ADVANCED SOFTWARE ENGINEERING SCHRIFTLICHE DOKUMENTATION

Matrikelnummer: 3935033

Inhaltsverzeichnis

1 Domain Driven Design

1.1 Analyse der Ubiquitous Language

Die Ubiquitous Language ist ein Konzept des Domain Driven Design, das dazu dient, Verständnisprobleme zwischen Entwicklern und Domänenexperten zu vermeiden, indem eine gemeinsame Projektsprache definiert wird. Diese Projektsprache umfasst Definitionen für ein gemeinsames Verständnis von Konzepten, Prozessen und Regeln innerhalb der Domäne.

Wichtige Begriffe und Prozesse innerhalb der Problemdomäne "To-Do-App" sind:

- Nutzer
- To-Do
- To-Do-Liste
- Sub-To-Do
- Fälligkeitsdatum
- Erinnerungsdatum
- Benachrichtigung
- Liste anlegen
- Liste löschen/bearbeiten
- Todo anlegen
- Todo verschieben/löschen/bearbeiten
- Benachrichtigung anlegen
- Benachrichtigung löschen/bearbeiten

Nutzer:

Ein Nutzer registriert und authentifiziert sich in der Anwendung durch eine Kombination aus einer E-Mail und einem Passwort. Die Mail-Adresse muss dabei so aufgebaut sein, dass sie aus einem lokalen Teil und einem Domänenteil besteht, die durch das @-Zeichen getrennt sind und nur valide Zeichen enthalten. Ein Passwort muss aus mindestens 8 Zeichen bestehen und darf nur Buchstaben und Zahlen beinhalten.

To-Do:

Ein To-Do (Aufgabe) besteht aus einem Namen, einer Beschreibung, einer beliebigen Anzahl von Sub To-Dos (Unteraufgaben), einem Erinnerungsdatum sowie einem Fälligkeitsdatum. Außerdem wird für ein To-Do intern das Erstellungs- und Abschlussdatum gespeichert. Zudem hat ein To-Do mehrere mögliche Zustände:

- Abgeschlossen und nicht abgeschlossen
- Fälligkeitsdatum überschritten/nicht überschritten

To-Do-Liste:

Eine To-Do-Liste (Aufgabenliste) kann eine beliebige Anzahl von To-Dos beinhalten. Sie kann aber auch keine To-Dos enthalten. To-Do-Listen können vom Benutzer angelegt werden. Dabei wird vom Benutzer ein Name vergeben, der über alle Listen hinweg eindeutig sein muss. Die Bestandteile einer To-Do-Liste sind also ihr Name und eine Reihe von To-Dos.

Sub-To-Do:

Ein Sub-To-Do (Unteraufgabe) kann selbst wieder eine Aufgabe sein, mit dem Unterschied, dass ein Sub-To-Do keine weiteren Sub-To-Dos enthalten kann.

Fälligkeitsdatum:

Ein Fälligkeitsdatum wird einem To-Do zugeordnet und beschreibt das Datum, bis zu dem die Aufgabe abgeschlossen werden soll. Es ist darauf zu achten, dass das Fälligkeitsdatum nicht in der Vergangenheit liegen darf. Als Fälligkeitsdatum werden nur Tag, Monat und Jahr berücksichtigt. Ist das Fälligkeitsdatum eines To-Do überschritten, wird dies in der der Anwendung entsprechend gekennzeichnet.

Erinnerungsdatum:

Auf einer Aufgabe kann ein Erinnerungsdatum gesetzt werden, welches das Datum beschreibt, zu dem der Nutzer eine Erinnerung bekommen soll, die ihn auf den rechtzeitigen Abschluss erinnern soll. Auch hier ist darauf zu achten, dass das Erinnerungsdatum nicht in der Vergangenheit liegt. Ebenso werden nur Tag, Monat und Jahr für das Datum berücksichtigt. Wenn das Erinnerungsdatum auf einer Aufgabe erreicht ist, kann der Benutzer darüber informiert werden, indem er zuvor eine Benachrichtigung angelegt hat.

Benachrichtigung:

Eine Benachrichtigung besteht aus einem Namen und einer Webhook-URL. Der Name soll Nutzern die Verwaltung von beliebig vielen Benachrichtigungen erleichtern. Als Webhook-URL kann nur eine URL eingetragen werden, die Nutzer von einem Discord-Server erhalten können. Andere Webhooks werden nicht unterstützt. Sobald eine Benachrichtigung eingerichtet ist, wird der Nutzer über Ereignisse in der Anwendung informiert.

To-Do-Liste anlegen:

Eine neue To-Do-Liste erstellen bedeutet, ihr einen eindeutigen Namen zu geben. Nach der Erstellung ist die Liste zunächst leer. Der Benutzer hat nun die Möglichkeit, dieser To-Do-Liste Aufgaben hinzuzufügen.

To-Do-Liste löschen/bearbeiten:

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

To-Do löschen/bearbeiten:

Ein To-Do zu löschen bedeutet, alle mit ihr verbundenen Daten aus der Anwendung zu entfernen, so dass sie nicht mehr zugänglich sind.

