln und umgekehrt.

Diese Programmlogik wurde in eigene Mapper-Klassen ausgelagert und wird bei Bedarf aus den passenden Klassen referenziert, um diese Konvertierungen durchzuführen.

Kapitel 4

Refactoring

4.1 Code-Refactoring 1 | Extract Method

Innerhalb der ProjectView der Initialisierungsmethode für die Frontend-Komponenten war vor dem Refactoring zu unleserlich und zu groß, weswegen ein Bedarf herrschte, Teile dieser Initialisierungsmethode auszulagern, um diese leserlicher zu gestalten.

```
UserDTO[] projectMembers = project.getProjectUsers();
StringBuilder projectMemberNamesBuilder = new StringBuilder();

for (int i = 0; i < projectMembers.length; i++) {

    projectMemberNamesBuilder.append(projectMembers[i]);

    if (i < projectMembers.length - 1) {
        projectMemberNamesBuilder.append(", ");
    }
}

Label projectMembersLabel = new Label("Project Members: " + projectMemberNamesBuilder.toString());
projectMembersLabel.setSizeUndefined();
projectMembersLabel.addClassName("overviewElement");

UserDTO projectCreationUser = RestRequestHelper.findUser(project.getCreatedByUser());

SimpleDateFormat issueCreationDateFormatter = new SimpleDateFormat("dd MMM yyyy HH:mm:ss", Locale.ENGLISH);
Label projectCreationLabel =
        new Label("Created at: " + issueCreationDateFormatter.format(project.getProjectCreationDate()) + " by " + projectCreationUser.getUserName());
projectCreationLabel.setWidthFull();
projectCreationLabel.addClassName("centeredText");

HorizontalLayout buttonLayout = new HorizontalLayout();
buttonLayout.addClassName("overviewElement");

Button modifyIssueButton = new Button("Modify Project");
modifyIssueButton.addClickListener(e -> {
    UI.getCurrent().navigate(ModifyProjectView.class, this.projectId);
});
```

Abbildung 1: Ausschnitt der Initialisierungsmethode der ProjectView

Um dieses Problem zu Lösen wird die Initialisierung von Komponenten, welche mehr wie einige Programmcode-Zeilen benötigen, in eigene Methoden ausgelagert, um die Initialisierungsmethode auf das Wesentliche zu beschränken.

Wie man sehen kann, wurden die meisten Komponenten, welche mehr wie drei Zeilen verbraucht haben, in eigene Methoden ausgelagert sowie der Programmcode zur Generierung der Projektmitglieder-Liste. Die eigentliche Logik der Initialisierungsmethode bleibt dennoch vorhanden, nur wurden in diesem Schritt die Details in eigene Methoden ausgelagert, um die Hauptmethode leserlicher zu gestalten.

```
private void initProjectView() {

    ProjectDTO project = RestRequestHelper.getProject(this.projectId);

    H2 projectTitle = new H2(project.getProjectTitle());
    projectTitle.setWidthFull();
    projectTitle.addClassName("centeredText");

    Label projectDescription = new Label(project.getProjectDescription());
    projectDescription.setWidthFull();
    projectDescription.addClassName("centeredText");

    Label projectMembersLabel = new Label("Project Members: " + generateProjectMemberList(project.getProjectUsers()));
    projectMembersLabel.setSizeUndefined();
    projectMembersLabel.addClassName("overviewElement");

    Label projectCreationLabel = generateProjectCreationLabel(project);
    HorizontalLayout buttonLayout = generateButtonLayout();
    HorizontalLayout inviteLayout = generateInviteLayout(project);

    add(projectTitle, projectDescription, projectMembersLabel, projectCreationLabel);
    addProjectIssuesIfNeeded(project.getProjectIssues());
    add(inviteLayout, buttonLayout);
}
```

