Grobentwurf

Freyschmidt, Henry Lewis (HLF)
Hama, Zana Salih (ZSH)
Krasnovska, Paula (PK)
Krüger, Lucas (LK)
Prüger, Marvin Oliver (MOP)
Seep, Tom-Malte (TMS)
Zabel, Steven (SZ)

4. Februar 2024

	Verantwortung	Inhalt	Korrektur
Backlog	SZ	SZ	HLF, LK, PK, ZSH
User-Storys	SZ	SZ	HLF, LK, ZSH
Use Case Diagramme	SZ	SZ	HLF, LK
Tasks	SZ	SZ	HLF, LK, PK, ZSH
Klassendiagramm	SZ	SZ	PK, ZSH
Architektur	HLF, LK	HLF, LK, ZSH	LK, PK, SZ, ZSH
Komponentendiagramm	HLF, LK	HLF, LK, PK	LK, SZ, ZSH
Interface	LK	HLF, LK, PK, ZSH	PK, SZ, ZSH
Sequenzdiagramm	LK	LK	HLF, PK, ZSH
Datenstruktur	TMS	MOP, TMS	HLF, LK, PK, ZSH
Datenbankmodell	TMS	MOP, TMS	LK, PK, ZSH
Klassendiagramm	TMS	MOP	LK, SZ
Datenbankmodell	TMS	TMS	LK
Technologien	LK	LK, ZSH	PK, ZSH