Ein To-Do bearbeiten bedeutet, die oben genannten Elemente einer Aufgabe zu ändern oder die Aufgabe als abgeschlossen oder nicht abgeschlossen zu markieren. Wenn ein To-Do als abgeschlossen markiert wird, wird es in der Oberfläche aus der aktuellen To-Do-Liste entfernt und in einer separaten Liste für abgeschlossene To-Dos angezeigt. Wenn ein To-Do wieder als nicht abgeschlossen markiert wird, ändert sich der Status entsprechend und der To-Do wird wieder in seiner ursprünglichen To-Do-Liste angezeigt.

Benachrichtigung anlegen

Ein Nutzer kann eine Benachrichtigung anlegen, um über Ereignisse in der Anwendung über einen Webhook nach Discord informiert zu werden.

Benachrichtigung löschen/bearbeiten

Wird eine Benachrichtiung gelöscht, bekommt der Nutzer keine Informationen mehr über den Webhook.

Eine Benachrichtigung zu bearbeiten bedeutet entweder den Namen anzupassen oder die URL des Webhooks anzupassen.

1.2 Analyse und Begründung der verwendeten Muster

Value Objects:

Value Objects (Wertobjekte) erfassen einen oder mehrere Werte in einem neuen Wert oder Typ. Das Value Object kann daraufhin bestimmte Regeln überprüfen, die in der Problemdomäne für den eingekapselten Wert gelten. Ein VO muss unveränderlich sein, was bedeutet, dass sein Zustand nach der Erzeugung nicht mehr geändert werden kann. Zwei VOs sind gleich, wenn ihre eingekapselten Werte übereinstimmen.

Die Klasse `de.dhbw.ase.todoapp.domain.vo.Email` ist ein solches Value Object. Sie wird verwendet, um eine Zeichenkette zu kapseln, die dem Format einer E-Mail entsprechen muss. Nach der Instanziierung wird die Zeichenkette auf das korrekte Format überprüft und kann danach nicht mehr geändert werden. Die beiden Methoden `hashCode()` und `equals()` werden überschrieben, um die Gleichheit zweier Objekte bei Gleichheit der E-Mail-Adressen zu gewährleisten.

Weitere Value Objects sind:

- `de.dhbw.ase.todoapp.domain.vo.Name`: Speichert den Namen eines To-Do, welcher nicht `null` oder leer sein darf.
- `de.dhbw.ase.todoapp.domain.vo.Description`: Speichert die Beschreibung eines To-Do, welcher eine bestimmte Zeichenlänge nicht überschreiten darf.
- `de.dhbw.ase.todoapp.domain.vo.CalendarDate`: Dient der Kapselung eines Werts vom Typ `java.util.LocalDate` zur Speicherung vom Erinnerungs- und Abschlussdatum eines To-Do.
- `de.dhbw.ase.todoapp.domain.vo.Password`: Prüft die Anforderungen an das Passwort eines Nutzers in der Anwendung.
- `de.dhbw.ase.todoapp.domain.vo.WebHook`: Speichert eine Zeichenkette zur Verwendung als Discord-WebHook-URL einer Benachrichtigung.

Auch diese Value Objects sind unveränderlich, kapseln ein ''Wertkonzept'' und sind so implementiert, dass die Gleichheit zweier Objekte bei gleichen Werten gewährleistet wird.

Entities:

Entitäten haben eine eindeutige Identität und einen eigenen Lebenszyklus. Sie aggregieren Werte, einschließlich Wertobjekte, zu einem Ganzen und repräsentieren die Daten oder "Dinge", die für die Domäne relevant sind und mit denen die Anwendung arbeitet. Entitäten werden in der Persistenz-Schicht gespeichert.

Die Klasse `de.dhbw.ase.todoapp.domain.entities.todo.Todo` ist eine Entität, die den Kern der Daten darstellt, mit denen eine To-Do-Anwendung arbeitet. Sie eindeutig durch eine ID identifizierbar und muss, um für den Anwender und die Anwendung von Nutzen zu sein, persistiert werden. Sie aggregiert dafür mehrere Daten wie beispielsweise die VOs `de.dhbw.ase.todoapp.domain.vo.Name`, `de.dhbw.ase.todoapp.domain.vo.Description` und `de.dhbw.ase.todoapp.domain.vo.Email`. Die Daten, die von der Entität aggregiert werden, können des Weiteren vom Anwender geändert werden d.h., es existiert ein eigener Lebenszyklus.

Weitere Entities sind:

- `de.dhbw.ase.todoapp.domain.entities.todo.TodoList`: Eine TodoList aggregiert einen eindeutigen Namen und eine beliebige Anzahl von To-Dos.
- `de.dhbw.ase.todoapp.domain.entities.user.User`: Ein User aggregiert eine Kombination aus E-Mail und Passwort.
- `de.dhbw.ase.todoapp.domain.entities.notification.Notification`: Eine Notification aggregiert einen Namen und eine WebHook-URL.

Auch diese Entities sind durch eine ID identifizierbar, aggregieren mehrere Daten, die durch Value Objects beschrieben werden und haben alle einen eigenen Lebenszyklus innerhalb der Anwendung, da sie erstellt (TodoList, User, Notification), bearbeitet (TodoList) und wieder gelöscht werden können (TodoList, Notification).

