DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

SIEMENS
Ingenuity for life

Technische Dokumentation des Projekts „Taskitory“

Programmentwurf für die Vorlesung Software Engineering II

für die Prüfung zum

Bachelor of Science

des Studiengangs Informatik
Studienrichtung Angewandte Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Fabian Schwickert

16. Mai 2022

Matrikelnummer 4439027

Kurs TINF19 B4

Ausbildungsfirma ..... Siemens AG, Karlsruhe

Gutachter der Studienakademie ..... Mirko Dostmann

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Problemstellung, Ziel und Aufbau

Es gibt eine Reihe von Aufgabenverwaltungs-Systemen wie z. B. Jira, Youtrack oder Microsoft Planner, die mit einem Kanban Board arbeiten. Die hier dokumentierte Klausurersatzleistung besteht in der Entwicklung einer Applikation, die eine solche Aufgabenverwaltung modelliert. Es wird keine graphische Benutzeroberfläche (GUI) entwickelt und lediglich die notwendige Anwendungslogik programmiert und über einen REST-Service (Representational State Transfer) verfügbar gemacht. Für die Implementierung des Rest-Services wird die Programmiersprache Java mit dem Package Manager „Maven“ und dem „Spring-Boot-Framework“ verwendet.

Mit der Anwendung soll es möglich sein, die Aufgaben in einem Projekt gemeinsam in einem Projekt-Team zu verwalten. Dazu müssen Benutzer Accounts anlegen, die in den Kontext gemeinsamer Projekte gebracht werden. Das Identity- und Accessmanagement (IAM) soll dabei durch einen Keycloak umgesetzt werden. Eine persistente Datenspeicherung wird mit einer Postgresql-Datenbank (DB) umgesetzt. In der DB werden die Daten des Keycloaks und ein eigenes DB-Modell gehalten. Die benannten Komponenten (Keycloak, DB und REST-Service) der Applikation sollen dockerisiert werden und mit „Docker-Compose“ konfiguriert und betrieben werden.

Um die Bedienung des REST-Services zu vereinfachen, wird eine Sammlung von Anfragen und Konfigurationen für einen HTTP-Client (Hypertext Transfer Protocol) zur Verfügung gestellt. Dazu wird eine „Collection“ für Anwendung Postman zur Verfügung gestellt.

Alle Inhalte der hier dokumentierten Klausurersatzleistung sind auf GitHub unter dem Link „https://github.com/Krayaty/Taskitory“ erreichbar.

2 Clean Architecture

2.1 Grundlagen

Unter „Clean Architecture“ versteht man einen Architektur-Ansatz für Software-Applikationen. Die Clean Architecture fußt auf der Idee, dass Applikationen in mehrere funktional abgegrenzten Schichten aufgeteilt werden können. Wie bei einer Zwiebel umschließt eine Schicht die andere und nur die äußerste Schicht ist für Benutzer der Applikation zugänglich. Die Funktionen der inneren Schichten sind über definierte Schnittstellen für die jeweils folgende Schicht zugänglich. Die äußeren Schichten sind dadurch von den inneren Schichten abhängig; umgekehrt sind die inneren jedoch nicht von den äußeren Schichten abhängig. Daher wird es schwerer den Code auf einer Schicht zu ändern, je weiter innen sie liegt. Die Clean Architecture ermöglicht durch den beschriebenen modularen Aufbau, dass die für eine Domäne und Applikation spezifische Geschäftslogik nicht geändert werden muss, wenn konkrete technische Implementierungen einer Applikation wie geändert werden müssen. Auf diese Weise kann ein Applikation langfristig mit wenig Aufwand gewartet und angepasst werden.

Im Allgemeinen werden die folgenden fünf Schichten der Clean Architecture definiert (von innen nach außen):

1. Abstraktions-Schicht:

Die Abstraction-Schicht ist der Kern einer Applikation. In dieser Schicht wird sog. domänenübergreifendes Wissen implementiert. Das meint Algorithmen und Konzepte, die nicht an die Domäne einer Applikation gebunden sind. Zudem werden hier Software-Werkzeuge und -Bibliotheken eingebunden, die auf allen folgenden Schichten verwendet werden und technologische Standards umsetzen.

2. Domänen-Schicht:

Die Domänen-Schicht umschließt die Abstraktions-Schicht und hängt daher von ihr ab. In der Domänen-Schicht wird die Fachlichkeit einer Applikation umgesetzt. Hier wird die übergreifende Geschäftslogik implementiert, die für die Abbildung einer Domäne in

vielen verschiedenen Applikation benötigt wird. Die Domänen-Schicht wird durch das Domain Driven Desing (siehe Kapitel 3) modelliert.

3. Applikations-Schicht:

Die Applikations-Schicht folgt direkt auf die Domänen-Schicht und hängt daher von ihr ab. In der Applikations-Schicht werden die Use Cases einer Applikation abgebildet. In diesem Teil einer Applikation werden die für den Benutzer erfahrbaren Funktionen implementiert. Das umfasst auch die Implementierung der Applikations-spezifischen Geschäftslogik einer Domäne.

4. Adapter-Schicht:

Die Adapter-Schicht baut auf der Applikations-Schicht auf und hängt daher von ihr ab. Der Zweck der Adapter-Schicht ist es, Benutzerinteraktionen an die Applikations-Schicht weiterzugeben. Dabei dient die Adapter-Schicht als „Anti Corruption Layer“, der Domänen- und Applikations-spezifischen Quellcode von festen technischen Implementierungen abkoppelt. Das meint auch die Konvertierung der Daten aus eingehenden Anfragen in ein für die darunter liegenden Schichten verständliches Format.

5. Plugin-Schicht:

Die Plugin-Schicht liegt ganz außen und hängt von der darunter liegenden Adapter-Schicht ab. Auf der Plugin-Schicht werden technische Implementierungen wie z. B. das Einbinden einer Datenbank oder eines IAM-Services vorgenommen. Außerdem werden auf dieser Schicht die Funktionen einer Applikation den Benutzern zugänglich gemacht.

2.2 Implementierung in der Applikation „Taskitory“

Die Applikation Taskitory verwendet das Konzept der Clean Architecture mit allen oben beschriebenen Schichten. Um die Eigenschaft der nach innen gerichteten Abhängigkeiten in der Applikation Taskitory sicherzustellen, wird für jede Schicht ein eigenes Maven- Modul angelegt. Die Maven-Module werden wie oben beschrieben über Abhängigkeiten miteinander verbunden. Nachfolgend wird auf alle Schichten und deren Aufgabenfelder eingegangen.

1. Abstraktions-Schicht:

Die Abstraction-Schicht der Applikation Taskitory enthält keinen eigenen Quellcode. Lediglich die folgenden Abhängigkeiten werden auf dieser Schicht eingebunden:

- Java Persistence API (JPA)
- Apache Commons-Lang
- Org.JSON
- Spring-Context
- Lombok

Die „Java Persistence API“ (JPA) ist ein de fakto Standard für die Integration einer Datenbank in eine Java-Applikation. Da die JPA eine weit verbreitete, beständige Software-Bibliothek ist, ist es nicht notwendig sie als Teil der Plugin-Schicht einzubinden. Denn das würde umfassende Implementierungen auf der Adapter-Schicht nach sich ziehen. Die JPA hat keinen Bezug zu der Domäne der Applikation Taskitory und kann als Standard-Technik angesehen werden. Daher gehört diese Abhängigkeit auf die Abstraktions-Schicht.

Die Software-Bibliotheken Apache Commons-Lang und Org.JSON stellen Funktionalitäten für die Verarbeitung von Inhalten zur Verfügung, die nicht von den Java Standard-Bibliotheken mitgeliefert werden. Mit Apache Commons-Lang werden allgemeine Problemstellungen wie z. B. das Hashen erledigt. Org.JSON auf die Verarbeitung von Inhalten im JSON-Format. Diese allgemeinen Funktionen können für alle folgenden Schichten relevant sein und sind nicht an eine bestimmte Domäne gebunden. Daher werden diese Abhängigkeit auf der Abstraktions-Schicht eingebunden.

Die Abhängigkeiten „Spring-Context“ und „Lombok“ sind zwei Software-Bibliotheken, deren Funktionen allen anderen Schichten zur Verfügung stehen soll. Mit Spring-Context werden Funktionen zur Verfügung gestellt, die das Betreiben einer eigenen Applikation mit dem Spring-Framework ermöglichen. Lombok ist eine Software-Bibliothek mit der Quellcode vereinfacht werden kann. Lombok ermöglicht es, Standard-Konstrukte in Java wie z. B. Getter und Setter durch eine Annotation zu ersetzen. Die Funktionen der beiden Software-Bibliotheken Spring-Context und Lombok beziehen sich auf allgemeine Pro-

grammier-Konzepte und Konstrukte der Programmiersprache Java. Daher gehören beide Abhängigkeiten auf die Abstraktions-Schicht.

2. Domänen-Schicht:

Die Domänen-Schicht ist das Herzstück der Applikation Taskitory. Hier wird die grundlegende Geschäftslogik der Domäne definiert. Es wird ein statisches Modell aufgestellt, das die fachlichen Akteure der Applikation Taskitory definiert. Das umfasst besonders die Entitäten „Benutzer“, „Projekt“, „Projekt-Mitgliedschaft“, „Aufgabe“, „Kanbanboard“, „Tag“, „Nachricht“. In Kapitel 3 werden diese Entitäten näher beschrieben. Die Applikation Taskitory kommuniziert über JPA mit einer Datenbank und persistiert so die anfallenden relevanten Daten.

Der Einsatz von JPA erfordert ein „Objektrelationales Mapping“ (OR-Mapping). Dabei werden die Tabellen einer Datenbank durch Klassen und die Datensätze durch Instanzen dieser Klassen repräsentiert. In der Applikation Taskitory müssen die auf der Domänenschicht definierten Entitäten (siehe oben) persistiert werden. Um für diese Entitäten ein OR-Mapping herstellen zu können, werden technische Implementierungen von JPA in den Domänen-Quellcode eingebracht. Es wurde bewusst entschieden das OR-Mapping auf der Domänen-Schicht umzusetzen, da eine Implementierung auf der Plugin-Schicht zu den Domänen-Klassen äquivalente Klassen erfordert hätte. Diese räumliche Trennung von Fachlichkeit und technischen Implementierungen wird nicht durch den deutlich höheren Implementierungsaufwand gerechtfertigt.

Es werden JPA-Annotationen verwendet, die dem fachlichen Quellcode der Domäne angeheftet werden. Dadurch kann der Domänen-Quellcode auch weiterhin von technischen Implementierungen getrennt betrachtet werden.

3. Applikations-Schicht:

Auf Grundlage des statischen Modells der Akteure aus der Domänen-Schicht wird in der Applikations-Schicht ein dynamisches Modell der Abläufe aufgestellt. Hier werden die Use Cases der Applikation Taskitory umgesetzt. Dabei wird der Informationsfluss von Akteur zu Akteur unter Berücksichtigung der Geschäftslogik aus der Domänen-Schicht abgebil-

det. Die Use Cases werden zudem durch Applikations-spezifische Geschäftslogik gelenkt. Die wichtigsten Use Cases der Applikation Taskitory sind in Kapitel 4 beschrieben.

4. Adapter-Schicht:

Die Applikation Taskitory benutzt für Benutzerinteraktionen eine REST-Schnittstelle. Es gibt kein Frontend, das die Validierung von Anfrage-Parametern übernimmt. Daher müssen die eingehenden HTTP-Anfragen in dem Backend validiert und in passende Formate umgewandelt werden, bevor sie an die inneren Schichten der Applikation weitergegeben werden. Für diesen Zweck ist die Adapter-Schicht gedacht. Hier befindet sich der Quellcode, mit dem die HTTP-Anfragen der REST-API in Methodenaufrufe und Datentransferobjekte (DTO) der Applikationsschicht umgewandelt werden. Dieser Umwandlungs-Prozess findet unter Beachtung der Validierungs-Logik statt.

5. Plugin-Schicht:

Die Plugin-Schicht der Applikation Taskitory enthält die technischen Implementierungen für die Kommunikation mit einer Postgresql Datenbank und einem Keycloak IAM-Server. Dazu wird das Persistenz-Framework „Hibernate“ und das Sicherheits-Framework „Spring-Security“ verwendet. Zudem werden die Funktionen der Applikation Taskitory auf dieser Schicht den Benutzern über eine REST-Schnittstelle verfügbar gemacht.

3 Domain Driven Design

3.1 Problemdomäne

Die Problemdomäne einer Applikation setzt sich aus einer Kerndomäne, mehreren unterstützenden und mehreren generischen Domänen zusammen. Die Kerndomäne bestimmt die zentralen Funktionalitäten und den Zweck einer Applikation. Unterstützenden Domänen beziehen sich auf Aufgaben-Felder, die nicht direkt Teil der Kerndomäne sind, aber die Kerndomäne mit ihren Funktionen unterstützen. Die Kerndomäne verwendet die unterstützenden Domänen, um die zentralen Funktionen einer Applikation umzusetzen. Die Generischen Domänen beziehen sich auf Aufgabenfelder, die nicht direkt mit dem Zweck einer Applikation in Verbindung stehen, jedoch für das Betreiben einer Applikation not-

wendig sind. Die Generischen Domänen liefern Basis-Funktionalitäten, ohne die eine Applikation nicht lauffähig ist.

Die Problemdomäne der Applikation „Taskitory“ ist die Aufgabenverwaltung in Projekten mit mehreren Projekt-Teilnehmern. Dabei ist die Kerndomäne das Verwalten von Aufgaben. Das Umfasst nicht nur das Erstellen, Verändern und Löschen von Aufgaben, sondern auch die Funktionen eines Kanban Boards.