Inhaltsverzeichnis

1 Backlog

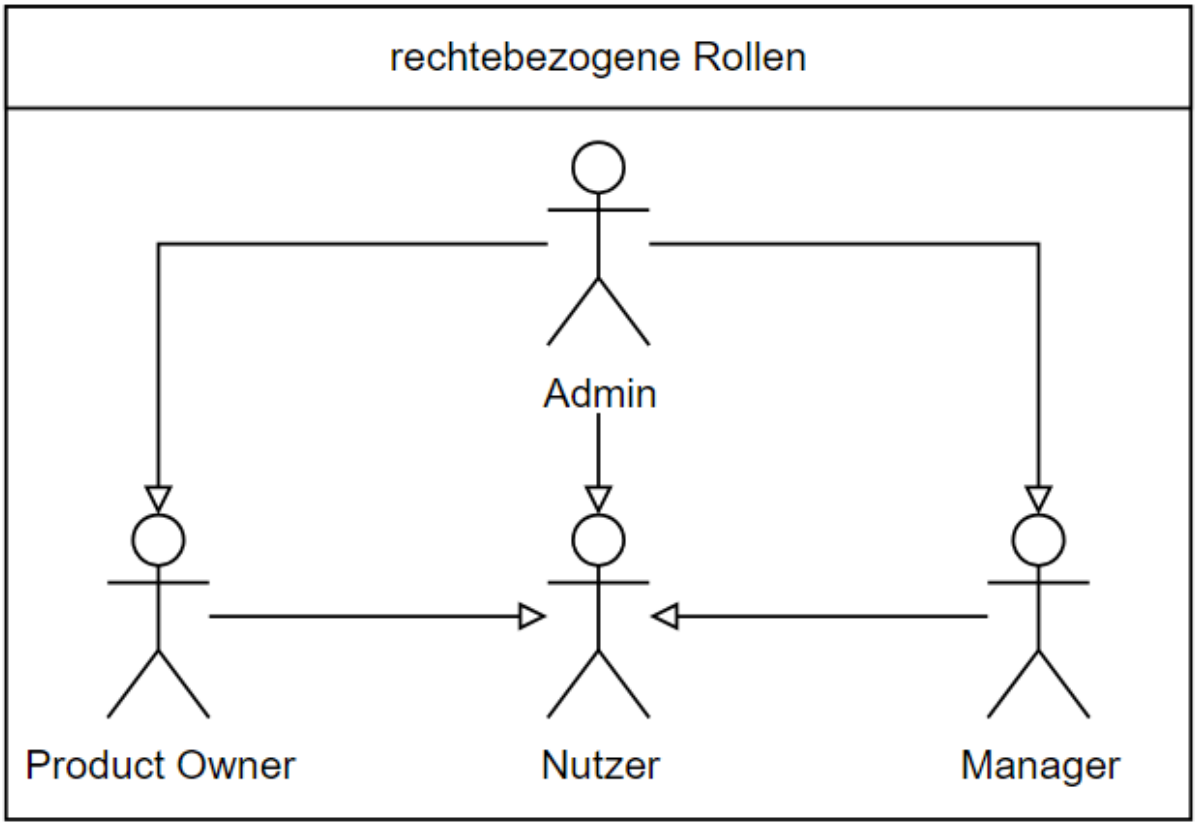
Anforderungsgruppen

Die Anforderungen sind in die Gruppen Rolle, Account, Profil, User-Story, Task, Planning Poker, Task-Liste, Task-Board, Kalender und System unterteilt. Diese Unterteilung hilft bei der Prüfung der Einzigartigkeit der User-Storys und bei der Implementierung. Jede Anforderungsgruppe kann als Konstrukt aufgefasst werden und wird zu Beginn kurz beschrieben. Alle User-Storys der Gruppe haben mit diesem Konstrukt zu tun und definieren es oder sein Verhalten. Bei den Konstrukten, welche durch Anforderungsgruppen beschrieben sind, handelt es sich nicht um Vorgaben zur Architektur.

1.1 [R] Rolle

Im Projekt gibt es drei Arten von Rollen: technische, rechtebezogene und visuelle Rollen. Die Domäne des Systems für rechtebezogene Rollen ist durch die Use-Case-Diagramme zu den Anforderungsgruppen ausgedrückt.

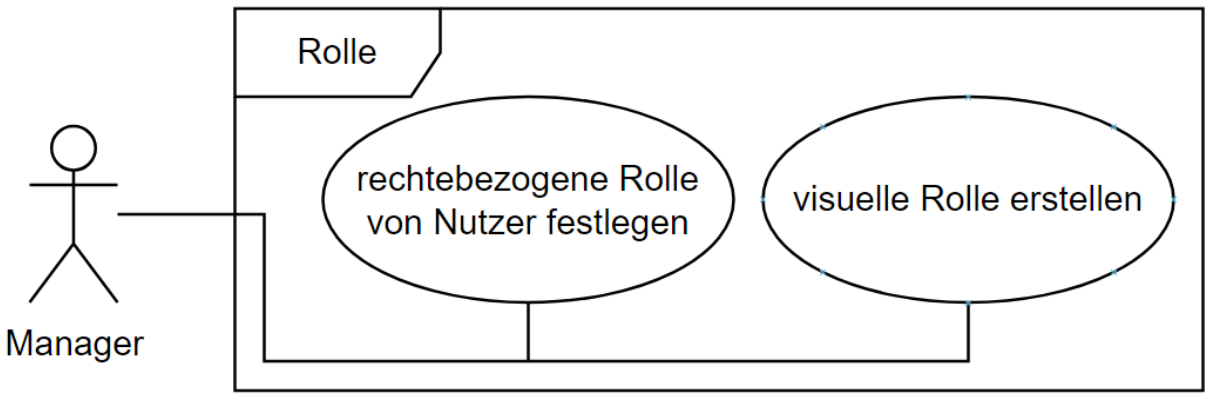
- Mit technischen Rollen werden die Funktionsweisen wichtiger Schnittstellen ausgedrückt. Technische Rollen werden nicht explizit implementiert.
  - Das System ist die Zusammenfassung aller Software-Komponenten, welche zum Projekt gehören.
  - Ein potenzieller Nutzer ist eine Person, welche sich dem System gegenüber noch nicht authentifiziert hat.
  - Ein Nutzer ist eine Person, welche sich dem System gegenüber authentifiziert hat.
  - Ein Entwickler ist eine Person, welche das System anpasst.
- Mit rechtebezogenen Rollen werden die Rechte zur Nutzung des Systems verwaltet.
  - Ein Admin kann im System alle Funktionalitäten nutzen.
  - Ein Standard-Nutzer kann im System alle Funktionalitäten nutzen, welche nicht explizit einer rechtebezogenen Rolle zugeordnet sind.
  - Ein Product Owner verfügt über die Rechte eines Standard-Nutzers und darf User-Storys verwalten.
  - Ein Manager verfügt über die Rechte eines Standard-Nutzers und darf rechtebezogenen Rollen aller Nutzer außer des Admins verwalten.



- Mit visuellen Rollen werden Spezialisierungen innerhalb einer rechtebezogenen Rolle angegeben. Visuelle Rollen haben keinen Einfluss auf die Rechte.
  - Ein Frontend-Entwickler ist eine Spezialisierung des Nutzers.
  - Ein Backend-Entwickler ist eine Spezialisierung des Nutzers.
  - Ein Datenbank-Entwickler ist eine Spezialisierung des Nutzers.