Domain Services:

Ein Domain Service ist eine Hilfsklasse, die komplexe Regeln innerhalb des Domänenmodells implementiert. Diese Regeln können nicht eindeutig einem Wertobjekt oder einer Entität zugeordnet werden. Ein solcher Service wird benötigt, wenn mehrere Entitäten und Wertobjekte zur Überprüfung herangezogen werden müssen oder externe Dienste eingebunden werden müssen. Durch die Auslagerung solcher Überprüfungen in Domain Services wird verhindert, dass die Domäne unnötig komplex wird.

Als Domain Services dienen in der Anwendung die Klassen `de.dhbw.ase.todoapp.domain.entities.notification.reminder.ReminderDateStrategy` und `de.dhbw.ase.todoapp.domain.entities.notification.reminder.DueDateStrategy` .

Beide Klassen sind außerdem Implementierungen eines Strategy-Pattern und sind dazu da, das Erinnerungsdatum und Abschlussdatum eines To-Dos zu überprüfen. Ist eines der beiden Daten überschritten, werden Notification-Objekte, die ebenfalls an den Domain Service übergeben werden,

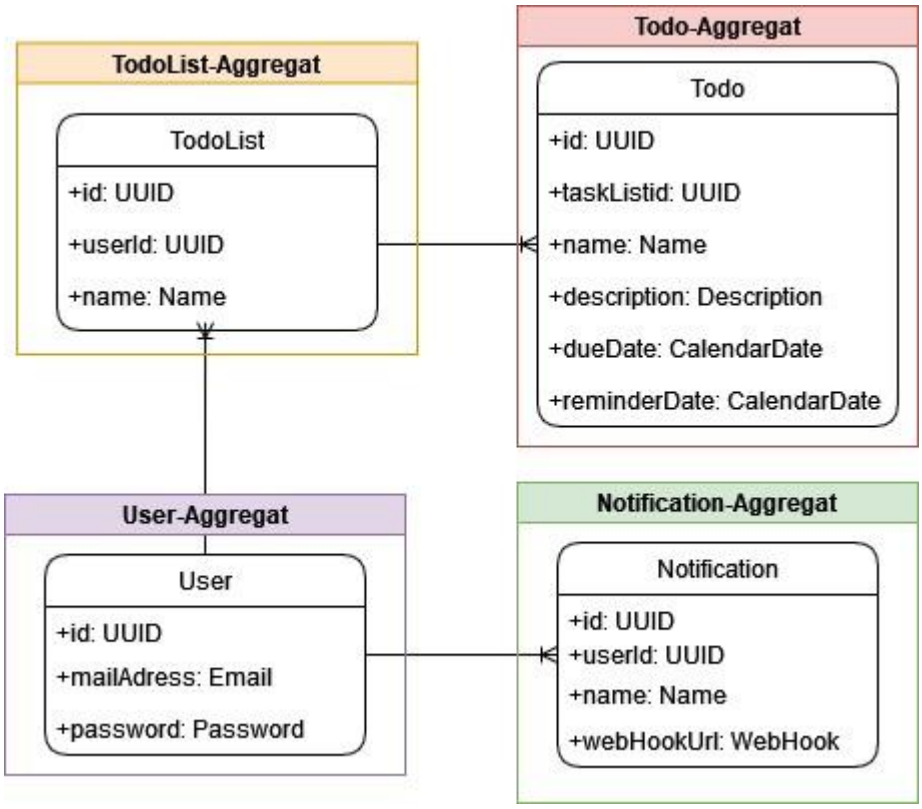
mit einer entsprechenden Nachricht darüber informiert. Daher werden hier für die Überprüfung mehrere Entitäten und VOs benötigt, was die Auslagerung dieser Komplexität in einen Domain Service begründet.

Aggregates:

Aggregate sind Sammlungen von Entitäten oder Wertobjekten, die in Beziehung zueinander stehen können (1:n, n:m oder 1:1 usw.). Jedes Aggregate hat eine sogenannte Aggregate-Root, über die auf die Elemente des Aggregats zugegriffen werden kann. Aggregate werden verwendet, um

Objektbeziehungen zu organisieren und Domänenregeln einzuhalten. Die Aggregate-Root fungiert als Zugangspunkt und steuert den Zugriff auf Objekte innerhalb des Aggregats sowie die Änderung von Daten.

Die Anwendung wurde so gestaltet, dass jede Entität ein eigenes Aggregate bildet, mit sich selbst als Aggregate Root. Es existieren also die Aggregate: User-Aggregat, TodoList-Aggregat, Todo-Aggregat und Notification-Aggregat. Die zwei Entitäten TodoList und Todo hätte man als ein Aggregat zusammenfassen können, jedoch wurde sich dagegen entschieden, da so die Implementierung der Persistenz-Schicht etwas übersichtlicher gestaltet ist, durch die Trennung des Anwendungscode zwischen den beiden Entitäten.



Repositories:

Ein Repository ist die Schnittstelle zwischen der Domäne und der Persistenz-Schicht. Für jedes Aggregate Root (s.o.) wird ein Repository benötigt. In diesem Fall sind dies das `de.dhbw.ase.todoapp.domain.entities.user.UserRepository`,

`de.dhbw.ase.todoapp.domain.entities.todo.TodoListRepository`,
`de.dhbw.ase.todoapp.domain.entities.todo.TodoRepository` und das
`de.dhbw.ase.todoapp.domain.entities.notification.NotificationRepository`.