Die über die Kerndomäne zur Verfügung gestellten Daten und Funktionen werden Benutzern der Applikation über die Domäne IAM zur Verfügung gestellt. Ohne eine Benutzerverwaltung könnte das Konzept von Projekt-Teams und damit die ganze Applikation nicht umgesetzt werden. IAM hat keine inhaltliche Verbindung zu der Kerndomäne und wird daher als generische Domäne eingestuft.

Die Verwaltung von Projekten ist kein zwingend notwendiger Aufgabenbereich für eine Aufgabenverwalten. Allerdings ist dieser Aufgabenbereich bei der Applikation „Taskitory“ relevant für die Funktionen der Kerndomäne und inhaltlich mit dieser verbunden. Denn alle Objekte, die in der Kerndomäne verwendet, erzeugt oder gelöscht werden, sind einem Projekt zugeordnet. Auch die Rolle eines Benutzers in einem Projekt kann in diese Domäne gezählt werden. Denn der Zugriff auf die Funktionen der Kerndomäne wird durch die Rolle eines Benutzers in einem Projekt eingeschränkt. Auf Grund der inhaltlichen Verbindung zur Kerndomäne wird die Verwaltung von Projekten und der damit assoziierten Projekt-Teams als unterstützende Domäne eingestuft.

Das Erstellen von Statistiken ist keine für eine Aufgabenverwaltung notwendige Funktion. Allerdings werden mit Statistiken die Daten von Kanban Boards verarbeitet, die aus der Kerndomäne stammen. Es besteht damit eine klare inhaltliche Verbindung zur Kerndomäne. Das Erstellen von Statistiken ist damit eine unterstützende Domäne.

Die Applikation „Taskitory“ informiert Benutzer mit Nachrichten über bestimmte Ereignisse, die für den Benutzer relevant sind. Diese Ereignisse betreffen i. d. R. nicht die Kerndomäne, sondern unterstützende Domänen. Es besteht daher keine inhaltliche Verbindung zur Kerndomäne. Das Messaging ist eher eine Basisfunktionalität und wird daher als

generische Domäne eingestuft. Abbildung 1 zeigt noch einmal die Problemdomäne der Applikation „Taskitory“ mit allen Bestandteilen.

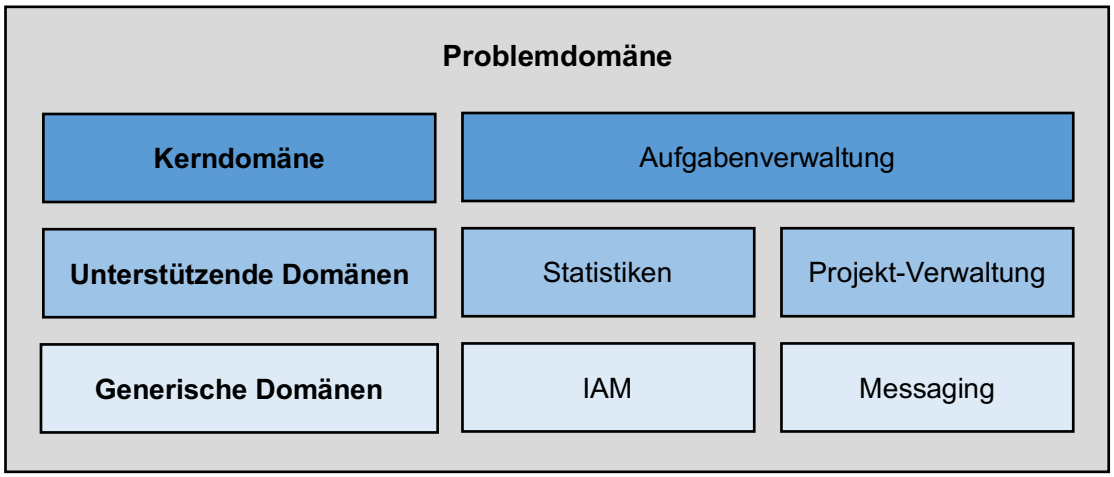


Abb. 1: Die Problemdomäne der Applikation „Taskitory“

3.2 Ubiquitous Language

Die „Ubiquitous Language“ ist eine Sammlung von Begriffen, mit denen eine Applikation wird. Das umfasst Fachbegriffe der Problemdomäne und der technischen Implementierung. Es müssen nicht triviale Objekte, Konzepte und Regeln eindeutig definiert werden.

3.2.1 Begriffe der Fachlichkeit

Die Ubiquitous Language der Applikation Taskitory umfasst die folgenden Begriffe für das Beschreiben problemdomänenspezifischer Objekte, Konzepte und Regeln.

1. Projekt

Ein Projekt ist ein zielgerichtetes Vorhaben, das aus einer Menge von Aufgaben besteht, die von einer Gruppe von Benutzern verwaltet und bearbeitet werden. Ein Projekt hat eine eindeutige Bezeichnung und eine Beschreibung, die von Projekt-Administratoren festgelegt und verändert werden kann. Für die Erfüllung dieser Aufgaben, stellen Projekte Kanbanboards und Tags zur Verfügung. Projekte sind der zentrale Kontext, in dem ein Benutzer die Applikation Taskitory verwendet.

Über die Zeit können Benutzer ein Projekt verlassen und wieder beitreten. Für das Verlassen eines Projekts wird keinerlei Zustimmung von Seiten des Projekt-Teams benötigt. Allerdings kann ein Benutzer einem Projekt nur dann beitreten, wenn er von einem Projekt-Administrator eingeladen wird.

2. Projekt-Einladung

Eine Einladung in ein Projekt kann nur von einem Projekt-Administrator ausgestellt werden. Eine Einladung ist eine Instanz der Nachrichten-Entität und ist daher genau einem Projekt und einem Benutzer zugeordnet. Eine Einladungs-Nachricht enthält den geheimen Projekt-Schlüssel in gehashter Form und die Rolle, die der Benutzer im Projekt einnehmen soll.

3. Projekt-Schlüssel

Der Schlüssel eines Projekts ist eine geheime Zeichenkette, die nur von Projekt-Administratoren verwaltet werden kann. Der Schlüssel eines Projekts ist nur gehasht einsehbar und kann ausschließlich auf einen zufällig generierten Wert geändert werden.

4. Benutzer

Ein Benutzer wird durch ein Benutzer-Konto repräsentiert. Dieses Benutzer-Konto besteht aus einem Benutzernamen und einem Passwort. Beide Werte können durch den Benutzer selbst verändert werden.

Benutzer müssen keinem Projekt angehören, können aber beliebig vielen Projekten beitreten. Benutzer können die Applikation Taskitory ohne eine Projektmitgliedschaft nicht sinnvoll nutzen. Daher können Benutzer jederzeit neue Projekte erstellen, in denen sie automatisch als Projekt-Administrator eingetragen werden.

Es werden der Beginn von Projektmitgliedschaften und die eingenommene Rolle in einem Projekt festgehalten. Benutzer können die Rollen „Mitglied“ und „Admin“ einnehmen.

5. Projekt-Administrator

Projekt-Administratoren sind Benutzer mit der Admin-Rolle. Diese Rolle ist projektspezifisch. Benutzer mit der Admin-Rolle haben alle Rechte eines Mitglieds und zusätzliche Rechte für die Projekt-Verwaltung.

Es muss zu jeder Zeit einen Administrator in einem Projekt geben. Wenn der letzte Administrator ein Projekt verlässt, muss er einen neuen Projekt-Administrator aus dem Projekt-Team bestimmen. Wenn es keine anderen Team-Mitglieder gibt, wird das Projekt gelöscht. Es gibt keine Obergrenze für die Anzahl von Projekt-Administratoren. Nur Projekt-Administratoren können Benutzer einladen, entfernen und deren Rolle ändern.

6. Projekt-Mitglieder

Projekt-Mitglieder sind Benutzer mit der Mitglieds-Rolle. Diese Rolle ist projektspezifisch. Die Projekt-Mitgliedschaft ermächtigt einen Benutzer die Aufgaben und Kanbanboards eines Projekts zu verwalten.

7. Aufgabe

Aufgaben sind Teil eines Projekts und beschreiben eine Tätigkeit, die durchgeführt werden soll. Bei Erfüllung aller Aufgaben wird ein übergeordnetes Projekt-Ziel erreicht.

Aufgaben sind einem Projekt zugeordnet und haben eine im Projekt eindeutige Bezeichnung und eine nicht eindeutige Beschreibung. Über einen Erstellungs- und einen Fertigstellungs-Zeitpunkt wird der Bearbeitungszeitraum einer Aufgabe festgehalten. Der Bearbeitungs-Status einer Aufgabe ist eng damit verbunden. Der Bearbeitungs-Status kann die Werte der Spalten eines Kanbanboards (siehe unten) annehmen. Die Komplexitätund die Priorität in der Bearbeitungsreihenfolge werden jeweils über einen eigenen Wert festgehalten. Eine Aufgabe enthält Informationen über den Autor der Aufgabe und die für die Bearbeitung aktuell zuständige Person. Die Aufgaben werden neben diesen Attributen auch durch Tags mit Meta-Informationen beschrieben.

Aufgaben sind immer einem Projekt zugeordnet. Aufgaben können sich entweder auf einem oder keinem Kanbanboard befinden. Wenn eine Aufgabe nicht auf einem Kannbanboard zu finden ist, liegt sie im Backlog. Aufgaben können entweder direkt auf einem Kanbanboard erzeugt werden oder vorerst im Backlog abgelegt werden.

8. Backlog

Das Backlog eines Projekts ist die Sammlung aller Aufgaben, die keinem Kanbanboard zugeordnet sind. Aus dem Backlog können Aufgaben einem beliebigen Kanbanboard des

gleichen Projekts zugewiesen werden. Aufgaben, die von einem Kanbanboard entfernt werden, werden automatisch in das Backlog zurückgelegt.

9. Kanban Board

Eine Kanbanboard ist eine Tabelle, bei der jede Spalte einen Bearbeitungs-Status repräsentiert. Aufgaben können in diese Spalten einsortiert werden, um den Bearbeitungs-Status der Aufgabe zu visualisieren. Um einen Fortschritt abzubilden, werden die Aufgaben von einer in eine andere Spalte geschoben.

Kanbanboards sind einem Projekt zugeordnet und haben eine im Projekt eindeutige Bezeichnung und eine nicht eindeutige Beschreibung. Ein Kanban Board betrifft einen bestimmten „Sprint“ genannten Zeitraum. Dieser Zeitraum wird durch einen Start- und End-Zeitpunkt festgehalten. Nach einem Sprint besteht das Kanban Board weiter, kann aber nicht mehr bearbeitet werden. Ein altes Kanban Board kann in ein neues Kanban Board überführt werden, das bearbeitet werden kann.

Kanbanboards können grundsätzlich eine beliebige Anzahl Spalten haben. Weit verbreitet sind die folgenden fünf Spalten (Die Spalten werden in der unten genutzten Reihenfolge auf einem Kanbanboard angeordnet).

- Todo

  In dieser Spalte werden alle Aufgaben abgelegt, die sich nicht aktuell in Bearbeitung befinden. Aufgaben haben bei ihrer Erzeugung diesen Status. In der Applikation Taskitory ist diese Spalte immer sichtbar.

- In Progress

  In dieser Spalte werden alle Aufgaben abgelegt, die aktuell bearbeitet werden. In der Applikation Taskitory ist diese Spalte immer auf einem Kanbanboard sichtbar.

- Review

  In dieser Spalte werden alle Aufgaben abgelegt, die bereits bearbeitet wurden und deren Ergebnis durch eine verantwortliche Person fachlich abgenommen werden muss. In der Applikation Taskitory muss diese Spalte nicht auf einem Kanbanboard sichtbar sein. Die Spalte kann ein- und ausgeblendet werden. Dabei werden die in dieser Spalte

befindlichen Aufgaben nicht entfernt. Allerdings können Aufgaben in einer ausgeblendeten Spalte nicht verwaltet werden.

- Testing

Diese Spalte enthält alle Aufgaben, die bereits bearbeitet wurden und noch technischen Tests unterzogen werden. In der Applikation Taskitory muss diese Spalte nicht auf einem Kannbanboard sichtbar sein. Die Spalte kann ein- und ausgeblendet werden. Dabei werden die in diesen Spalten befindlichen Aufgaben nicht entfernt. Allerdings können Aufgaben in einer ausgeblendeten Spalte nicht verwaltet werden.

- Done

Diese Spalte enthält alle Aufgaben, die abgeschlossen wurden. In der Applikation Taskitory muss diese Spalte immer auf einem Kanbanboard vorhanden sein.

10. Sprint

Der Begriff „Sprint“ entstammt den agilen Vorgehensmodellen. Sprints sind feste Zeiträume, in die ein Projekt eingeteilt wird. In diesen Zeiträumen nimmt sich das Projekt-Team bestimmte Aufgaben vor, die mit Hilfe eines Kanban Boards verwaltet werden können. Nach Beendigung eines Sprints werden die Arbeitsergebnisse Qualitätsgesichert. Auf diese Weise wird ein Produkt iterativ weiterentwickelt und kann mit den Anforderungen abgeglichen werden.

11. Statistiken

Es sollen einfache statistische Analysen auf Kanbanboards durchgeführt werden können. Ein Beispiel ist die Berechnung des Anteils der Aufgaben auf einem Kanban Board, der zum Ende des Sprints in einem bestimmten Bearbeitungs-Status ist.

12. Tag

Ein Tag ist ein Etikett, das einer Aufgabe angeheftet werden kann, um diese Aufgabe mit Meta-Informationen zu versehen. Tags bestehen aus einer global eindeutigen Bezeichnung und einer nicht eindeutigen Beschreibung. Tags sind seinem Erzeuger (Benutzer) zugeordnet.

13. Nachricht

Eine Nachricht dient der Übermittlung von für einen Benutzer relevanten Informationen zu einem Projekt, dem er angehört. Solche relevanten Informationen sind z. B. der Rauswurf aus einem Projekt, die Änderung der eigenen Rolle oder das Löschen des Projekts.