Abbildung 2: ProjectView Initialisierungsmethode nach Refactoring

4.2 Code-Refactoring 2 | Duplicated Code

Nach der Einführung des Observers und eigenen Labelkomponenten zum automatischen Aktualisieren dieser Komponenten war duplizierter Code vorhanden, welcher zur Darstellung von der vergangenen Zeitdauer für die Labels diente.

```
private void updateText() {

    long issueTimePending = this.issue.getIssueTimePending();
    setText(this.messageBase + generateIssueTimePendingString(issueTimePending));
}

private String generateIssueTimePendingString(long issueTimePending) {

    StringBuilder issueTimePendingBuilder = new StringBuilder();
    int days = setTimeValue(86400000, issueTimePending);
    int hours = setTimeValue(3600000, issueTimePending);
    int minutes = setTimeValue(60000, issueTimePending);

    if (days > 0) {
        issueTimePendingBuilder.append(days + " day(-s) ");
    }

    if (hours > 0) {
        issueTimePendingBuilder.append(hours + " hours(-s) ");
    }

    issueTimePendingBuilder.append(minutes + " minutes(-s)");

    return issueTimePendingBuilder.toString();
}

private int setTimeValue(long timePeriod, long issueTimePending) {

    int timeUnit = 0;

    while (issueTimePending > timePeriod) {
        timeUnit++;
        issueTimePending -= timePeriod;
    }

    return timeUnit;
}
```

Abbildung 3: Ausschnitt des duplizierten Codes

Dieser Code-Ausschnitt existiert in beiden Labelkomponenten, um die zeitliche Darstellung der vergangenen Zeitdauer zu generieren. An dieser Stelle können wir, da beide Klassen denselben Programmcode enthalten für diesen Abschnitt diese Generierung in eine Hilfsklasse auslagern. Somit verpacken wir den duplizierten Programmcode an einer einzigen Stelle und referenzieren an beiden Stellen auf diese Hilfsklasse und die entsprechende Methode, um so die Duplikation zu eliminieren.

```
private void updateText() {

    long issueTimePending = this.issue.getIssueTimePending();
    setText(this.messageBase + TimeStringGenerationHelper.generateTimeStringValue(issueTimePending));
}
```

Abbildung 4: Referenzierung mittels Hilfsklasse

Kapitel 5

Unit-Tests

Unit-Testing trägt zwar theoretisch nichts zu dem Endprodukt bei, ist jedoch essenziell zur Sicherstellung des Qualitätsanspruchs eines Softwareprojekts, da nahezu jede Software fehlerbehaftet ist. Da es den Rahmen des Projekts übersteigen würde, wurde innerhalb der Tests sich auf einige wenige wichtige Komponenten beschränkt, dessen Funktionalität essenziell für die Gesamtapplikation ist.

Diese Komponenten sind dabei folgende:

- Die verschiedenen Controller innerhalb der Plugin-Schicht, welche REST-Endpunkte nach Daten abfragen.
- Die verschiedenen Adapter, welche die intern verwendeten Objekte in DataTransfer-Objekte konvertieren und umgekehrt.
- Objektkreationstests für die verschiedenen Domänenobjekte.

5.0.1 ATRIP-Regeln

Die ATRIP-Regeln werden innerhalb des Projekts folgendermaßen umgesetzt:

- **Automatic** -> Gegeben durch die einfache Ausführbarkeit mittels Maven. (mvn test)
- **Thorough** -> Gegeben durch den Fakt, dass die Tests die wichtigsten Komponenten innerhalb des Projekts abdecken und nach dessen Funktionalität überprüfen.
- **Repeatable** -> Gegeben, da jeder Test unendlich oft wiederholbar ist und das Selbe Ergebnis liefert.
- **Independent** -> Gegeben, da kein Test Abhängigkeiten zu einem anderen Test besitzt.
- **Professional** -> Es wurden keine für den Test nicht nötigen Codeabschnitte geschrieben.