Ein Repository ist zunächst nur eine Schnittstelle (Interface) die aber eine Vielzahl von Ausprägungen haben kann. Wichtig ist, dass diese konkreten Ausprägungen nicht mit der Domäne vermischt werden, da diese sich nicht für die konkrete Implementierung der Persistenz interessiert. Der Domäne wird nur das Interface präsentiert. Dies hat den Vorteil, dass im Hintergrund beliebige Implementierungen genutzt und diese auch reibungslos ausgetauscht werden können. Folgende Implementierungen sind in dieser Applikation vorhanden:

- `de.dhbw.ase.todoapp.plugins.persistence.UserRepositoryBridge`
- `de.dhbw.ase.todoapp.plugins.persistence.TodoListRepositoryBridge`
- `de.dhbw.ase.todoapp.plugins.persistence.TodoRepositoryBridge`
- `de.dhbw.ase.todoapp.plugins.persistence.NotificationRepositoryBridge`

Bei allen vier Klassen handelt es sich um Brücken zwischen dem Repository-Interface und der `JpaRepository`-Klasse, die die tatsächliche Implementierung der Persistierung der Daten über JPA ermöglicht.

2 Clean Architecture

Das Hauptziel der Clean Architecture besteht darin, Software so zu konzipieren, dass sie langfristig Bestand hat. Eine grundlegende Konzeption dabei ist die klare Separierung zwischen technologieabhängigen und technologieunabhängigen Teilen. Die Architektur soll es ermöglichen, die zugrundeliegenden Technologien auszutauschen, ohne dass der Kern der Anwendung davon beeinflusst wird.

2.1 Schichtarchitektur planen und begründen

Die Clean Architecture unterteilt eine Software in mehrere Schichten, wobei Abhängigkeiten immer nur von außen nach innen gehen dürfen. Dabei ist es wichtig, dass innere Schichten nichts von den äußeren Schichten wissen. Eine klare Schichtarchitektur in Clean Architecture hilft dabei, die Software langfristig wartbar, flexibel und unabhängig von den zugrundeliegenden Technologien zu gestalten.

Im vorliegenden Projekt lassen sich theoretisch fünf Schichten unterscheiden, die von außen nach innen angeordnet sind:

- **Plugins:** In der Plugin-Schicht befindet sich das Benutzerinterface und sonstige Ressourcen, die dafür benötigt werden. Außerdem enthält sie Code mit direkten Abhängigkeiten zu externen Bibliotheken oder Frameworks.
- **Adapters:** Die Adapter-Schicht enthält hauptsächlich Mapping-Code, der die Daten aus der Plugin-Schicht in ein Format umwandelt, das von der Applikationsschicht verstanden wird, und umgekehrt. Diese Schicht kann oft weggelassen werden, wenn die Datenmodelle sich nicht stark unterscheiden.
- **Application Code:** Diese Schicht enthält Code, der direkt mit den Anwendungsfällen und Anforderungen der Applikation zusammenhängt. Sie fasst Elemente aus dem Domänen-Code zusammen und verwendet diese zur Umsetzung der Anwendungsfälle. Änderungen in dieser Schicht dürfen den Domänen-Code nicht beeinflussen und umgekehrt. Die Use Cases dieser Schicht sollten nicht interessiert sein, wer sie aufruft und wie das Ergebnis präsentiert wird.
- **Domain Code:** Hier befindet sich der Code, der direkt mit der Problemdomäne zusammenhängt, wie Entitäten und Value Objects. Änderungen in dieser Schicht sollten unabhängig von Änderungen an anderen Schichten sein.
- **Abstraction Code:** Diese Schicht enthält domänenübergreifendes Wissen und Funktionalitäten, die in vielen oder allen anderen Schichten benötigt werden. Sie abstrahiert beispielsweise von Code oder Abhängigkeiten auf Bibliotheken.

Die Schichten wurden jeweils als einzelne Maven-Module umgesetzt. Im Folgenden werden die Schichten näher erläutert:

- **0-todoapp-plugins:** Die Plugin-Schicht ist die am Weitesten außen liegende Schicht deren Code sich am häufigsten ändert, da er am konkretesten ist und direkte Abhängigkeiten zu externen Bibliotheken und Frameworks besitzt. Hier liegen alle UI-Elemente und die dafür benötigten Ressourcen (Thymeleaf). Außerdem werden die Repository-Interfaces durch konkrete Implementierungen ersetzt (Spring/JPA).