Nachrichten werden nicht von einem Benutzer verschickt und können daher nicht dazu genutzt werden, um eine Konversation zu führen. Nachrichten werden von einem Projekt versendet und von einem Benutzer empfangen. Der Inhalt von Nachrichten kann nicht personalisiert werden und ist für den jeweiligen Anwendungsfall vorbereitet. Über ein Attribut können Nachrichten als „gelesen“ markiert werden.

3.2.2 Begriffe der technischen Implementierung

Die Ubiquitous Language der Applikation Taskitory umfasst die folgenden Begriffe für Objekte, Konzepte und Regeln der technischen Implementierung.

1. Backend

Der Begriff „Backend“ meint im Umfeld der Applikation Taskitory die Java-Anwendung, in der die Geschäftslogik, die Datenbank- und IAM-Anbindung sowie die REST-Schnittstelle der Applikation Taskitory implementiert ist. Das steht im Widerspruch zu der eigentlichen Bedeutung des Begriffs, da Benutzer direkt auf das Backend zugreifen. Alle anderen Eigenschaften sind jedoch typisch für ein Backend. Wenn zukünftig die Applikation Taskitory durch ein Frontend mit GUI erweitert wird, kann der direkte Zugriff der Benutzer auf das Backend verhindert werden. Dann handelt es sich um ein echtes Backend.

2. Universally Unique Identifier (UUID)

„Universally Unique Identifier“ (UUID) sind 128-Bit-Zahlen, die für die Identifikation von Objekten besonders in Informationstechnik-Systemen verwendet werden. UUIDs gelten als praktisch global eindeutig, da die Anzahl möglicher UUIDs sehr groß ist.

3. String

„String“ ist ein komplexer Datentyp von Java. Variablen von diesem Typ können mit endlichen unterschiedlich langen Zeichenketten befüllt werden. Strings werden dazu verwendet Text zu repräsentieren.

4. Final (Unveränderlichkeit)

Bestimmte Strukturen in Java wie z. B. Variablen können mit dem Schlüsselwort „final“ deklariert werden. Dadurch können die Werte der Variablen nach der Initialisierung nicht mehr verändert werden.

5. Zeitstempel

Zeitstempel ist ein komplexer Datentyp für die Repräsentation eines Zeitpunkts. Dabei wird i. d. R. die Zeitspanne seit einem bestimmten Zeitpunkt – oft 01.01.1970 00:00 – gespeichert. Der Wert des Zeitstempels ergibt sich durch die Addition der Zeitspanne zu diesem Ausgangspunkt. Der Wert kann in verschiedenen Formaten dargestellt werden.

6. Representational State Transfer (REST)

Representational State Transfer (REST) ist ein Konzept für die Architektur von verteilten Software-Systemen. Ein REST-Service stellt Hypermedia-Ressourcen über HTTP im Internet zur Verfügung. Hypermedia-Ressourcen sind Informationen, die nicht auf ein Medium festgelegt sind und sich aus mehreren per Hyperlink verbundenen Daten-Objekten zusammensetzen. Eine solche Ressource wird über eine global eindeutige Adresse und eine Zugriffsmethode (HTTP-Verben) zugänglich gemacht.

7. HTTP

Das Hypertext Transfer Protocol ist ein zustandsloses Protokoll zur Übermittlung von Daten in einem Netzwerk. HTTP wird für den Zugriff auf die REST-API der Applikation Taskitory und innerhalb für die Kommunikation der Applikationskomponenten verwendet.

8. Java Persistence API (JPA)

Die „Java Persistence API“ (JPA) ist ein de fakto Standard für die Integration verschiedener Datenbanken in eine Java-Applikation. Durch das Konzept vom Objektrelationalen Mapping (OR-Mapping) können in einer Applikation erzeugte Objekte ohne großen Programmieraufwand direkt in einer Datenbank persistiert werden.

9. Objektrelationales Mapping

Objektrelationales Mapping bezeichnet eine Technologie, mit der die Objekt-Struktur einer objektorientierten Programmiersprache auf die Relationen-Struktur eines relationalen Datenbank-Systems abgebildet wird. Dabei werden Tabellen mit Spalten durch

Klassen mit Attributen repräsentiert. Die Datensätze entsprechen im OR-Mapping den Instanzen der Klassen.

10. IAM-/Keycloak-Server

Unter Identity und Access Management (IAM) werden allgemein die Techniken verstanden, mit denen Benutzer einer Software-Applikation authentifiziert und autorisiert werden. Das umfasst die Konfiguration von Authentifizierungs- sowie Autorisierungsmethoden, Rechtegruppen uvm.

Ein IAM-Server ist im Kontext der Applikation Taskitory einer Anwendung, die diese IAM-Funktionen für eine andere Applikation übernimmt. Für die Applikation Taskitory wird dabei konkret die Applikation „Keycloak“ verwendet. Diese Applikation speichert ihre Daten in der gleichen Datenbank, in der auch das Backend Daten persistiert.

11. Datenbank/Postgresql

Eine Datenbank ist eine Software-Anwendung, die für die persistente Speicherung von Daten verwendet wird. Die Daten werden dabei hauptsächlich von anderen Anwendungen zur Verfügung gestellt. Mit der Structured Query Language (SQL) können die Datenbestände einer DB bearbeitet werden. Für die Applikaiton Taskitory wird eine Postgresql-DB verwendet.

12. Docker-Container

Docker ist eine Software-Applikation, mit der andere Software-Applikationen in „Containern“ betrieben werden können. Diese Container sind virtualisierte Laufzeitumgebungen für Software-Applikation, die mit den Ressourcen eines Gastgeber-Systems betrieben werden. Mit „Dockerfiles“ können diese Laufzeitumgebungen konfiguriert und auf verschiedenen Computern exakt wiederhergestellt werden. Dadurch können ganze Applikationen schnell auf andere Server bewegt und wieder hergestellt werden. Um den Zustand einer Applikation zu speichern und in einem neuen Container wieder herzustellen, werden „Volumes“ verwendet. Volumes sind Verzeichnisse des Gastgeber-Systems, die in einen Container gemounted werden können. Die Verzeichnisse des Gastgeber-Systems sind unabhängig von den Containern und werden daher nicht durch das Löschen einer

Container-Instanz geleert. Der Zustand einer Applikation bleibt in solchen Volumes für neue Container-Instanzen erhalten.

13. Docker Compose

Docker-Compose ist eine Technologie, mit der mehrere Docker-Container zusammen verwaltet werden können. Das umfasst die Konfiguration der einzelnen Container, aber auch die Kommunikationswege der Container untereinander und nach außen. Auf diese Weise können mehrere Container zu einem System verbunden werden. Mit einer „Docker-Compose.yml“ Datei können solche Systeme beschrieben werden.

Durch einbinden einer „.env“ Datei ist es möglich sensible Informationen wie Passwörter aus dem Docker-Compose.yml und anderen Applikations-internen Dateien in Umgebungsvariablen auszulagern.

14. Postman Collection

Postman ist eine Software-Applikation mit der HTTP-Anfragen gezielt konfiguriert, versendet und empfangen werden können. Eine „Collection“ ist eine Sammlung von HTTP-Anfragen, die mit einer Verzeichnis-Struktur verwaltet und durch übergreifende Konfigurationen wie z. B. die Authentifizierung von HTTP-Anfragen erweitert werden. Für die Applikation Taskitory wird eine solche Collection mit zentralen HTTP-Anfragen erstellt.

15. Hashing

Beim Hashen einer Zeichenfolge wird eine Einweg-Hash-Funktion auf diese Zeichenfolge angewendet. Hash-Funktionen bilden Zeichenfolgen beliebiger Länge auf Zeichenfolgen mit einer festen Länge ab. Der erzeugte Hash-Wert einer Hash-Funktion darf praktisch nicht auf seinen Ausgangswert zurückführbar sein. Zudem darf es in annehmbarer Zeit nicht möglich sein, zwei Zeichenketten zu finden, die den gleichen Hash-Wert haben.

3.3 Taktische Muster des Domain Driven Design

Die taktischen Muster des Domain Driven Desings werden dazu verwendet ein möglichst präzises und gut verständliches Modell der Fachlichkeit einer Applikation zu erstellen. Dafür werden fünf Muster definiert, die unterschiedliche Aufgabenbereich haben. Nach-

folgend wird auf die fünf taktischen Muster des Domain Driven Designs mit ihren Aufgabenbereichen und der Umsetzung in der Applikation Taskitory eingegangen.

3.3.1 Entitäten

Entitäten sind Objekte der Domäne, die nicht durch ihre Attribute, sondern durch ihre Identität unterschieden werden können. Die Identität wird dabei i. d. R. durch ein Schlüssel-Attribut dargestellt.

In der Problemdomäne der Applikation Taskitory interagieren die Akteure Projekt, Projekt-Mitgliedschaft, Nachricht, Kanbanboard, Aufgabe, Benutzer, Tag. Daher sind all diese Akteure Kandidaten für Domänen-Entitäten. Die benannten Akteure enthalten wichtige Informationen, die in der DB persistiert werden müssen. Dazu werden die Akteure mit einem JPA-OR-Mapping versehen. Dadurch enthalten alle Akteure ohnehin einen DB-Primärschlüssel. Allerdings muss das nicht heißen, dass ein Akteur auch eine Entität ist. Daher wird nachfolgend überprüft, welche Akteure auch Entitäten sind.

1. Projekt

Projekte werden durch eine eindeutige Bezeichnung identifiziert. Diese Bezeichnung ist der Primärschlüssel der zugehörigen DB-Entität. Es handelt sich um einen natürlichen Schlüssel, der die Identität eines Projekts repräsentiert. Die Beschreibung und das Projekt-Team eines Projekts repräsentieren nicht die Identität eines Projekts. Es kann mehrere Projekte mit gleichen Werten für diese Attribute geben.

Der geheime Projekt-Schlüssel ist ein zusätzlicher Schlüsselkandidat, da dieser Projekt-Schlüssel praktisch global eindeutig ist. Allerdings ist eine Dopplung theoretisch möglich, da der Projekt-Schlüssel zufällig generiert wird. Die Wahrscheinlichkeit, dass eine Dopplung auftritt, ist bei 36^{128} möglichen Projekt-Schlüsseln allerdings marginal.

Projekte bestehen neben beschriebenen Attributen wie der Bezeichnung und einer Beschreibung aus Kanban Boards, Aufgaben, Nachrichten und Projekt-Mitgliedschaften. Diese anderen Akteure sind unveränderbar diesem einen Projekt zugeordnet und können in keinem anderen Projekt auftreten. Durch die Kanban Boards, Aufgaben, Nachrichten und Projekt-Mitgliedschaften könnte ein Projekt identifiziert werden.

Allerdings sind alle Attribute eines Projekts mit Ausnahme der Bezeichnung veränderbar und repräsentieren den Lebenszyklus dieses Projekts. Daher eignet sich die Bezeichnung am besten als Primär-Schlüssel

Projekte unterscheiden sich durch ihre Identität in Form einer eindeutigen Bezeichnung. Zudem unterliegen Projekte einer ständigen Veränderung. Daher ist ein Projekt eine Domänen-Entität.

2. Projekt-Mitgliedschaft

Eine Projekt-Mitgliedschaft verbindet ein Projekt mit einem Benutzer. In der DB wird eine Projekt-Mitgliedschaft durch eine n:m-Beziehung abgebildet. Eine Projekt-Mitgliedschaft wird daher eindeutig durch eine Kombination von Projekt-Bezeichnung und Benutzer-ID identifiziert. Benutzer sollen maximal eine Mitgliedschaft in einem Projekt besitzen. Es soll nicht gespeichert werden, ob und wie oft ein Benutzer einem Projekt beigetreten ist und es wieder verlassen hat. Daher wird der Zeitstempel des Starts der Mitgliedschaft nicht als Teil des Primärschlüssels verwendet.

Der Start der Mitgliedschaft wird neu gesetzt, wenn der Benutzer befördert wird. Eine Beförderung meint das Ändern der zugeordneten Rolle von „Mitglied“ zu „Admin“. Projekt-Mitgliedschaften bestehen also aus einem zusammengesetzten Primärschlüssel (Projekt-Bezeichnung, User-ID) der die Identität der Projekt-Mitgliedschaft ausmacht und zwei beschreibenden Attributen, die während des Lebenszyklus einer Projekt-Mitgliedschaft verändert werden können.

Projekt-Mitgliedschaften unterscheiden sich durch ihre Identität in Form einer Zuordnung eines Benutzers zu einem Projekt. Zudem können Projekt-Mitgliedschaften über die Zeit verändert werden. Daher ist eine Projekt-Mitgliedschaft eine Domänen-Entität.

3. Nachricht

Nachrichten haben einen Absender und einen Empfänger. Der Absender kann bei der Applikation Taskitory nur ein Projekt sein. Der Empfänger ist immer ein Benutzer. Auch dieser Akteur verbindet die Akteure Projekt und Benutzer. Allerdings werden Nachrichten nicht durch ihre Projekt- und Benutzer-Zugehörigkeit identifiziert. Es kann viele ver-

schiedene Nachrichten geben, die von dem gleichen Projekt an den gleichen User verschickt werden.

Nachrichten haben einen Typ und einen Inhalt, die zusammengenommen die Bedeutung einer Nachricht ausmachen. Allerdings können auch diese Attribute nicht zur Identifizierung einer Nachricht verwendet werden. Denn es können mehrere Nachrichten des gleichen Typs mit dem gleichen Inhalt von dem gleichen Projekt an den gleichen Benutzer verschickt werden.

Durch Hinzufügen des Empfangszeitpunkts einer Nachricht zu den bisher betrachteten Attributen könnte ein Primärschlüssel entstehen. Praktisch scheint es sehr unwahrscheinlich, dass zwei Nachricht mit gleichen Werten für diese fünf Parameter vorliegen. Theoretisch ist das allerdings kein Problem.