5.0.2 Verwendung von Mocks

Manchmal ist es innerhalb von Tests nicht möglich, eine bestimmte Klasse zu testen, da bestimmte Abhängigkeiten existieren, welche nicht innerhalb eines anderen Tests getestet werden müssen. In solchen Fällen werden sogenannte *Mocks* verwendet, um diese anderen Objekte innerhalb des Testkontexts zu konsistent simulieren.

Gerade innerhalb der Controller-Tests in der Plugin-Schicht werden Mocks verwendet, um Repositories zu simulieren, welche im realen Projektkontext durch das Spring-Framework automatisiert erstellt werden, was innerhalb des Testkontexts nur schwer möglich ist.

Kapitel 6

Entwurfsmuster

Bei Entwurfsmustern handelt es sich um Objekte und Muster innerhalb der objektorientierten Programmierung, welche auf verschiedenen Problemen wiederverwendbar sind. Dabei können diese Muster entweder auf Klassenebene oder auf Objektebene ihren Geltungsbereich besitzen. Der Unterschied zwischen diesen beiden Varianten liegt darin, dass Muster auf Klassenebene statisch und zur Kompilierzeit festgelegt werden und Muster auf Objektebene dynamisch und zur Programmlaufzeit.

Ein Beispiel für ein Klassenmuster ist eine Fabrikmethode innerhalb einer Klasse, welche ein bestimmtes Objekt beispielsweise generiert.

Beispiele für Objektmuster sind Strukturierungen wie der Decorator, die Facade oder der Observer.

6.1 Analyse des verwendeten Entwurfsmusters

6.1.1 Observer-Pattern

Das Observer-Entwurfsmuster ist wie bereits angesprochen in der Kategorie der Entwurfsmuster auf Objektebene vorzufinden und beschäftigt sich damit mit der Objektstruktur innerhalb des Projekts. Dabei besitzt ein sogenanntes *Subjekt* oder auch *Publisher* eine Liste von *Beobachtern*. Wenn nun Zustandsänderungen bei allen Beobachtern geschehen sollen, kann bei dem Subjekt eine Aktualisierungsmethode aufgerufen werden, welche schließlich die komplette Liste von Beobachtern durchgeht und für jeden Beobachter eine passende Aktualisierungsmethode aufruft.

Somit ist es einfach für eine Menge an Objekten über eine einzelne Schnittstelle möglich, Aktualisierungen auszuführen.

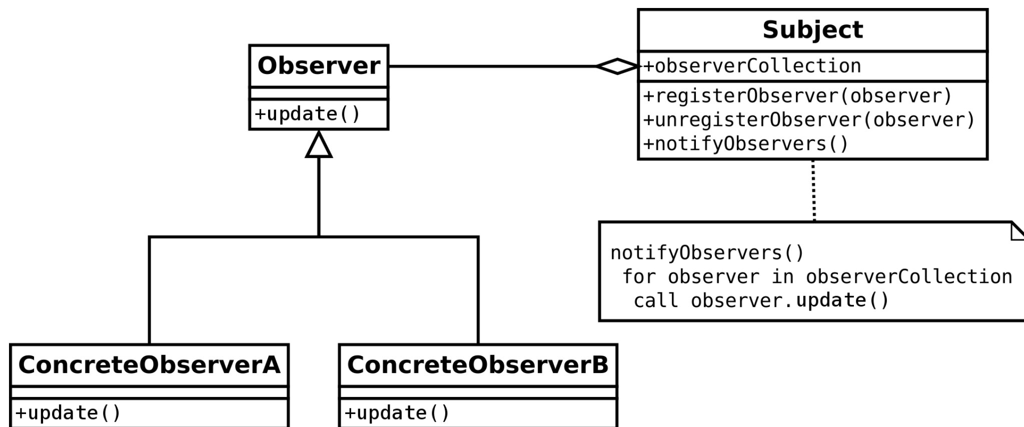

Abbildung 5: Beispiel UML-Diagramm zum Observer-Pattern [1]

6.1.2 Anwendung im Projekt

Innerhalb der Issue-Ansicht existieren zwei verschiede Informationstexte, welche den Nutzer informieren, wie lange eine Issue insgesamt existiert und wie lange sie in einem gewissen Status verweilt. Die Idee war nun an dieser Stelle über einen nebenläufigen Thread ohne die Website zu aktualisieren, die jeweiligen Informationstexte zu aktualisieren.