- **1-todoapp-adapters:** Die Adapter-Schicht wurde in dieser Anwendung nicht mit Leben befüllt. Hier könnte man bspw. nötigen Mapping-Code unterbringen, um vom Datenmodell der PluginSchicht auf das Datenmodell der Applikationsschicht abzubilden.
- **2-todoapp-application:** Die Applikations-Schicht enthält Klassen und Methoden zur Umsetzung der Use-Cases. Sie verwendet dafür die Repositories, Entitäten und VOs aus der darunterliegenden Schicht. In diesem Projekt wurden Use-Cases unterschieden, welche die To-Do-Listen, To-Dos, Benutzer oder Benachrichtigungen betreffen und diese jeweils in einer Service-Klasse zusammengefasst. Die von den Klassen bereitgestellten Methoden werden in Plugin-Schicht verwendet.
- **3-todoapp-domain:** In der Domänen-Schicht befinden sich die Entitäten und VOs, da diese aus der Domäne und während des DDD-Prozesses entstanden sind und die dort geltenden Regeln einhalten müssen. Ebenfalls untergebracht sind dort die Repositories der vorhandenen Aggregate, da die Verwaltung der Entitäten ebenfalls zur Problemdomäne gehört.
- **4-todoapp-abstractioncode:** In der Abstraktions-Sicht befinden sich nur die Klassen zur Umsetzung des Beobachtermusters, weil dieser Code sich nur sehr selten ändert und innerhalb des gesamten Projektes nutzbar sein soll.

3 Programming Principles

3.1 Single Responsibility Principle (SRP)

Das Single Responsibility Principle (SRP) besagt, dass eine Softwarekomponente nur eine einzige bestimmte Aufgabe haben sollte, für die sie zuständig ist. Dies kann auf verschiedenen Ebenen angewendet werden, einschließlich Module, Klassen, Methoden und Variablen. Ein Beispiel für die Anwendung des SRP ist eine der Repository-Klassen wie zum Beispiel das `UserRepository`. Gemäß dem SRP sollte diese Klasse nur eine einzige Verantwortung haben, nämlich die Interaktion mit der Datenbank für das Speichern und Laden von Benutzerdaten. Das SRP bietet durch seine Eigenschaften den Vorteil, dass innerhalb des Codes eine niedrigere Kopplung und Komplexität entsteht.

3.2 Interface Segregation Principle (ISP)

Das Interface Segregation Principle (ISP) besagt, dass es besser ist, viele spezifische Interfaces zu haben, die jeweils nur die Methoden enthalten, die von den Clients benötigt werden, anstatt ein großes, allgemeines Interface zu verwenden, das alle möglichen Methoden enthält. Durch die Aufteilung von Interfaces nach den Bedürfnissen der Clients wird vermieden, dass Clients Methoden implementieren müssen, die sie gar nicht verwenden.

Im vorliegenden Projekt erfüllt das Interface `UserRepository` das Prinzip des Interface Segregation. Statt ein allgemeines Repository-Interface zu haben, das Methoden zur Verwaltung aller Entitäten enthält, wird für jede Entität ein separates Repository-Interface definiert. Das `UserRepository`-Interface enthält nur die Methoden, die für die Verwaltung von User-Daten benötigt werden, ohne unnötige Methoden für andere Entitäten.

Durch diese Aufteilung wird sichergestellt, dass Implementierungen des `UserRepository`-Interfaces nur die Methoden für die Benutzerverwaltung implementieren müssen. Andere Repositories, die beispielsweise für die Verwaltung von To-Do-Listen oder To-Dos zuständig sind, implementieren jeweils ihre eigenen, spezifischen Interfaces. Dadurch werden leere Implementierungen vermieden und das Interface Segregation Principle eingehalten.

3.3 Dependency Inversion Principle (DIP)

Das Dependency Inversion Principle (DIP) besagt, dass Module auf höherer Ebene nicht von den Details der Module auf niedrigerer Ebene abhängig sein sollten. Stattdessen sollten beide Ebenen von einer Abstraktion abhängen. Dadurch wird die Flexibilität und Wartbarkeit des Systems verbessert, da Änderungen in den niedrigeren Ebenen keine direkten Auswirkungen auf die höheren Ebenen haben.

Im vorliegenden Projekt wird das DIP durch die Einhaltung der Clean Architecture umgesetzt. Diese Architektur stellt sicher, dass Abhängigkeiten ausschließlich von äußeren zu inneren Schichten verlaufen. Das bedeutet, dass beispielsweise die Anwendungslogik nicht direkt von Datenbankoperationen oder anderen technischen Details abhängig ist. Stattdessen kommuniziert sie über definierte Schnittstellen mit den darunterliegenden Schichten, wodurch eine lose Kopplung und eine klare Trennung der Verantwortlichkeiten erreicht werden.

Ein gutes Beispiel dafür ist das Interface `UserRepository`. Durch die Verwendung dieses Interfaces in der Anwendungslogik wird vermieden, dass die Anwendungslogik direkt von der Persistenz-Schicht abhängig ist. Stattdessen hängt sie von der abstrakten Schnittstelle `UserRepository` ab, was es ermöglicht, verschiedene Implementierungen dieser Schnittstelle zu verwenden, ohne Änderungen an

der Anwendungslogik vornehmen zu müssen. Dadurch wird das DIP eingehalten und die Flexibilität des Systems verbessert.

3.4 Pure Fabrication (reine Erfindung)

Das Prinzip der Pure Fabrication (reine Erfindung), auch bekannt als Pure Fabrication, bezieht sich auf Klassen oder Module, die in der Problemdomäne nicht existieren, sondern nur zur Trennung von Technologiewissen und Domänenwissen dienen. Diese reinen Erfindungen sind oft Schnittstellen oder Implementierungen, die dazu dienen, bestimmte Aspekte der Anwendung zu abstrahieren oder zu kapseln, ohne dass sie direkte Entsprechungen in der Problemdomäne haben.