Abschließend bleibt noch das Attribut, mit dem eine Nachricht als gelesen markiert wird. Dieses Attribut liefert keine Möglichkeit zur Identifizierung einer Nachricht. Hinzu kommt, dass sich dieser Status einer Nachricht während des Lebenszyklus einer Nachricht ändern kann. Alle anderen Eigenschaften einer Nachricht können nicht verändert werden.

Die geschilderten Eigenschaften von Nachrichten, sprechen nur eingeschränkt dafür, dass Nachrichten Domänen-Entitäten sind. Dass nahezu alle Attribute einer Nachricht unveränderlich und notwendig sind, um eine Nachricht zu identifizieren, spricht eher für ein Value Object. Allerdings können Nachrichten zur Laufzeit mit dem Attribut „read“ (zu Deutsch „gelesen“) verändert werden. Zudem ist dieses Attribut nicht relevant für die Identifizierung von Nachrichten. Daher werden Nachrichten in der Applikation Taskitory als Entität definiert. Um Nachrichten zu identifizieren wird eine UUID (Surrogatschlüssel) als Primärschlüssel-Attribut verwendet.

4. Kanban Board

Kanban Boards besitzen eine Menge von beschreibenden Attributen. Dazu zählen eine für ein Projekt eindeutige Bezeichnung, eine Beschreibung, ein Sprint-Zeitraum sowie zwei Attribute für das Ein- und Ausblenden von der Review- und Testing-Spalte eines Kanban

Boards. Alle diese Attribute können allein nicht zur Identifizierung von Kanban Boards verwendet werden.

Kanban Boards sind immer einem Projekt unveränderlich zugeordnet. Dieses Projekt kann zusammen mit der Bezeichnung zur Identifizierung eines Kanban Boards verwendet werden. Allerdings kann die Bezeichnung eines Kanban Boards genau wie die anderen beschreibenden Attribute verändert werden.

Eine weitere Möglichkeit zur Identifizierung eines Kanban Boards könnte die Sammlung der dem Kanban Board zugewiesenen Aufgaben sein. Allerdings kann auch diese Sammlung von Aufgaben sich verändern.

Letztlich bleibt nur noch die Möglichkeit zur Identifizierung eines Kanban Boards mittels eines Surrogatschlüssels in Form einer UUID. Da die Identität eines Kanban Boards von einem solchen Schlüssel abhängt und Kanban Boards einen Lebenszyklus mit Veränderungen durchläuft, werden Kanban Boards als Entität definiert.

5. Aufgabe

Aufgaben sind die zentralen Träger von Informationen in der Applikation Taskitory und bestehen daher zu einem Großteil aus beschreibenden Attributen. Dazu zählen eine für ein Projekt eindeutige Bezeichnung, eine Beschreibung sowie zwei Attribute für die Bemessung der Komplexität und Priorität einer Aufgabe. Weiterhin wird der Bearbeitungs-Zeitraum, der Autor und der für die Bearbeitung zuständige Benutzer festgehalten. Alle diese Attribute enthalten Werte, die nicht spezifisch für eine einzelne Aufgabe sind.

Höchstens die Bezeichnung der Aufgabe liefert in Kombination mit dem Projekt, zu dem die Aufgabe gehört, eine Möglichkeit zur Identifizierung einer Aufgabe. Doch wie schon bei den Kanban Boards, können auch bei den Aufgaben alle beschreibenden Attribute im Lebenszyklus einer Aufgabe verändert werden. Ähnlich verhält es sich mit einer Kombination aus dem Kanban Board und dem Projekt, dem eine Aufgabe zugewiesen ist, sowie der Bezeichnung einer Aufgabe.

Auch über die Sammlung der einer Aufgabe zugewiesenen Tags, kann diese Aufgabe nicht identifiziert werden. Abschließend bleibt auch hier wieder ein Surrogat-Schlüssel in Form

einer UUID. Bei dem Akteur „Aufgabe“ können die gleichen Schlüsse wie schon beim Akteur „Kanban Board“ gezogen werden. Aufgaben werden daher als Entität definiert.

6. Benutzer

Benutzer sind die Repräsentationen von Menschen. Menschen können nicht durch Attribute wie den Vor- und den Nachnamen allgemein identifiziert werden. Die Benutzerkonten dieser Menschen können allerdings oft durch Attribute wie eine E-Mail-Adresse oder einen Benutzernamen identifiziert werden. In der Applikation Taskitory wird allerdings ein von Keycloak bereitgestellter Surrogatschlüssel verwendet. Da die gesamte DB-Tabelle von Keycloak bereitgestellt und verwaltet wird, kann dies nicht verändert werden.

Möglich wäre eine Identifizierung auch über die erstellten Aufgaben und Tags. Da sich diese Sammlung von anderen Akteuren verändern kann, ist das allerdings schlechter als die von Keycloak verwendete Methode.

Mit Ausnahme der ID (Identifikationsnummer) eines Benutzers können alle von Keycloak vorgegebenen Attribute eines Benutzers geändert werden. Das wird allerdings durch die Funktionen von Keycloak und nicht durch eigene Operationen auf den DB-Tabellen des Keycloak umgesetzt. Auch die Verbindungen zu anderen Akteuren (z. B. erstellte Aufgaben) verändern sich während des Lebenszyklus eines Benutzers. Daher kann der Akteur „Benutzer“ als Entität definiert werden.

7. Tag

Tags bestehen lediglich aus einer Bezeichnung, einer Beschreibung und einer Referenz auf den Erzeuger. In diesem Fall muss allerdings kein Surrogatschlüssel verwendet werden, da die Bezeichnung eines Tags global eindeutig ist. Die Bezeichnung ist damit ein natürlicher Schlüssel.

Die Bezeichnung und die Referenz auf den Erzeuger eines Tags sind unveränderlich. Allerdings kann die Beschreibung eines Tags während des Lebenszyklus eines Tags verändert werden.

Da die Identität eines Tags eindeutig durch einen Schlüssel repräsentiert wird und Tags veränderbare Attribute beinhalten, wird der Akteur „Tag“ als Entität definiert. Wenn ein Attribut keine Beschreibung hätte, wäre der Akteur „Tag“ ein Value Object.

3.3.2 Value Objects

Value Objects Modellieren die Elemente der Problemdomäne, die keine inhärente Identität besitzen. Die Identität von Value Objects wird durch die Werte aller Attribute des Value Objects repräsentiert. Dadurch können zwei Java-Objekte dasselbe Value Object repräsentieren. Value Objects sind unveränderlich. Value Objects kapseln in sich. In Entitäten werden Value Objects dazu verwendet, um geschlossene Subaufgabenbereiche zu kapseln und dadurch die Komplexität der Entitäten zu reduzieren.

Auf der Domänenschicht der Applikation Taskitory werden Value Objects ausschließlich in Entitäten eingebettet verwendet. Nachfolgend werden die Value Objects beschrieben.

1. Projekt-Schlüssel

Der Projekt-Schlüssel besteht aus einem einzelnen unveränderlichen Attribut, das die global eindeutige Zeichenfolge – den geheimen Projekt-Schlüssel – enthält. Das einzige Attribut des Value Objects ist unveränderlich und stellt selbst den Projekt-Schlüssel da.

Das Value Object „Projekt-Schlüssel“ ist als Attribut in der Projekt Entität eingebettet. Dadurch wird die Erzeugung und Bereitstellung des Projekt-Schlüssels als Hash von dem Rest der Projekt Entität abgeschottet.

2. Nachrichten-Inhalt

Auch der Nachrichten-Inhalt besteht nur aus einem einzelnen Attribut. In diesem Attribut wird der Inhalt einer Nachricht als String gespeichert. Dieses Attribut wurde nicht als „final“ definiert, da JPA einen Konstruktor ohne Parameter für Bestandteile des OR-Mappings fordert.

Da es keinen sinnvollen Standard-Wert für den Nachrichten-Inhalt gibt, muss die Unveränderlichkeit dieses Attributs auf anderem Wege erreicht werden. Um zu verhindern, dass der Konstruktor ohne Attribute von der Domänen-Logik verwendet werden kann, wird die Sichtbarkeit dieses Konstruktors eingeschränkt. Zusätzlich gibt es keine Methoden, mit denen das Attribut nach seiner Initialisierung auf einen anderen Wert gesetzt werden kann. Dadurch ist das Attribut effektiv unveränderlich.

Das Value Object „Nachrichten-Inhalt“ ist als Attribut in der Nachrichten Entität eingebettet. Dadurch wird das Umwandeln des Nachrichten-Inhalts von String zu JSON und umgekehrt von dem Rest der Nachriten Entität abgeschottet.

3. Sprint

Ein Sprint ist ein Zeitraum und wird in der Applikation Taskitory jeweils durch einen Zeitstempel für den Start und für das Ende des Sprints definiert. Beide Attribute sind unveränderlich und realisieren zusammen einen Sprint. Ein Attribut allein reicht dafür nicht aus. Beide Attribute formen zusammen die Identität des Sprints.

Der Standardwert für den Startzeitpunkt ist der Zeitpunkt der Erstellung des Objekts und der Endzeitpunkt exakt 2 Wochen darauf. Diese zwei Wochen sind ein gängiger Wert für einen Sprint-Zeitraum und können daher als Standardwert verwendet werden.

Das Value Object „Sprint“ ist als Attribut in der Kanban Board Entität eingebettet und schottet dadurch die Validierung der Zeitstempel und die Prüfung, ob der Sprint vorbei ist von der Entität ab.

4. (Aufgaben-)Komplexität

Die Komplexität einer Aufgabe wird durch ein einzelnes Attribut realisiert. Dieses Attribut hält in Form einer Zahl im Bereich [0, 20] den Grad der Komplexität fest. Dabei sprechen höhere Werte für eine größere Komplexität. Das Attribut ist unveränderlich und benötigt daher eine Standard-Initialisierung. Als Standard-Wert wird in diesem Fall der „leere“ Wert 0 verwendet.

Das Value Object „Komplexität“ ist als Attribut in der Aufgaben Entität eingebettet und schottet dadurch die Validierung der Komplexitätswerte von der Entität ab.

5. (Aufgaben-)Priorität

Die Priorität einer Aufgabe wird durch ein einzelnes Attribut realisiert. Dieses Attribut hält in Form einer Zahl im Bereich [0, 20] die Priorität fest. Ein geringerer Wert bedeutet eine höhere Priorität. Auch dieses Attribut ist unveränderlich und benötigt eine Standard-Initialisierung. Als Standard-Wert wird in diesem Fall der „leere“ Wert 0 verwendet.

Das Value Object „Priorität“ ist als Attribut in der Aufgaben-Entität eingebettet und schottet dadurch die Validierung der Prioritätswerte von der Entität ab.

6. Aufgaben-Lebenszyklus

Der fachliche Lebenszyklus einer Aufgabe in der Problemdomäne der Applikation Taskitory wird jeweils durch einen Zeitstempel für den Start und für das Ende des Bearbeitungszeitraums sowie einen Status definiert. Dieser Status wird verwendet, um Aufgaben auf einem Kanban Board einzuordnen. Der Startzeitpunkt eines Aufgaben-Lebenszyklus ist unveränderlich. Dieser Wert wird mit dem Zeitpunkt der Erzeugung des Objekts initialisiert.

Die beiden Attribute für den Status und den Endzeitpunkt sind allerdings veränderlich, um den Bearbeitungs-Prozess einer Aufgabe abzubilden. Besonders der Endzeitpunkt kann nicht bei der Erzeugung des Value Objects initialisiert werden, da der Endzeitpunkt noch ungewiss ist. Allerdings ist das nicht der einzige Fall, in dem ein Attribut verändert werden muss. Daher kann der Aufgaben-Lebenszyklus nicht als reines Value Object definiert werden. Allerdings handelt es sich auch nicht um eine Entität. Denn die Identität eines Aufgaben-Lebenszyklus wird durch alle drei Parameter-Werte definiert und nicht durch eine ID. Hinzu kommt die starke Bindung ausschließlich an die Aufgaben Entität.

Der Value Object Aufgaben-Lebenszyklus ist das Pendant zu der Tag Entität. Denn in beiden Fällen passt eine Definition als Entität oder Value Object nur mittelmäßig. Das Value Object „Aufgaben-Lebens-Zyklus“ ist als Attribut in der Aufgaben Entität eingebettet und schottet dadurch die Zustandsänderung einer Aufgabe von den anderen Funktionen des Aufgaben Entität ab.

7. Irrelevante Benutzerdaten

Wie bereits erwähnt, wird die DB-Tabelle der Benutzer Entität durch die Applikation Keycloak verwaltet. Auf die Benutzer Entität soll daher nur lesend zugegriffen werden können. Zudem sollen nur der Benutzername und die ID des Benutzers gelesen werden können. Die anderen von Keycloak verwendeten Attribute sind für diese Applikation nicht relevant. Der Zugriff auf diese Attribute wird daher vollständig verwehrt. Um dies effizient und übersichtlich umzusetzen wird dieses Value Object definiert.

Alle Attribute des Value Objects „Irrelevante Benutzerdaten” können nach der Initialisierung nicht geändert werden. Theoretisch kann eine Identität über ein Schlüssel-Attribut

wie z. B. die E-Mail-Adresse festgestellt werden. Praktisch lässt sich allerdings schwierig sagen, ob diese Annahme stimmt, da diese Tabelle von Keycloak verwaltet wird. Es bleibt zu beachten, dass dieses Value Object nie ohne eine Benutzer-Instanz verwendet wird und nur der Effizienz und Übersichtlichkeit halber verwendet wird. Man könnte von einer Art Schein Value Object sprechen.

8. Bezeichnung

Dieses Value Object wird übergreifend definiert, da fast alle Entitäten eine Bezeichnung mit den gleichen Eigenschaften und Restriktionen haben. Mit diesem Value Object wird sichergestellt, dass keine leeren oder zu langen Bezeichnungen vergeben werden. Das Value Object besteht lediglich aus einem Attribut für die Bezeichnung selbst. Es wird keine ID benötigt, da das Attribut das Objekt selbst repräsentiert.