R1 Als Manager kann ich die rechtebezogene Rolle eines Nutzers festlegen, um die Rechte des Nutzers im Projekt zu definieren.

R2 Als Manager kann ich zu einer rechtebezogenen Rolle eine visuelle Rolle erstellen, damit Nutzer sich anhand dieser einordnen können.



1.2 [A] Account

Ein Account identifiziert einen Nutzer im System.

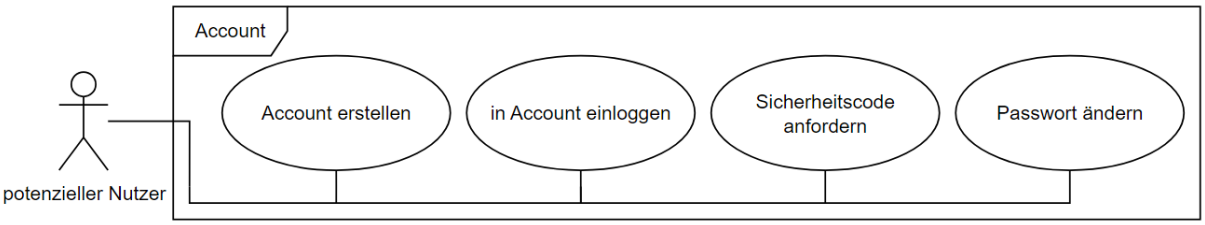
A1 Als Nutzer habe ich einen Account, um mich mit diesem gegenüber des Systems zu authentifizieren.

A2 Als potenzieller Nutzer kann ich unter Angabe einer Mail und eines Passworts einen Account erstellen, um mich zukünftig mit diesem gegenüber des Systems zu authentifizieren.

A3 Als potenzieller Nutzer kann ich mich mit meiner Mail und meinem Passwort in meinen Account einloggen, um mich gegenüber des Systems zu authentifizieren.

A4 Als potenzieller Nutzer kann ich mit meiner Mail einen Sicherheitscode anfordern, um diesen, falls ich mein Passwort vergessen habe, zur Authentifikation zu nutzen.

A5 Als potenzieller Nutzer kann ich mit meiner Mail und meinem aktuellsten Sicherheitscode mein Passwort zurücksetzen, um einen verlorenen oder unsicheren Zugang selbstständig wiederherzustellen oder sicherer zu machen.



1.3 [P] Profil

Ein Profil erlaubt einem Nutzer, Angaben über sich zu verfassen. Das Profil eines Nutzers ist für andere Nutzer im System einsehbar.

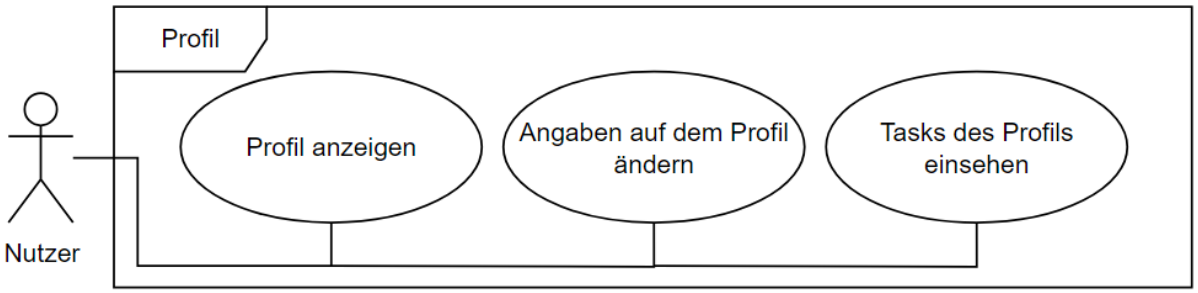
P1 Als Nutzer habe ich ein Profil, um mich mit diesem im System zu zeigen.

P2 Als Nutzer kann ich auf meinem Profil meinen Namen, eine persönliche Beschreibung, eine visuelle Rolle und eine projektbezogene Beschreibung angeben, um mich zu beschreiben.

P3 Als Nutzer kann ich meine Angaben auf meinem Profil ändern, um diese zu aktualisieren.