Ein anschauliches Beispiel dafür ist ebenfalls das Interface `UserRepository`. In der realen Welt gibt es keine explizite Entität oder Domänenkonzept namens `UserRepository`. Stattdessen wird dieses Interface erstellt, um die Schnittstelle zwischen der Anwendung und der Persistenz-Schicht zu definieren. Es hilft dabei, die Domänenlogik von den technischen Details der Datenbankzugriffe zu entkoppeln. Durch die Verwendung von Pure Fabrication wird die Flexibilität der Anwendung erhöht, da Änderungen in der Persistenz-Schicht keine direkten Auswirkungen auf die Domänenlogik haben.

3.5 Low Coupling (lose Kopplung)

Low Coupling (lose Kopplung) bezieht sich auf den Grad der Abhängigkeit zwischen zwei Softwarekomponenten. Bei der Entwicklung von Software strebt man in der Regel eine geringe Kopplung an, was bedeutet, dass die Abhängigkeiten zwischen den Komponenten minimiert werden sollen.

Eine Möglichkeit, eine lose Kopplung zu erreichen, ist der Einsatz von Interfaces. Interfaces abstrahieren von der konkreten Implementierung und entkoppeln die Komponenten, die die definierte Schnittstelle nutzen, von der tatsächlichen Implementierung. Die Klasse `UserRepositoryBridge` erfüllt das Prinzip von Low Coupling, indem sie das entsprechende Repository-Interface implementiert. Durch die Verwendung dieses Interfaces wird die Klasse von der konkreten Implementierung des Repository entkoppelt. Sie interagiert nur über die definierte Schnittstelle mit dem Repository, ohne direkt von dessen Implementierungsdetails abhängig zu sein.

Diese Entkopplung bietet mehrere Vorteile: Erstens ermöglicht sie eine einfachere Wartung und Anpassung des Codes, da Änderungen an der Implementierung des Repository keinen direkten Einfluss auf die `UserRepositoryBridge` haben. Zweitens macht sie den Code besser testbar, da sich die Abhängigkeiten leichter simulieren oder durch Mock-Objekte ersetzen lassen.

4 Refactoring

4.1 Code Smells

Duplicate Code - `TodoController`

Der Code Smell `Dispensables` betrifft redundante oder überflüssige Codeabschnitte, die die Lesbarkeit und Wartbarkeit des Codes beeinträchtigen können. Ein häufiges Beispiel dafür ist duplizierter Code, bei dem identische oder ähnliche Codefragmente an verschiedenen Stellen im Code vorhanden sind. Diese Redundanz kann zu einem erhöhten Wartungsaufwand führen, da Änderungen an duplizierten Codeabschnitten an mehreren Stellen im Code vorgenommen werden müssen.

In der Klasse `TodoController` besteht der Code Smell `Duplicate Code`, da ein bestimmtes Codefragment zur Überprüfung, ob ein Benutzer bereits eingeloggt ist, mehrmals aufgerufen wird:

```
UUID userId = (UUID) session.getAttribute("userId");
if (userId == null) {
    return "redirect:/login";
}
```

Dieser Code wird am Anfang jeder Methode im `TodoController` verwendet, um sicherzustellen, dass nur eingeloggte Benutzer auf Funktionen zugreifen, die auf die Daten des Nutzers zugreifen. Diese Duplikation führt zu erhöhtem Wartungsaufwand und beeinträchtigt die Wartbarkeit des Codes. Es wäre besser, eine allgemeine Methode zu implementieren, die diese Überprüfung durchführt und somit Redundanzen vermeidet.

Long Class - `TodoController`

Der Code Smell `Long Class` aus der Kateogrie `Bloaters` bezieht sich auf Klassen, die zu viele Verantwortlichkeiten oder Funktionen haben und daher zu groß und unübersichtlich werden. Eine lange Klasse kann schwer zu verstehen und zu warten sein, da sie eine Vielzahl von Aufgaben in sich vereint.

In der Klasse `TodoController` scheint der Code Smell `Long Class` vorzuliegen, da die Klasse mit 307 Zeilen recht umfangreich ist. Die Länge der Klasse deutet darauf hin, dass sie möglicherweise zu viele Verantwortlichkeiten hat und verschiedene Funktionalitäten kapselt, die besser in separate Controller aufgeteilt werden sollten.

Durch die Aufteilung der Funktionalitäten in separate Controller, wie z.B. einen `TodoListController`, einen `TodoController` und einen `NotificationController`, kann die Lesbarkeit und Wartbarkeit des Codes verbessert werden. Jeder Controller kann sich dann auf eine spezifische Aufgabe konzentrieren und ist dadurch leichter zu verstehen und zu pflegen. Dies trägt dazu bei, den Code schlanker, besser strukturiert und leichter erweiterbar zu machen.