9. Beschreibung

Dieses Value Object wird übergreifend definiert, da fast alle Entitäten eine Beschreibung mit den gleichen Eigenschaften und Restriktionen haben. Mit diesem Value Object wird sichergestellt, dass keine leeren oder zu langen Beschreibungen vergeben werden. Das Value Object besteht lediglich aus einem Attribut für die Beschreibung selbst. Es wird keine ID benötigt, da das Attribut das Objekt selbst repräsentiert.

3.3.3 Aggregate

„Aggregate“ fassen Entitäten und Value Objects zu transaktionalen Einheiten zusammen. Aggregate werden über einen „Aggregate Root“ verwaltet, der die Einhaltung der Domänen-Regeln des Aggregats sicherstellt. In der Applikation Taskitory werden das „Projekt Aggregat“ und das „Benutzer Aggregat“ verwendet. Nachfolgend werden beide Aggregate näher beschrieben.

1. Projekt Aggregat

Das erste Aggregat verwendet die Projekt Entität als Aggregate Root. Zudem beinhaltet das Projekt Aggregat die Kanban Board, Aufgaben, Nachrichten und Projekt-Mitgliedschafts Entitäten. Die Value Objects „Projekt-Schlüssel“, „Komplexität“, „Priorität“, „Auf-

gaben Lebenszyklus“, „Nachrichten-Inhalt“ und „Sprint" sind ebenfalls Teil dieses Aggregats, da sich die Bezugs-Entitäten in diesem Aggregat befinden.

Die Kanban Board Entität wurde dem Projekt Aggregat zugeordnet, da Kanban Boards nur von der Projekt Entität abhängig sind und nur über ein Projekt erreichbar sein sollen. Das heißt, dass die Funktionen von Kanban Boards nur über ein Projekt aufgerufen werden dürfen. Dabei klärt die Projekt Entität alle Rahmenbedingungen (z. B. gehört das Kanban Board zu diesem Projekt) und gibt den Funktionsaufruf dann an das Kanban Board weiter.

Die Aufgaben Entität wurde dem Projekt Aggregat zugeordnet, da Aufgaben hauptsächlich von der Projekt und der Kanban Board Entität abhängig sind. Allerdings besteht über das Autor-Attribut auch eine zwingende Abhängigkeit von der User Entität. Trotzdem werden die Aufgaben dem Projekt Aggregat zugeordnet. Denn Aufgaben sollen nur über ein Projekt erreichbar sein. User können keine Aufgaben ohne Projekt erstellen. Projekte können für eine Benutzer-Referenz die in dem gleichen Aggregat enthaltenen Projekt-Mitgliedschaften befragen. Die Funktionen von Aufgaben dürfen indirekt über ein Kanban Board und direkt über ein Projekt aufgerufen werden. In jedem Fall klärt die Projekt Entität alle Rahmenbedingungen für die Verwendung einer Aufgabe.

Die Nachrichten sowie die Projekt-Mitgliedschafts Entität stellen Beide Verbindungen zwischen der Benutzer Entität und der Projekt Entität her. Die Nachrichten und die Projekt-Mitgliedschafts Entität könnten daher dem Projekt- oder dem Benutzer Aggregat zugeordnet werden. Für die Applikation Taskitory wurde in beiden Fällen entschieden, die Entitäten dem Projekt Aggregat zu zuordnen, da die Funktionen der beiden Entitäten immer von einem Projekt angestoßen werden. Die initiale Interaktion geht immer von dem Projekt aus. Daher muss der Zugriff auf die Funktionen der Entitäten über die Projekt Entität ablaufen.

2. Benutzer Aggregat

Das Benutzer Aggregat beinhaltet die Benutzer und die Tag Entität inklusive anhängiger Value Objects (Irrelevante Benutzerdaten). Die Benutzer Entität ist nicht Teil des Projekt Aggregats, da die Benutzer-Instanzen unabhängig von einem Projekt bestehen können.

Die Tag Entität kann dies ebenfalls. Zusätzlich ist die Tag Entität ausschließlich von der Benutzer Entität abhängig. Dadurch wird die Benutzer Entität zum Aggregate Root, die die Zugriffe auf die Tag Entität regelt.

3.3.4 Repositories

Repositories dienen als Schnittstelle der Domänenschicht zum Persistenz-Framework auf der Plugin-Schicht. Die Repositiories geben vor, mit welchen Funktionen auf die persistenten DB-Einträge zugegriffen werden darf. Es sollte im Allgemeinen für jedes Aggregat ein Repository verwendet werden, sodass Aggregate immer als Einheit geladen und persistiert werden. Die Applikation Taskitory verwendet entsprechend je ein Repository für das Projekt und das Benutzer Aggregat.

3.3.5 Domänen Services

Domänen Services werden verwendet, um Entitäten übergreifende Logik oder Funktionalitäten zu implementieren. Die Applikation Taskitory benötigt auf der Domänen-Schicht einen „RessourceAccessService“ mit dem überprüft wird, ob ein Benutzer auf bestimmte Ressourcen eines Projekts zugreifen kann. Diese Funktionen können keinem der beiden Aggregate allein zugeordnet werden, da sie in beiden Aggregaten eingesetzt werden und sich immer auf Entitäten aus beiden Aggregaten beziehen.

4 Use Cases

In diesem Kapitel werden zentrale Use Cases der für die Applikation Taskitory thematisch sortiert aufgeführt. Die hier geschilderten Informationen sind besonders für die Applikations-Schicht der Clean Architecture relevant.

4.1 Projekte

UC 1 Projekt anlegen

Ein Projekt wird von einem Benutzer mit einer Bezeichnung und einer Beschreibung angelegt. Der geheime Gruppen-Schlüssel wird automatisch generiert. Optional kann der Be-

nutzer direkt ein Projekt-Team aus Benutzern zusammenstellen. Dabei kann er entscheiden, welche Rolle ein Benutzer-Konto erhält. Der Ersteller eines Projekts wird automatisch als Administrator dem Projekt zugeordnet. Ein Projekt beinhaltet nach der Initialisierung keine Aufgaben und keine Kanban Boards.

UC 2 Projekt-Attribute anpassen

Die Beschreibung und der geheime Projekt-Schlüssel eines Projekts sollen von einem Projekt-Administrator geändert werden können. Benutzer mit der Rolle „Mitglied“ können diese Attribute nicht ändern. Der geheime Schlüssel eines Projekts kann nur auf einen zufälligen Wert geändert werden.

UC 3 Projekt löschen

Das Löschen eines Projekts bewirkt, dass alle damit verbundenen Kanban Boards, Aufgaben, Nachrichten und Projekt-Mitgliedschaften gelöscht werden. Vor dem Löschen all dieser Verbindungen werden die Benutzer über das Löschen des Projekts per Nachricht informiert.

4.2 Projekt-Mitgliedschaften

UC 1 Benutzer-Konto zu einem Projekt hinzufügen

Das Projekt-Team kann auch nach der Initialisierung erweitert werden. Um einem Projekt beitreten zu können, muss ein Benutzer den geheimen Schlüssel eines Projekts angeben. Diesen geheimen Schlüssel können Projekt-Administratoren per Nachricht an den jeweiligen Benutzer schicken. Damit der Schlüssel nicht so einfach kompromittiert werden kann, wird der Schlüssel gehasht übertragen. Wenn ein Benutzer einem Projekt beigetreten ist, werden alle Projekt-Mitglieder und auch der neu hinzugefügt Benutzer darüber in einer Nachricht informiert.

UC 2 Ein Benutzer-Konto befördern

Projekt-Administratoren sollen einfache Mitglieder zu Projekt-Administratoren befördern können. Der beförderte Benutzer und alle Mitglieder eines Projekt-Teams sollen über die Beförderung informiert werden.

UC 3 Benutzer-Konto aus Projekt-Team entfernen

Ein Benutzer kann aus eigenem Antrieb aus einem Projekt austreten. Ein Benutzer mit der Rolle „Mitglied“ kann auch gegen seinen Willen von einem Projekt-Administrator aus einem Projekt entfernt werden. Dadurch werden die Referenzen auf den Benutzer aus dem Projekt entfernt. Der entfernte Benutzer soll per Nachricht informiert werden, wenn ein Administrator ihn aus einem Projekt entfernt hat.

4.3 Kanban Boards

UC 1 Kanban Board anlegen

Kanban Boards können mit einer für ein Projekt eindeutigen Bezeichnung, einer Beschreibung und einem festgelegten Sprint-Zeitraum erzeugt werden. Optional können noch die zwei Spalten „Review“ und „Testing“ aktiviert werden.

Ein Kanban Board ist bei Erzeugung leer und besitzt die oben benannten Spalten. Es sollen mehrere Kanban Boards in einem Projekt gleichzeitig bestehen können.

UC 2 Attribute eines Kanban Boards bearbeiten

Die Bezeichnung und die Beschreibung eines Kanban Boards können durch die Benutzer des entsprechenden Projekts angepasst werden. Dabei bleibt zu beachten, dass die Bezeichnung in einem Projekt eindeutig sein muss.

Außerdem ist es möglich die Spalten „Review“ und „Testing“ auszublenden. Alle in diesen Spalten befindlichen Aufgaben werden daraufhin ausgeblendet. Allerdings behalten sie den Status und können anschließend wieder eingeblendet werden.

UC 3 Kanban Board löschen

Kanban Boards können durch Benutzer des entsprechenden Projekts gelöscht werden. Die darauf befindlichen Aufgaben werden daraufhin in das Backlog gelegt. Die Aufgaben behalten ihren Bearbeitungs-Status.

UC 4 Aufgabe auf einem Kanban Board verschieben

Die zentrale Funktion eines Kanban Boards ist, Aufgaben über ihren Bearbeitungs-Status zu verwalten. Benutzer können die Aufgaben auf einem Kanban Board in einen anderen

Bearbeitungs-Status verschieben. Der Bearbeitungs-Status einer Aufgabe im Backlog eines Projekts kann nicht geändert werden.

UC 5 Kanban Board überführen

Das Kanban Board eines vergangenen Sprints kann in ein neues Kanban Board überführt werden. Das ist dann interessant, wenn sich auf dem alten Kanban Board noch nicht abgeschlossene Aufgaben befinden. Diese Aufgaben sollen in den gleichen Status des neuen Kanban Boards übernommen werden.

UC 6 Statistiken erstellen

Es soll möglich sein, Statistiken über ein bestehendes Kanban Board anzulegen. Mögliche Anwendungsfälle sind z. B. die Verteilung von Aufgaben nach Bearbeitungs-Status oder die Berechnung einer durchschnittlichen Bearbeitungsdauer.

4.4 Nachrichten

UC 1 Benutzer zu Projekt einladen

Projekt-Administratoren können Benutzer, die noch kein Teil des Projekt-Teams sind zu dem Projekt einladen. Dazu muss der geheime Schlüssel eines Projekts von einem Projekt-Administrator an diesen Benutzer in gehashter Form geschickt werden. Der Benutzer muss diesen gehashten Schlüssel vorweisen, um einem Projekt beizutreten.

UC 2 Nachrichten empfangen

Benutzer können die erhaltenen Nachrichten empfangen und lesen. Dabei ist es möglich Nachrichten einzeln und in größeren Gruppen zu empfangen. Wurde eine Nachricht einmal empfangen, wird sie unwiederbringlich gelöscht.

4.5 Aufgaben

UC 1 Aufgaben anlegen

Aufgaben sind immer einem Projekt zugeordnet. Aufgaben können daher nur angelegt werden, wenn ein Benutzer mindestens einem Projekt angehört. Im Standardfall wird eine neue Aufgabe zu dem Backlog eines Projekts hinzugefügt. Wenn bereits ein Kanban Board erstellt wurde, kann eine Aufgabe auch direkt dem Kanban Board hinzugefügt wer-

den. Dort wird die Aufgabe in der Spalte „Todo“ abgelegt. In diesem Fall wird die Aufgabe nicht mehr Im Backlog angezeigt.

UC 2 Benutzern Aufgaben zuweisen

Eine bestehende Aufgabe kann einem Mitglied des Projekt-Teams zur Bearbeitung zugewiesen werden. Dabei ist es egal, ob die Aufgabe vorher bereits einer Person zugewiesen war oder nicht. Es ist auch möglich die Zuweisung aufzuheben und keine neue Zuweisung vorzunehmen („NULL“). Der zugewiesene Benutzer wird über eine Nachricht informiert.

UC 3 Aufgaben zu einem Kanban Board hinzufügen

Aufgaben können aus dem Backlog zu genau einem Kanban Board hinzugefügt werden. Dazu muss eines aus vielen möglichen Kanban Boards eines Projekts gewählt werden. Außerdem muss festgelegt werden, in welcher Spalte die Aufgabe abgelegt werden soll. Der Standardfall ist, dass die Aufgabe in der Spalte „Todo“ abgelegt wird.

UC 4 Aufgaben bearbeiten

Die Attribute „Bezeichnung“, „Beschreibung“, „Komplexität“ und „zuständige Person“ (siehe oben) können nach der Erstellung einer Aufgabe jederzeit durch einen Benutzer verändert werden. Es bleibt zu beachten, dass die Bezeichnung der Aufgaben in einem Projekt eindeutig ist.

UC 5 Aufgaben löschen

Aufgaben können von allen Benutzern gelöscht werden. Dabei spielt es keine Rolle, ob die Aufgabe auf einem Kanban Board oder dem Backlog liegt.

4.6 Benutzer

UC 1 Benutzer-Konto registrieren

Um ein Benutzer-Konto zu registrieren, müssen ein Benutzername und ein Passwort bestimmt werden. Bei erfolgreicher Registrierung soll der Keycloak dem Benutzer direkt einen Access-Token zurückgeben. Bei einem Fehler wird ein passender Fehlercode zurückgegeben. Das Erstellen eines Eintrags in der Datenbank für das neue Benutzerkonto wird von dem Keycloak erledigt und kann über die REST-API des Keycloaks getriggert

werden. Für das Registrieren eines Benutzerkontos muss das Backend eine Anfrage lediglich an den Keycloak weiterleiten.