Um dies zu bewerkstelligen, wird zunächst einmal ein nebenläufiger Thread erstellt, welcher in diesem Anwendungsfall jede Minute eine Aktion ausführt.

Daraufhin wird ein Subjekt erstellt, welches wie bereits angesprochen eine Liste von Beobachtern enthält, worin schließlich beide Textkomponenten hinzugefügt werden. Beide Textkomponenten werden schließlich mit einer Aktualisierungsmethode über ein Interface erweitert, welche den Text der Komponente auf den neuesten Stand aktualisiert.

Somit können nun schließlich durch das Aufrufen der Aktualisierungskomponente innerhalb des Threads beide Textkomponenten nebenläufig gleichzeitig aktualisiert werden.

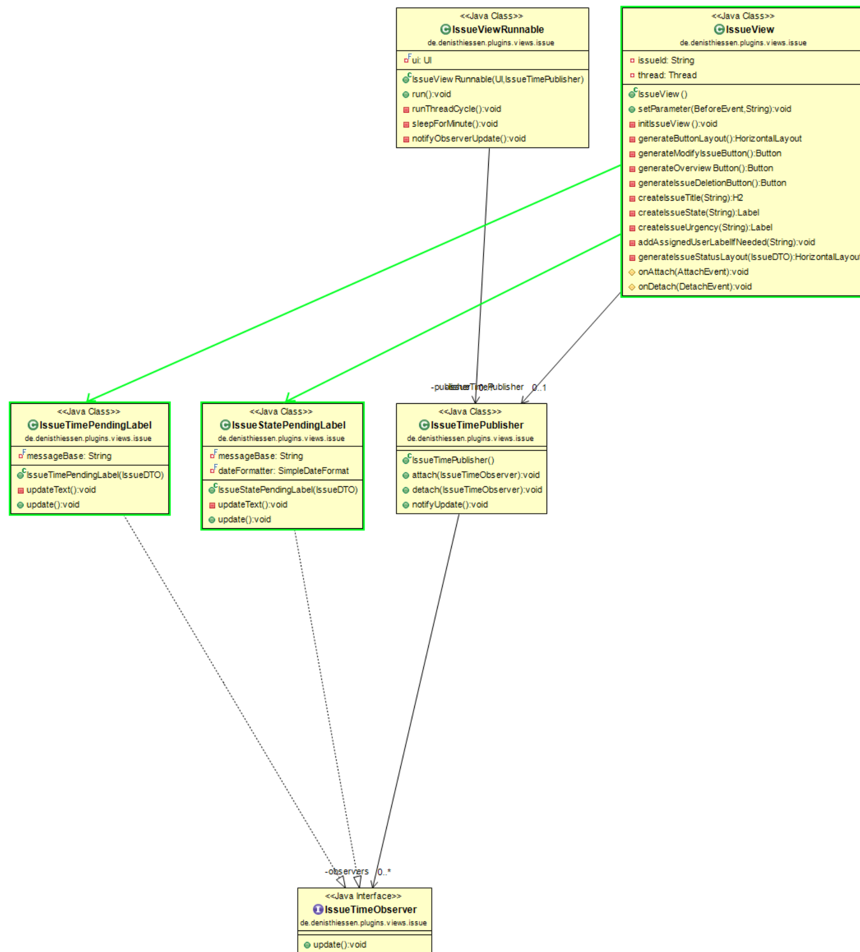

Abbildung 6: UML-Diagramm des Observer-Musters (grün: vorher)

Literatur

[1] Wikipedia. *Observer Reference UML-Diagram*. 2018. URL: https://upload.wikimedia.org/wikipedia/commons/a/a8/Observer_w_update.svg.