Long Parameter List - `TodoController`

Der Code Smell `Long Parameter List` aus der Kateogrie `Bloaters` tritt auf, wenn Methoden eine große Anzahl von Parametern enthalten, was die Lesbarkeit und Wartbarkeit des Codes beeinträchtigen kann. In der Klasse `TodoController` scheint dieser Code Smell in den Methoden `createSubTodo` und `editTodo` vorhanden zu sein, da sie eine lange Liste von Parametern enthalten, die an die Methode übergeben werden müssen.

```
@PostMapping("/createSubTodo")
public String createSubTodo(@RequestParam("todoListId") UUID todoListId,
@RequestParam("todoId") UUID todoId, @RequestParam("name") String name,
@RequestParam("description") String description, @RequestParam("dueDate") LocalDate
dueDate, @RequestParam("reminderDate") LocalDate reminderDate, HttpSession session,
Model model)

@PostMapping("/editTodo")
public String editTodo(@RequestParam("todoId") UUID todoId, @RequestParam("name")
String name, @RequestParam("description") String description,
@RequestParam("dueDate") LocalDate dueDate, @RequestParam("reminderDate") LocalDate
reminderDate, HttpSession session, Model model)
```

Die Verwendung von so vielen Parametern in einer Methode kann dazu führen, dass der Methodenaufruf schwer zu lesen und zu verstehen ist. Außerdem erhöht sich das Risiko von Fehlern bei der Parameterübergabe, insbesondere wenn die Parameterreihenfolge nicht eindeutig ist.

Eine bessere Lösung für dieses Problem besteht darin, Data Transfer Objects (DTOs) zu verwenden, um die Daten zu kapseln und an die Methode zu übergeben. Ein DTO ist eine einfache Datenklasse, die nur die benötigten Attribute enthält und keine zusätzliche Logik oder Verantwortlichkeiten hat. Durch die Verwendung von DTOs können die Methodenaufrufe vereinfacht und die Lesbarkeit des Codes verbessert werden. Außerdem wird die Flexibilität erhöht, da Änderungen an den Parametern eines DTOs nur an einer Stelle vorgenommen werden müssen, anstatt an allen Stellen, an denen die Methode aufgerufen wird.

Cognitive Complexity too high - `TodoDateChecker`

Die Methode `checkReminder` in der Klasse `TodoDateChecker` weist den Code Smell `Cognitive Complexity of methods should not be too high` auf, welcher bspw. von SonarLint analysiert wird. Dieser Code Smell bezieht sich darauf, dass die Methode zu komplex ist und schwer zu verstehen sein kann, was die Lesbarkeit und Wartbarkeit des Codes beeinträchtigt.

```
public void checkReminder() {
    for (User user : userService.findAllUsers()) {
        for (Todo todo : todoService.findNotFinishedTodosForUser(user)) {
            LocalDate date = LocalDate.now();
            if (date.isAfter(todo.getReminderDate().getDate())) {
                for (Notification notification :
                    notificationService.findAllForUser(user)) {
                    notification.notify("Das To-Do \"" + todo.getName() + "\" wurde
                                        nicht zum eingetragenen Datum, dem
                                        " + todo.getDueDate().formatDate()
                                        + ", abgeschlossen!");
                }
```

```
            }
            if (date.isAfter(todo.getDueDate().getDate())) {
                for (Notification notification :
                notificationService.findAllForUser(user)) {
                notification.notify("Das To-Do \"" + todo.getName() + "\" wurde
                                    nicht zum eingetragenen Datum, dem " +
                                    todo.getDueDate().formatDate() + ",
                                    abgeschlossen!");
                }
            }
        }
    }
}
```

In der Methode werden mehrere verschachtelte Schleifen verwendet, um durch alle Benutzer und deren unvollständigen To-Dos zu iterieren. Innerhalb dieser Schleifen werden verschiedene Überprüfungen auf das Fälligkeitsdatum und das Erinnerungsdatum jedes To-Dos durchgeführt. Abhängig von diesen Überprüfungen werden Benachrichtigungen für den Benutzer generiert.

Die Verschachtelung von Schleifen und die wiederholte Verwendung ähnlicher Logik führen zu einem hohen kognitiven Arbeitsaufwand, um das Verhalten der Methode zu verstehen. Dadurch wird es schwieriger, Fehler zu finden, Änderungen vorzunehmen und den Code zu warten.

Um diesen Code Smell zu beheben, könnte die Methode aufgeteilt werden, um die Verantwortlichkeiten besser zu separieren und die Komplexität zu reduzieren. Dies könnte beispielsweise durch die Extraktion von Methoden erfolgen, die spezifische Teile der Logik behandeln, wie die Überprüfung des Erinnerungsdatums oder die Generierung von Benachrichtigungen. Durch diese Aufteilung könnte die Methode klarer strukturiert und leichter verständlich gemacht werden, was die Lesbarkeit und Wartbarkeit verbessern würde.

4.2 Refactorings

Long Class - `TodoController`

Durch die Aufteilung der Funktionalitäten in separate Controller-Klassen, wie den `TodoListController`, den `TodoController` und den `NotificationController`, wurde der Code Smell "Long Class" in der `TodoController`-Klasse erfolgreich behoben. Jeder Controller ist nun für spezifische Aufgaben verantwortlich, was zu einer klareren Struktur und einem leichter verständlichen Code führt.