UC 2 Benutzer-Konto löschen

Wenn ein Benutzer sich entscheidet, sein Konto zu löschen, werden alle damit verbundenen Daten unwiederbringlich gelöscht. Das Ergebnis des Löschvorgangs wird durch einen passenden HTTP-Status quittiert. Das Löschen des Benutzerkonto-Datensatzes in der Datenbank wird von dem Keycloak erledigt und kann über die REST-API des Keycloaks getriggert werden. Für das Löschen eines Benutzerkontos muss das Backend eine Anfrage lediglich an den Keycloak weiterleiten.

4.7 Tags

UC 1 Tag erstellen

Tags können mit einer Bezeichnung und einer Beschreibung erzeugt werden und sind einem Benutzer zugeordnet. Die Bezeichnung ist global eindeutig.

UC 2 Tag bearbeiten

Die Beschreibung eines Tags kann im Nachhinein vom Erzeuger des Tags geändert werden.

UC 3 Tag löschen

Ein Tag kann von seinem Erzeuger gelöscht werden. Dadurch werden automatisch alle Referenzen von Aufgaben entfernt.

UC 4 Tags zuweisen

Aufgaben können über Tags mit Metadaten wie z. B. Versionsnummern uvm. versehen werden. Dabei ist der Zugriff nicht auf die eigens erzeugten Tags beschränkt.

5 Entwurfsmuster

Das Verwenden von Entwurfsmustern ist eine Programmiertechnik, um häufig auftretende Probleme zu lösen. Entwurfsmuster sind nicht auf eine Problemdomäne beschränkt,

sondern beziehen sich auf allgemeine Problemstellungen. Nachfolgend wird beispielhaft ein in der Applikation Taskitory verwendetes Entwurfsmuster beschrieben.

Das Erbauer Muster gehört zur Untergruppe der Erzeugungsmuster. Erzeugungsmuster werden verwendet, um die Instanziierung von Objekten zu regeln. Dafür wird die Logik extrahiert, die sich normalerweise in einem Konstruktor befindet. Mit Erzeugungsmustern wird festgelegt, welche Attribut-Belegungen erlaubt sind.

Mit einem Erbauer können die Attributwerte von Objekten schrittweise konfiguriert werden ohne, dass Objekte in einen unerlaubten Zustand initialisiert werden können. Dafür werden die Attribute einer Klasse in obligatorische und optionale Attribute unterteilt. Die optionalen Attribute werden mit Standardwerten belegt, sodass diese nicht explizit initialisiert werden müssen. Allerdings können diese Standardwerte explizit überschrieben werden. Jedes optionale Attribut kann mit einer eigenen Methode überschrieben werden und ermöglicht den direkt anschließenden Aufruf äquivalenter Methoden.

In der Applikation Taskitory wird das Erbauer Muster für das Erzeugen von Objekten der Klassen „Task“ (Aufgabe) und „KanbanBoard“ auf der Domänenschicht verwendet. Das Erbauer Muster bietet sich hier an, da Objekte dieser beiden Klassen eine Reihe von optionalen Parametern besitzen. Die Programmiersprache Java unterstützt optionale Methoden-Parameter nicht in der Form, wie es z. B. Python tut. Daher müsste in diesen beiden Fällen eine große Menge von Konstruktoren definiert werden, die den Code unübersichtlich und schlecht wartbar machen würde (siehe Abbildung 2).

Das Erbauer Muster wird für Aufgaben durch die Datei „CreateTask.java“ umgesetzt. Die Enthaltene Klasse „CreateTask“ definiert die gleichen Attribute wie eine Aufgabe mit Ausnahme der ID. Um den Arbeitsprozess des Erbauers anzustoßen, muss die statische Methode „CreateTask.named(String name)“ aufgerufen werden. Dadurch wird die Klasse CreateTask über den privaten Konstruktor mit Standard-Attributwerten und dem obligatorischen Namen instanziiert. Um den Erbauungs-Prozess fortzusetzen, muss die Methode „forProjectWithCreator(Project project, User creator)“ auf dem erzeugten Objekt aufgerufen werden, mit der die obligatorischen Attribute „Projekt“ und „Autor“ festgelegt werden. Der Rückgabetyp dieser Methode ist das Interface „Builder<KanbanBoard>“.

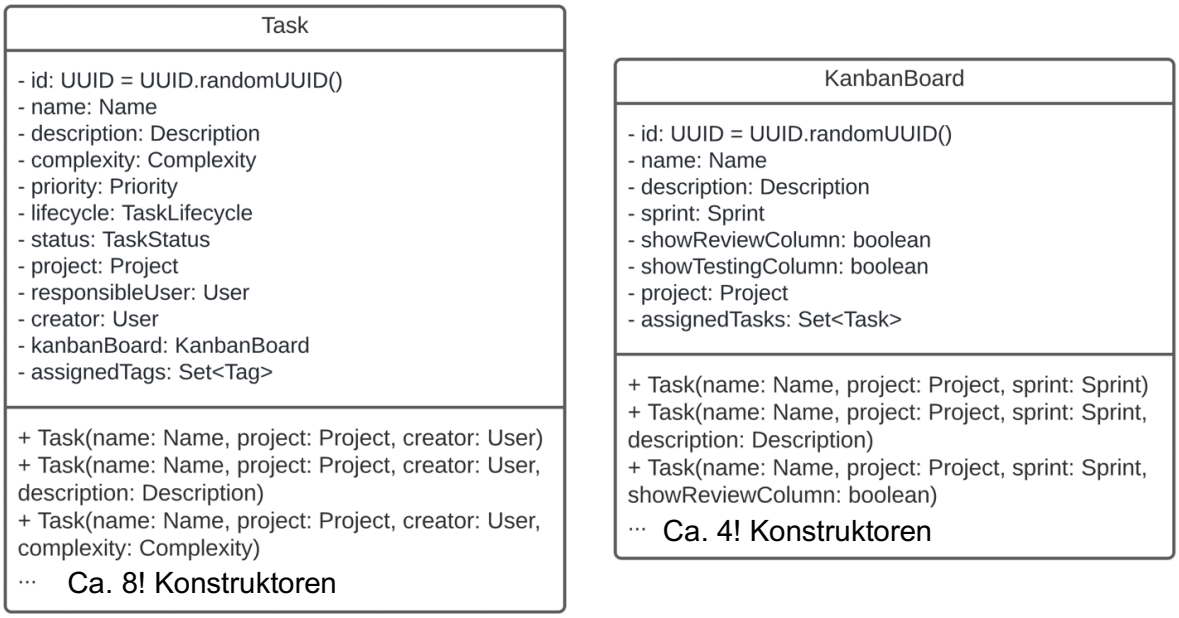


Abb. 2: Klassen „Task“ und „KanbanBoard“ ohne Erbauer Muster

Die Klasse „CreatableKanbanBoard“ implementiert dieses Interface. Die Klasse ist unter der Klasse „CreateTask“ definiert und wird verwendet, um die optionalen Parameter einer Aufgabe bei Bedarf festzulegen. Dazu wird für jeden optionalen Parameter eine Methode definiert. Diese Methoden haben alle Rückgabewerte vom Typ „CreatableTask“, sodass die Methoden kontinuierlich aufgerufen werden können. Um den Erbauungs-Prozess abzuschließen, muss die Methode „build()“ aufgerufen werden, die mit dem Konstruktor der Klasse „Task“ eine Objekt erzeugt. Um zu verhindern, dass Aufgaben auf anderem Weg erzeugt werden, wird die Sichtbarkeit der Konstruktoren der Klasse „Task“ eingeschränkt. Durch die Verwendung des Erbauer Musters verändert sich die Klassen-Struktur aus Abbildung 2 zu der Klassenstruktur in Abbildung 3.

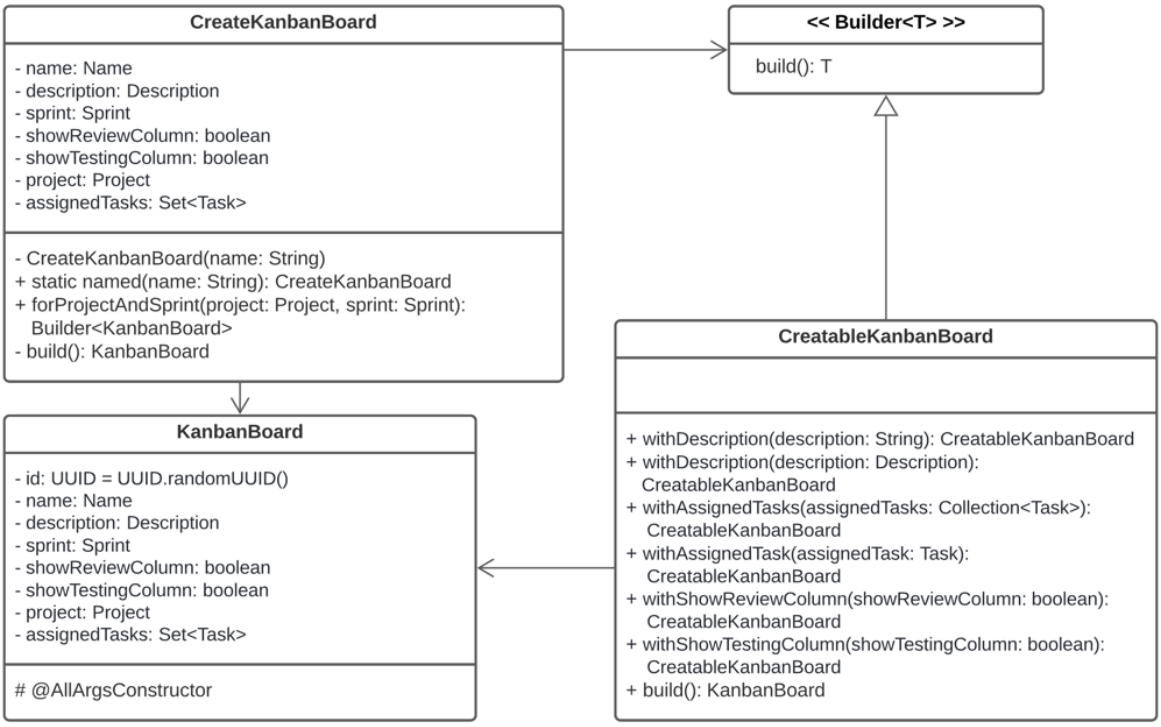


Abb. 3: Die Kanbanboard Entität mit integriertem Erbauer Muster

6 Programmierprinzipien

Programmierprinzipien sind Richtlinien für das Schreiben von verständlichem und einfach verwendbarem Quellcode. Es gibt eine Reihe von Programmierprinzipien, die in verschiedenen Kombinationen und mit verschiedenen Definitionen auftreten. Zwei weit verbreitete Listen von Programmierprinzipien sind die SOLID- und die General responsibility Assignment Patterns. Nachfolgend wird auf die in diesen Listen enthaltenen Programmierprinzipien und das DRY-Prinzip eingegangen. Es wird außerdem beschrieben, wie und wo diese Programmierprinzipien in der Applikation Taskitory verwendet wurden.

6.1 SOLID

6.1.1 Single Responsibility Principle

Das Single Responsibility Principle besagt, dass Quellcode-Einheiten nur eine Aufgabe erfüllen sollen. Das Prinzip ist auf Module, Klassen, Methoden und Variablen anwendbar.

In der Applikation Taskitory wurde das Single Responsibility Principle regelmäßig auf der Domänenschicht verwendet. Die Value Objects sind dazu gedacht inhaltlich zusammengehörige Funktionalität zu kapseln und dadurch aus den Entitäten zu extrahieren. Ein Beispiel dafür ist das Value Object „ProjectSecurityKey“. In dieser Klasse werden alle Funktionen für die Verwaltung von den Projekt Sicherheitsschlüsseln implementiert. Es werden außerdem die Zugriffsmöglichkeiten auf Projekt Sicherheitsschlüsseln definiert.

6.1.2 Open Closed Principle

Das Open Closed Principle besagt, dass Klassen und Module offen für Erweiterungen, aber verschlossen für Modifikationen ist. Klassen sollen nicht durch Veränderung der Klasse selbst, sondern durch Vererbung um typspezifische Funktionen erweitert werden.

Das Open Closed Principle wird durch die Nachrichten Entität in der Applikation Taskitory umgesetzt. Die allgemeine Struktur von Nachrichten wird durch eine abstrakte Vaterklasse definiert. Die Konkreten Implementierungen dienen als Spezialisierungdieser allgemeinen Struktur. Eine Erweiterung der Anzahl verschiedener Nachrichten ist jederzeit durch Vererbung möglich. Die grundlegende Struktur von Nachrichten ist allerdings nicht veränderbar.

6.1.3 Liskov Substitution Principle

Das Liskov Substitution Principle besagt, dass Sohnklassen an jeder Stelle im Quellcode anstelle einer Vaterklasse verwendet werden kann ohne, dass ein Fehler auftritt.

Es kann mehrere verschiedene Arten von Nachrichten in der Applikation Taskitory geben. Alle diese Nachrichten werden durch Sohnklassen einer übergeordneten abstrakten Vaterklasse umgesetzt. Um die verschiedenen Nachrichten persistieren zu können, wird der

Nachrichten-Typ zusätzlich durch ein Attribut repräsentiert. Beim aktuellen Stand sind keine speziellen Funktionen der Sohnklassen implementiert, sodass ein Austausch der Vater- mit der Sohnklasse keinen Unterschied machen würde.

6.1.4 Interface Segregation Principle

Das Interface Segregation Principle besagt, dass Interfaces aufgabenspezifisch sein sollten. Es sollen möglichst keine zu allgemeinen Interfaces verwendet werden.

Die Repositories auf der Domänen Schicht sind nur für ein bestimmtes Aggregat zuständig. Dadurch ist der Aufgaben Bereich klar fokussiert und abgegrenzt. Durch eine Komposition der Repositories (siehe Services auf der Applikations Schicht) können alle Funktionen verwendet werden. Allerdings können auch gegebenenfalls durch Weglassen eines Repositories die Abhängigkeiten reduziert werden.

6.1.5 Dependency Inversion Principle

Das Dependency Inversion Principle besagt, dass Abstraktionen nicht von technischen Details abhängen dürfen. Umgekehrt sollen Details von Abstraktionen abhängen.

Die Applikation Taskitory hat eine Clean Architecture. Dadurch sind technische Implementierungen von der allgemeinen Domänenlogik abhängig. Da fällt besonders bei Betrachtung dem OR-Mapping auf. Dabei werden auf der Domänenschicht die allgemeinen Schnittstellen (Repositories) definiert, über die Objekte einer Entität angefragt werden können. Auf der Pluginschicht befindet sich die technische Implementierung dieser Schnittstellen (SpringDataRepositories), die das Interface „JpaRepository“ erweitern. Auf der allgemeineren Schicht werden somit die Anforderungen an eine technische Implementierung definiert. Es besteht keine Abhängigkeit von diesen technischen Implementierungen. Umgekehrt hängen die technischen Implementierungen von der Schnittstellendefinition ab, da sie die definierten Anforderungen erfüllen müssen.

6.2 GRASP

6.2.1 Geringe Kopplung

Unter „Kopplung“ wird der Grad der Abhängigkeit zwischen Objekten verstanden. Die Abhängigkeiten zwischen Quellcode mit verschiedenen Aufgaben sollten minimiert und so lose wie möglich gehalten werden.

Die Architektur der Applikation Taskitory sind so konzipiert, dass die inneren Schichten von keiner weiter außen liegenden Schicht abhängen. Eine besonders Lose Kopplung wird durch die Repositories auf der Domänen Schicht umgesetzt. Denn diese Interfaces definieren nur eine Schnittstelle, von der die konkreten Implementierungen auf der Plugin Schicht abhängen.

6.2.2 Hohe Kohäsion

Unter „Kohäsion“ wird die räumliche Nähe von inhaltlich zusammengehörigem Quellcode bezeichnet. Es sollte das Ziel sein, inhaltlich zusammengehörigen Quellcode möglichst nahe zusammenzuhalten.

Eine Hohe Kohäsion besteht auf der Domänenschicht besonders durch die verwendeten Value Objects. Wie bereits beim Single Responsibility Principle beschrieben wurde, kapseln diese Value Objects ganz bestimmtes inhaltlich zusammengehöriges Verhalten. Auch die Entitäten grenzen Quellcode inhaltlich ab. Eine Ausnahme dazu sind die Aggregate Roots, die als einziger zentraler Zugriffspunkt auf die Entitäten eines Aggregats auch Methoden für den Zugriff auf die Funktionen dieser enthaltenen Entitäten ermöglichen.

6.2.3 Information Expert

Als „Information Expert“ wird eine Klasse bezeichnet, die allen Quellcode für eine bestimmte Aufgabe enthält. Der gesamte diese Aufgabe betreffende Quellcode soll in dieser einen Klasse gekapselt werden. Interne Informationen dürfen die Klasse nicht verlassen.

Wie bereits beim Single Responsibility Principle beschrieben wurde, kapseln diese Value Objects ganz bestimmtes inhaltlich zusammengehöriges Verhalten. Zum Beispiel wird in

der Klasse „Sprint“ die Validierung von Sprintzeiträumen implementiert. Zusätzlich kann geprüft werden, ob ein Sprintzeitraum bereits abgelaufen ist. Alle einen Sprintzeitraum betreffenden, für die Applikation Taskitory relevanten Funktionen sind und werden zukünftig ausschließlich in dieser Klasse abgelegt. Das Geheimnisprinzip lässt sich besonders gut an dem Value Object „ProjectSecurityKey“ erkennen. Denn der eigentliche Sicherheitsschlüssel kann nicht im Klartext aus dem Objekt extrahiert werden. Lediglich ein kryptographischer Hash-Wert dieses Sicherheitsschlüssels kann herausgegeben werden.

6.2.4 Polymorphie

Unter „Polymorphie“ wird das unterschiedliche Implementieren von Verhalten durch Überschreiben von Methoden in einer Vererbungsbeziehung verstanden.

Die Nachrichten Entität wurde durch ein Refactoring (siehe Kapitel 7.1) so angepasst, dass konkrete Nachrichten von einer allgemeinen, abstrakten Klasse abgeleitet werden. Dadurch wird Polymorphie theoretisch implementierbar. Allerdings wird bei dem aktuellen Stand keine Funktion für Nachrichten implementiert, die Polymorphie bedarf. Denkbar wäre eine Verschlüsselung des Inhalts bei bestimmten Nachrichten-Typen.

6.2.5 Pure Fabrication

Das Pure Fabrication Prinzip besagt, dass es Klassen in der Domäne geben darf, die Funktionen zur Verfügung stellen, für die sie kein Information Expert sind.

Der einzige definierte Domain Service der Applikation Taskitory ist eine Klasse mit einer Reihe von statischen Methoden, die das Verhältnis zwischen einem Benutzer und einer Projekt-Ressource überprüfen. Diese Klasse ist kein Information Expert für dieses Verhältnis. Das wird besonders durch die Projekt Mitgliedschafts Entität deutlich, die das Verhältnis eines Benutzers zu einem Projekt und damit auch seinen Ressourcen definiert. In dieser Entität werden auch Funktionalitäten definiert, die das Verhältnis zwischen Benutzer und Projekt-Ressource beeinflusst.

6.2.6 Delegieren

Verhalten sollte sofern möglich an Information Experts delegiert werden.

Entitäten lagern inhaltlich zusammengehöriges Verhalten an Value Objects aus. Zum Beispiel wird die Validierung des Sprintzeitraums von der Kanban Board Entität an das Sprint Value Object delegiert.

6.2.7 Controller

Ein Controller ist die Schnittstelle zwischen dem Frontend und dem Backend. Controller leiten Systemereignisse an die richtigen Stellen weiter.

Die Applikations und die Adapter Schicht dienen als Schnittstellen für die Kommunikation der Anwendungslogik auf der Domänen Schicht und der Benutzerschnittstelle auf der Plugin Schicht. Die relevanten Informationen der auf der Plugin Schicht eingehenden HTTP Anfragen werden von der Adapterschicht umgeformt, damit sie von der Applikations Schicht verarbeitet werden können. Die Applikatins Schicht zerlegt die Anfrage in ihre Bestandteile und leitet diese an die zuständigen Stellen in der Domänen Logik weiter.

6.2.8 Creator Principle

Das Creator Principle definiert, wer Instanzen von einer Klasse erzeugen darf. Eine Klasse A darf Instanzen einer Klasse B erzeugen, wenn A Objekte von B enthält, diese verarbeitet oder von diesen abhängt. Es ist außerdem erlaubt, wenn A der Information Expert für das Erzeugen von B-Instanzen ist.

Das Creator Principle wird durch die Aggregate Roots auf der Domänenschicht und die Builder (siehe Kapitel 5) umgesetzt. Die Aggregate Roots sind die zentralen Verwalter der in einem Aggregat enthaltenen Entitäten und müssen daher Instanzen der enthaltenen Entitäten erzeugen können. Die Builder-Klassen sind Information Experts für das Erzeugen von Instanzen der jeweiligen Klassen (Kanban Board und Aufgaben Entitäten) und müssen daher ebenso Instanzen dieser Klassen erzeugen können.

6.3 DRY

DRY steht für Don’t Repeat Yourself. Dieses Prinzip besagt, dass keine Redundanzen im Quellcode vorkommen sollen.

Das DRY Prinzip wird in der Applikation wesentlich dadurch umgesetzt, dass das OR-Mapping auf der Domänen Schicht umgesetzt wird. Theoretisch müsste sich das OR-Mapping als technische Implementierung auf der Pluginschicht befinden. Allerdings müsste dafür der Domänen-Code zu einem Großteil dupliziert werden. Zusätzlich müsste ein Mapping zwischen den beiden Schichten aufgebaut werden. Diese Dopplung hätte zu enormem unnötigem Aufwand bei der Entwicklung der Applikation Taskitory geführt. Daher wurde das OR-Mapping auf der Domänen Schicht umgesetzt.

7 Code Smells und Refactoring

Code Smells sind Strukturen im Quellcode, die die Entwicklung, Wartung und/oder das Testen behindern. Um diese Strukturen zu beseitigen oder zu verbessern, wird ein Refactoring durchgeführt. Dabei werden dysfunktionale Strukturen identifiziert und durch funktionale Strukturen ersetzt. Nachfolgend werden zwei Code Smells in dem Quellcode der Applikation Taskitory identifiziert und durch ein Refactoring beseitigt.

7.1 Open Closed Principle Verstoß

Die Nachrichten Entität auf der Domänen Schicht unterscheidet mit dem Attribut „type“ zwischen verschiedenen Arten von Nachrichten. Um unterschiedliches Verhalten der gleichen Methode bei unterschiedlichen Nachrichten zu implementieren, müssten vor dem Refactoring If- oder Switch-Anweisungen über das besagte Attribut verwendet werden. Das entspricht nicht dem Open Closed Principle (siehe Kapitel 6) und kann daher als Code Smell angesehen werden. Erweiterungen oder Änderungen würden aufwändige Anpassungen in der Klasse erfordern. Zudem werden solche Unterscheidungen bei einer großen Anzahl von verschiedenen Nachrichts-Typen schnell unübersichtlich. Abbildung 4 stellt den Zustand vor dem Refactoring dar.

```java
public class Message {

    @Id
    private final UUID id = UUID.randomUUID();

    1 usage
    @Embedded
    @AttributeOverrides({ @AttributeOverride(name = "content", column = @Column(length = 500, nullable = false, updatable = false)) })
    @NonNull
    private MessageContent content;

    1 usage
    @Column(nullable = false, updatable = false)
    @Enumerated(EnumType.STRING)
    @NonNull
    private MessageType type;

    @Column(length = 6, nullable = false, updatable = false)
    private final Timestamp dispatch = Timestamp.valueOf(LocalDateTime.now());

    1 usage
    @Column(nullable = false)
    private boolean read = false;

    @ManyToOne(targetEntity = Project.class)
    @JoinColumn(name = "origin", referencedColumnName = "name", nullable = false, updatable = false)
    @NonNull
    private Project origin;

    1 usage
    @ManyToOne(targetEntity = User.class)
    @JoinColumn(name = "recipient", referencedColumnName = "id", nullable = false, updatable = false)
    @NonNull
    private User recipient;

    3 usages  Krayaty
    public void markAsRead() { this.read = true; }

    2 usages  Krayaty
    public JSONObject getContentAsJson() { return new JSONObject(this.content.getContent()); }

    5 usages  Krayaty
    public boolean hasType(MessageType type) { return this.type == type; }

    2 usages  Krayaty
    public boolean isFor(User user) { return this.recipient.equals(user); }
```

Abb. 4: Die Nachrichten Entität

Um das Open Closed Principle für die Nachrichten Entität auf der Domänen Schicht umzusetzen, werden extra Klassen für die verschiedenen Nachrichten erstellt und von der allgemeinen Nachrichten-Klasse abgeleitet. Um nur valide Nachrichten erstellen zu können, wird eine Factory verwendet und die Sichtbarkeit des Konstruktors der Nachrichten Vaterklasse eingeschränkt. Die Nachrichten Vaterklasse kann zusätzlich als abstrakt definiert werden. Die Abbildungen 5 und 6 zeigen den Zustand nach dem Refactoring.

```java
1 usage
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class Invitation extends Message {

    1 usage
    protected Invitation(@NonNull MessageContent content, @NonNull Project origin, @NonNull User recipient) {
        super(content, MessageType.INVITATION, origin, recipient);
    }
}
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class NewTeamMemberMessage extends Message{
    1 usage
    protected NewTeamMemberMessage(@NonNull MessageContent content, @NonNull Project origin, @NonNull User recipient) {
        super(content, MessageType.NEW_TEAM_MEMBER, origin, recipient);
    }
}
1 usage
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class KickedFromProjectMessage extends Message{
    1 usage
    protected KickedFromProjectMessage(@NonNull MessageContent content, @NonNull Project origin, @NonNull User recipient) {
        super(content, MessageType.KICKED_FROM_PROJECT, origin, recipient);
    }
}

@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class RoleChangeMessage extends Message {

    1 usage
    protected RoleChangeMessage(@NonNull MessageContent content, @NonNull Project origin, @NonNull User recipient) {
        super(content, MessageType.ROLE_CHANGE, origin, recipient);
    }
}
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class ProjectDeletedMessage extends Message{

    1 usage
    protected ProjectDeletedMessage(@NonNull MessageContent content, @NonNull Project origin, @NonNull User recipient) {
        super(content, MessageType.PROJECT_DELETION, origin, recipient);
    }
}
```