Die Verwendung dieser spezialisierten Controller hat nicht nur die Klassen schlanker und besser strukturiert gemacht, sondern auch die Kohäsion erhöht und die Kopplung zwischen den Komponenten verringert. Dadurch wird die Flexibilität und Erweiterbarkeit der Anwendung verbessert, da Änderungen oder Erweiterungen an einem bestimmten Bereich der Anwendung isoliert durchgeführt werden können, ohne andere Teile der Anwendung zu beeinflussen.

Insgesamt hat dieses Refactoring zu einem saubereren und wartungsfreundlicheren Code geführt, der nun einfacher zu pflegen und zu erweitern ist.

Refactoring in Commit: Refactored TodoController by splitting into multiple controller classes

Cognitive Complexity too high – `TodoDateChecker`

Nach dem Refactoring wurde der Code in der `checkReminder`-Methode in der Klasse `TodoDateChecker` neu strukturiert, um den Code Smell `Cognitive Complexity of methods should not be too high` zu beheben. Dabei wurde das Strategy-Pattern geschickt eingesetzt, um die Verantwortlichkeiten zur Überprüfung und Benachrichtigung in die Strategy-Klassen auszulagern.

Die neue Methode `checkTodoDates` verwendet eine Liste von `ReminderStrategy`-Objekten, die verschiedene Überprüfungsstrategien repräsentieren. In den Schleifen über alle Benutzer und deren To-Dos wird nun für jedes To-Do die `checkDate`-Methode jeder `ReminderStrategy` aufgerufen. Dadurch wird die Logik zur Überprüfung und Benachrichtigung auf die entsprechenden Strategy-Klassen ausgelagert.

Durch die Anwendung des Strategy-Patterns wurde die Komplexität der `checkReminder`-Methode reduziert, da die Verantwortlichkeiten klarer verteilt sind und die einzelnen Schritte besser voneinander abstrahiert wurden. Die Methode ist nun einfacher zu verstehen, da sie sich auf die Koordination der verschiedenen Strategien beschränkt, während die spezifischen Details der Überprüfung und Benachrichtigung in den Strategy-Klassen gekapselt sind.

Dieses Refactoring hat nicht nur die Lesbarkeit und Wartbarkeit des Codes verbessert, sondern auch die Flexibilität erhöht, da neue Überprüfungsstrategien einfach hinzugefügt werden können, ohne die `TodoDateChecker`-Klasse groß ändern zu müssen.

Refactoring in Commit: Reduced method complexity in TodoDateChecker by introducing domain

5 Entwurfsmuster

5.1 Observer-Pattern

Das Observer-Pattern wird in der Anwendung konkret eingesetzt, um automatische Benachrichtigungen von Nutzern bei Änderungen an To-Do-Listen und To-Dos zu ermöglichen. Dabei dienen die Klassen `Observable` und `TodoObserver` als Basis. Die `TodoObserver`-Schnittstelle definiert die Methoden, die Beobachter implementieren müssen, während die `Observable`-Klasse eine Liste von Beobachtern aggregiert und Methoden zum Hinzufügen und Entfernen von Beobachtern bereitstellt.

Begründung

Im `TodoController` wird das Observer-Pattern implementiert, indem die Klasse `Notifications` das `TodoObserver`-Interface implementiert und der `TodoController` von `Observable` erbt. Nach jeder erfolgreich ausgeführten Aktion, wie dem Hinzufügen oder Löschen von To-Dos, benachrichtigt der Controller die `Notifications`, die dann die Nachricht über den WebHook an den verlinkten DiscordServer sendet.

Vor- und Nachteile

Die Verwendung des Observer-Patterns erleichtert das Erstellen automatischer Benachrichtigungen und gewährleistet gleichzeitig eine lose Kopplung zwischen dem `TodoController` und den `Notifications`, was den Code flexibler und wartbarer macht. Die Einführung des Musters erfordert jedoch, dass neue Entwickler das Muster verstehen müssen, um das Verhalten der Anwendung zu verstehen.

5.2 Strategy-Pattern

Das Strategy-Pattern wird in der Anwendung verwendet, um Benachrichtigungen automatisch zu versenden, wenn das Erinnerungs- oder Abschlussdatum eines To-Dos überschritten wurde. Dazu dienen das Interface `ReminderStrategy` sowie die Implementierungsklassen `ReminderDateStrategy` und `DueDateStrategy`.

Begründung

Das Strategy-Pattern wird in der Klasse `TodoDateChecker` eingesetzt, um die Methodenkomplexität gering zu halten. Die Überprüfung des Datums wird an die Strategy-Klassen delegiert, die wiederum das tatsächliche Überprüfen an die Value Objects delegieren und selbst nur für die Benachrichtigung bei Überschreitung verantwortlich sind.

Vor- und Nachteile

Die Verwendung des Strategy-Patterns erhöht die Lesbarkeit und Übersichtlichkeit des Codes, da jede Klasse gemäß dem Single Responsibility Principle nur eine Verantwortung trägt. Außerdem sorgt das Entwurfsmuster für Flexibilität und Wartbarkeit, da Funktionalitäten beliebig implementiert und hinzugefügt werden können.