Abb. 5: Die konkreten Nachrichten Implementierungen

Beim jetzigen Stand der Applikation Taskitory sind zwar noch keine polymorphen Methoden enthalten. Allerdings ist eine Implementierung von polymorphem Verhalten durch dieses Refactoring deutlich einfacher und übersichtlicher machbar.

```java
6 usages  Krayaty *
public class MessageFactory {

    1 usage  Krayaty *
    public static Message createInvitationToProjectForUser(Project origin, User recipient, ProjectRole role) {
        JSONObject contentAsJSON = new JSONObject();
        contentAsJSON.put(MessageContentJSONKey.TITLE.getValue(), "Projekt-Einladung");
        contentAsJSON.put(MessageContentJSONKey.PROJECT_NAME.getValue(), origin.getName());
        contentAsJSON.put(MessageContentJSONKey.PROJECT_SEC_KEY.getValue(), origin.getKey().getValue());
        contentAsJSON.put(MessageContentJSONKey.ROLE.getValue(), role.name);

        MessageContent content = new MessageContent(contentAsJSON);
        return new Invitation(content, origin, recipient);
    }

    1 usage  Krayaty *
    public static Message createProjectMemberRoleChangeMessage(User recipient, ProjectMembership membership, ProjectRole newRole) {
        JSONObject contentAsJSON = new JSONObject();
        contentAsJSON.put(MessageContentJSONKey.TITLE.getValue(), "Rolle geändert");
        contentAsJSON.put(MessageContentJSONKey.PROJECT_NAME.getValue(), membership.getProject().getName());
        contentAsJSON.put(MessageContentJSONKey.USER_NAME.getValue(), membership.getUser().getUsername());
        contentAsJSON.put(MessageContentJSONKey.OLD_ROLE.getValue(), membership.getRole());
        contentAsJSON.put(MessageContentJSONKey.NEW_ROLE.getValue(), newRole);

        MessageContent content = new MessageContent(contentAsJSON);
        return new RoleChangeMessage(content, membership.getProject(), recipient);
    }

    Krayaty *
    public static Message createProjectDeletedMessage(Project origin, User recipient) {
        JSONObject contentAsJSON = new JSONObject();
        contentAsJSON.put(MessageContentJSONKey.TITLE.getValue(), "Projekt gelöscht");
        contentAsJSON.put(MessageContentJSONKey.PROJECT_NAME.getValue(), origin.getName());

        MessageContent content = new MessageContent(contentAsJSON);
        return new ProjectDeletedMessage(content, origin, recipient);
    }

    1 usage  Krayaty *
    public static Message createKickedFromProjectMessage(Project origin, User recipient) {
        JSONObject contentAsJSON = new JSONObject();
        contentAsJSON.put(MessageContentJSONKey.TITLE.getValue(), "Aus Projekt entfernt");
        contentAsJSON.put(MessageContentJSONKey.PROJECT_NAME.getValue(), origin.getName());

        MessageContent content = new MessageContent(contentAsJSON);
        return new KickedFromProjectMessage(content, origin, recipient);
    }

    2 usages  Krayaty *
    public static Message createNewProjectMemberMessage(Project origin, User recipient, ProjectMembership membership) {
        JSONObject contentAsJSON = new JSONObject();
        contentAsJSON.put(MessageContentJSONKey.TITLE.getValue(), "Neues Projekt-Mitglied");
        contentAsJSON.put(MessageContentJSONKey.PROJECT_NAME.getValue(), origin.getName());
        contentAsJSON.put(MessageContentJSONKey.USER_NAME.getValue(), membership.getUser().getUsername());
        contentAsJSON.put(MessageContentJSONKey.ROLE.getValue(), membership.getRole().name);

        MessageContent content = new MessageContent(contentAsJSON);
        return new NewTeamMemberMessage(content, origin, recipient);
    }
```

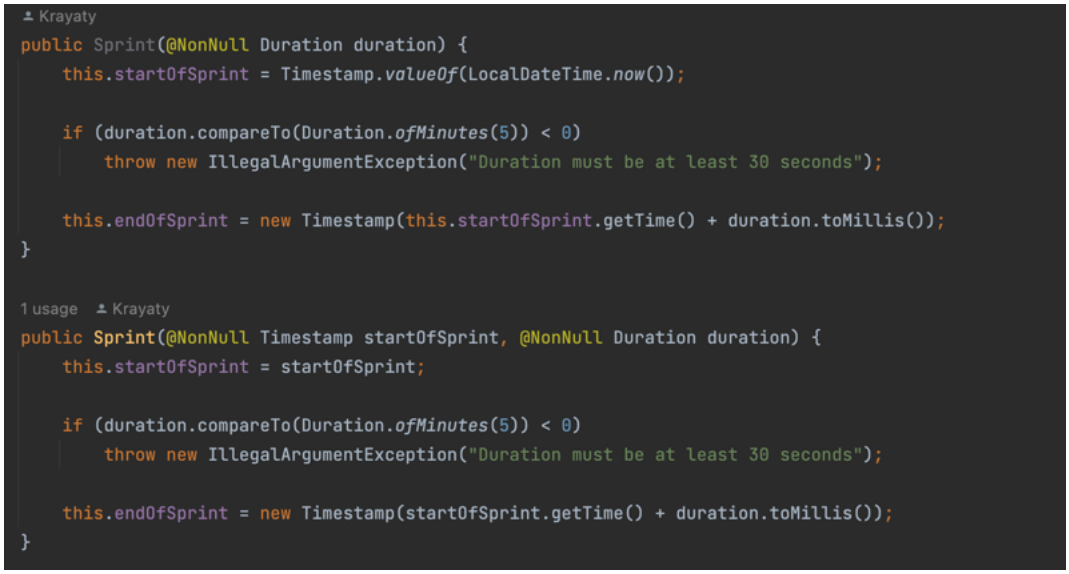
Abb. 6: Die Factory-Methoden für das erzeugen konkreter Nachrichten

7.2 DRY Prizip Verstoß

Das Value Object „Sprint“ auf der Domänen Schicht kapselt hauptsächlich die Erzeugung und Validierung von Sprintzeiträumen. Dabei kommen mehrere Konstruktoren mit unterschiedlichen Parametern zum Einsatz. Dem entsprechend müssen die gleichen Validierungs-Regeln in verschiedenen Konstruktoren angewendet werden. Vor einem Refactoring wurde die Validierungs-Logik in jedem Konstruktor redundant implementiert. Das verstößt klar gegen das DRY Prinzip (siehe Kapitel 6) und kann daher als Code Smell bewertet werden. Denn durch diese Dopplungen können Inkonsistenzen in der Validierungslogik auftreten und der Wartungsaufwand ist relativ hoch. Die Abbildungen 7 und 8 zeigen den Zustand vor einem Refactoring.

```
public Sprint(Timestamp endOfSprint) {
    this.startOfSprint = Timestamp.valueOf(LocalDateTime.now());

    if (endOfSprint == null) {
        long twoWeeks = Duration.ofDays(14).toMillis();
        this.endOfSprint = new Timestamp(startOfSprint.getTime() + twoWeeks);
        return;
    }

    if (endOfSprint.before(startOfSprint) || endOfSprint.equals(startOfSprint))
        throw new IllegalArgumentException("End of sprint must be after start of sprint");

    Duration duration = Duration.between(startOfSprint.toLocalDateTime(), endOfSprint.toLocalDateTime());
    if (duration.compareTo(Duration.ofMinutes(5)) < 0)
        throw new IllegalArgumentException("Duration must be at least 30 seconds");

    this.endOfSprint = endOfSprint;
}
 public Sprint(@NonNull Timestamp startOfSprint, Timestamp endOfSprint) {
     this.startOfSprint = startOfSprint;

     if (endOfSprint == null) {
         long twoWeeks = Duration.ofDays(14).toMillis();
         this.endOfSprint = new Timestamp(startOfSprint.getTime() + twoWeeks);

     } else {
         if (endOfSprint.before(startOfSprint) || endOfSprint.equals(startOfSprint))
             throw new IllegalArgumentException("End of sprint must be after start of sprint");

         Duration duration = Duration.between(startOfSprint.toLocalDateTime(), endOfSprint.toLocalDateTime());
         if (duration.compareTo(Duration.ofMinutes(5)) < 0)
             throw new IllegalArgumentException("Duration must be at least 30 seconds");

         this.endOfSprint = endOfSprint;
     }
 }
```

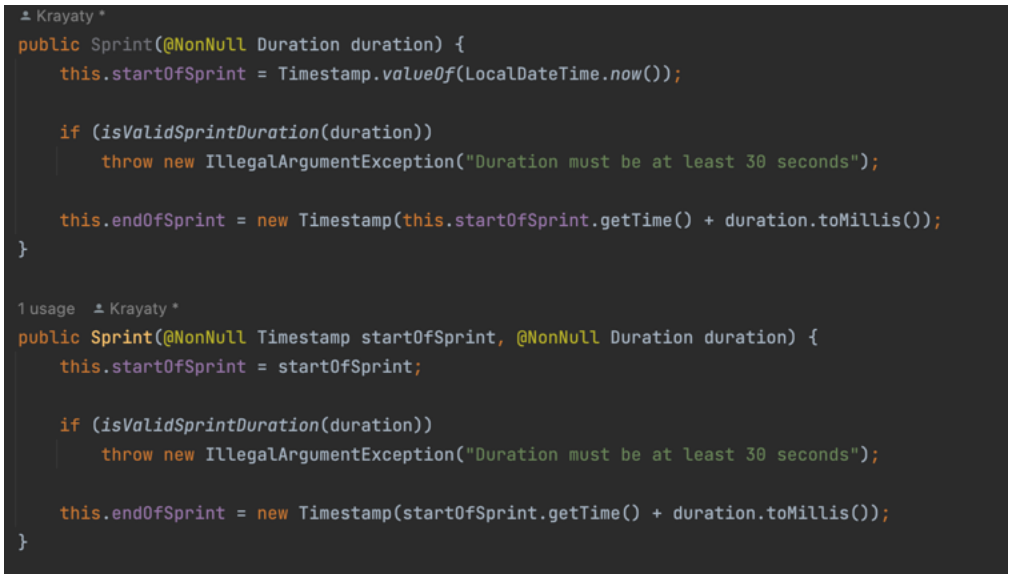
Abb. 7: Die Konstruktoren des Value Objects „Sprint“ (Teil 1)



```
Krayaty
public Sprint(@NonNull Duration duration) {
    this.startOfSprint = Timestamp.valueOf(LocalDateTime.now());

    if (duration.compareTo(Duration.ofMinutes(5)) < 0)
        throw new IllegalArgumentException("Duration must be at least 30 seconds");

    this.endOfSprint = new Timestamp(this.startOfSprint.getTime() + duration.toMillis());
}

1 usage  Krayaty
public Sprint(@NonNull Timestamp startOfSprint, @NonNull Duration duration) {
    this.startOfSprint = startOfSprint;

    if (duration.compareTo(Duration.ofMinutes(5)) < 0)
        throw new IllegalArgumentException("Duration must be at least 30 seconds");

    this.endOfSprint = new Timestamp(startOfSprint.getTime() + duration.toMillis());
}
```

Abb. 8: Die Konstruktoren des Value Objects „Sprint“ (Teil 2)

Durch Extrahieren der Validierungslogik für Sprint-Zeiträume und Sprint-Endzeitpunkte in eine eigene statische Methode, wird die Redundanz beseitigt. Bei einer Änderung der Validierungs-Logik muss nun nur noch die jeweilige statische Methode verändert werden und nicht mehr jeder betroffene Konstruktor. Die Abbildungen 9, 10 und 11 zeigen den Zustand nach dem Refactoring.



```
Krayaty *
public Sprint(@NonNull Duration duration) {
    this.startOfSprint = Timestamp.valueOf(LocalDateTime.now());

    if (isValidSprintDuration(duration))
        throw new IllegalArgumentException("Duration must be at least 30 seconds");

    this.endOfSprint = new Timestamp(this.startOfSprint.getTime() + duration.toMillis());
}

1 usage  Krayaty *
public Sprint(@NonNull Timestamp startOfSprint, @NonNull Duration duration) {
    this.startOfSprint = startOfSprint;

    if (isValidSprintDuration(duration))
        throw new IllegalArgumentException("Duration must be at least 30 seconds");

    this.endOfSprint = new Timestamp(startOfSprint.getTime() + duration.toMillis());
}
```

Abb. 9: Die refactorierten Sprint-Konstruktoren (Teil 1)

```java
public Sprint(Timestamp endOfSprint) {
    this.startOfSprint = Timestamp.valueOf(LocalDateTime.now());

    if (endOfSprint == null) {
        long twoWeeks = Duration.ofDays(14).toMillis();
        this.endOfSprint = new Timestamp(startOfSprint.getTime() + twoWeeks);
        return;
    }

    if (isValidEndOfSprint(startOfSprint, endOfSprint))
        throw new IllegalArgumentException("End of sprint must be after start of sprint");

    Duration duration = Duration.between(startOfSprint.toLocalDateTime(), endOfSprint.toLocalDateTime());
    if (isValidSprintDuration(duration))
        throw new IllegalArgumentException("Duration must be at least 30 seconds");

    this.endOfSprint = endOfSprint;
}

8 usages  Krayaty *
public Sprint(@NonNull Timestamp startOfSprint, Timestamp endOfSprint) {
    this.startOfSprint = startOfSprint;

    if (endOfSprint == null) {
        long twoWeeks = Duration.ofDays(14).toMillis();
        this.endOfSprint = new Timestamp(startOfSprint.getTime() + twoWeeks);

    } else {
        if (isValidEndOfSprint(startOfSprint, endOfSprint))
            throw new IllegalArgumentException("End of sprint must be after start of sprint");

        Duration duration = Duration.between(startOfSprint.toLocalDateTime(), endOfSprint.toLocalDateTime());
        if (isValidSprintDuration(duration))
            throw new IllegalArgumentException("Duration must be at least 30 seconds");

        this.endOfSprint = endOfSprint;
    }
}
```

Abb. 10: Die refactorierten Sprint-Konstruktoren (Teil 2)

```java
4 usages  new *
private static boolean isValidSprintDuration(@NonNull Duration duration) {
    return duration.compareTo(Duration.ofMinutes(5)) < 0;
}

2 usages  new *
private static boolean isValidEndOfSprint(@NonNull Timestamp startOfSprint, @NonNull Timestamp endOfSprint) {
    return endOfSprint.before(startOfSprint) || endOfSprint.equals(startOfSprint);
}
```

Abb. 11: Die extrahierte Validierungslogik für Sprints