P4 Als Nutzer kann ich mir das Profil eines Nutzers anzeigen lassen, um die Angaben des Nutzers auf seinem Profil zu sehen.

P5 Als Nutzer kann ich auf dem Profil eines Nutzers eine Liste von Tasks einsehen, um den Beitrag zum Projekt des Nutzers anzusehen.



1.4 [U] User-Story

Eine User-Story beschreibt eine Anforderung an das Endprodukt.

U1 Als Nutzer kann ich mir alle User-Storys anzeigen lassen, um eine Übersicht über diese zu bekommen.

U2 Als Nutzer kann ich eine User-Story auswählen, um mir alle zu ihr gespeicherten Informationen anzeigen lassen.

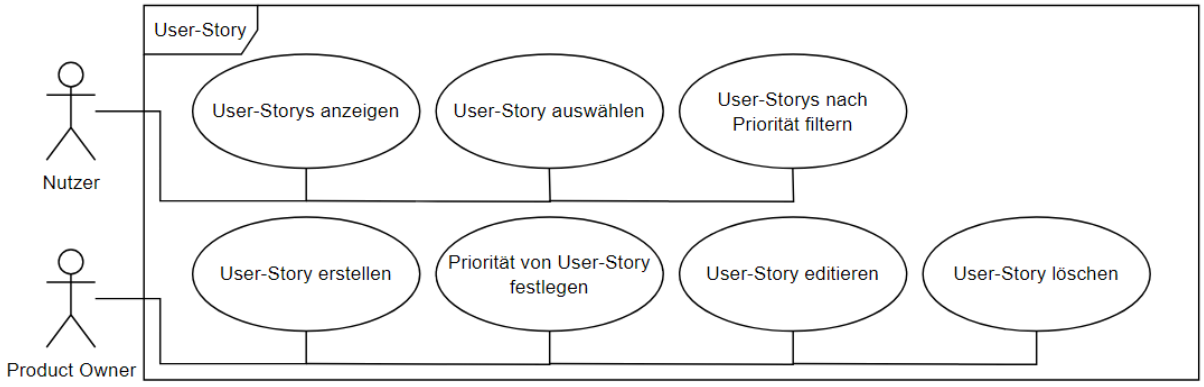
U3 Als Product Owner kann ich eine User-Story erstellen, um Anforderungen dem Projekt hinzuzufügen.

U4 Als Product Owner kann ich eine User-Story editieren, um sie an neue Umstände anzupassen.

U5 Als Product Owner kann ich einer User-Story eine Priorität (Urgent > High > Normal > Low) zuordnen, um die Dringlichkeit der User-Story auszudrücken.

U6 Als Product Owner kann ich eine User-Story löschen, um nicht mehr nötige Anforderungen zu entfernen.

U7 Als Nutzer kann ich User-Storys nach der Priorität (Urgent > High > Normal > Low) filtern, um nur User-Storys ausgewählter Prioritäten anzuzeigen.



1.5 [T] Task

Eine Task ist eine Aufgabe, die der Erfüllung einer User-Story dient.

T1 Als Nutzer kann ich mir alle Tasks anzeigen lassen, um eine Übersicht über diese zu bekommen.

T2 Als Nutzer kann ich eine Task auswählen, um mir alle zu ihr gespeicherten Informationen anzeigen lassen.

T3 Als Nutzer kann ich eine zu einer User-Story verlinkte Task erstellen, um diese User-Story zu verfeinern.

T4 Als Nutzer kann ich eine Task editieren, um sie an neue Umstände anzupassen.

T5 Als Nutzer kann ich eine Task löschen, um nicht mehr nötige Aufgabe zu entfernen.

T6 Als Nutzer kann ich eine Task als fertig markieren, um anzuzeigen, dass diese Task vollständig erfüllt ist.

T7 Als Nutzer kann ich einer Task eine Priorität (Urgent > High > Normal > Low) zuordnen, um die Dringlichkeit der Task auszudrücken.

T8 Als Nutzer kann ich einer Task Nutzer zuordnen, um auszudrücken, dass die Task von diesen Nutzern erledigt wird.

T9 Als Nutzer kann ich eine Task mit einer Abgabefrist versehen, um zu verdeutlichen, wann die Ergebnisse der Task benötigt werden.

T10 Als Nutzer kann ich eine Task mit einer Aufwandseinschätzung versehen, um auszudrücken, wie lange die Bearbeitung der Task schätzungsweise dauert.

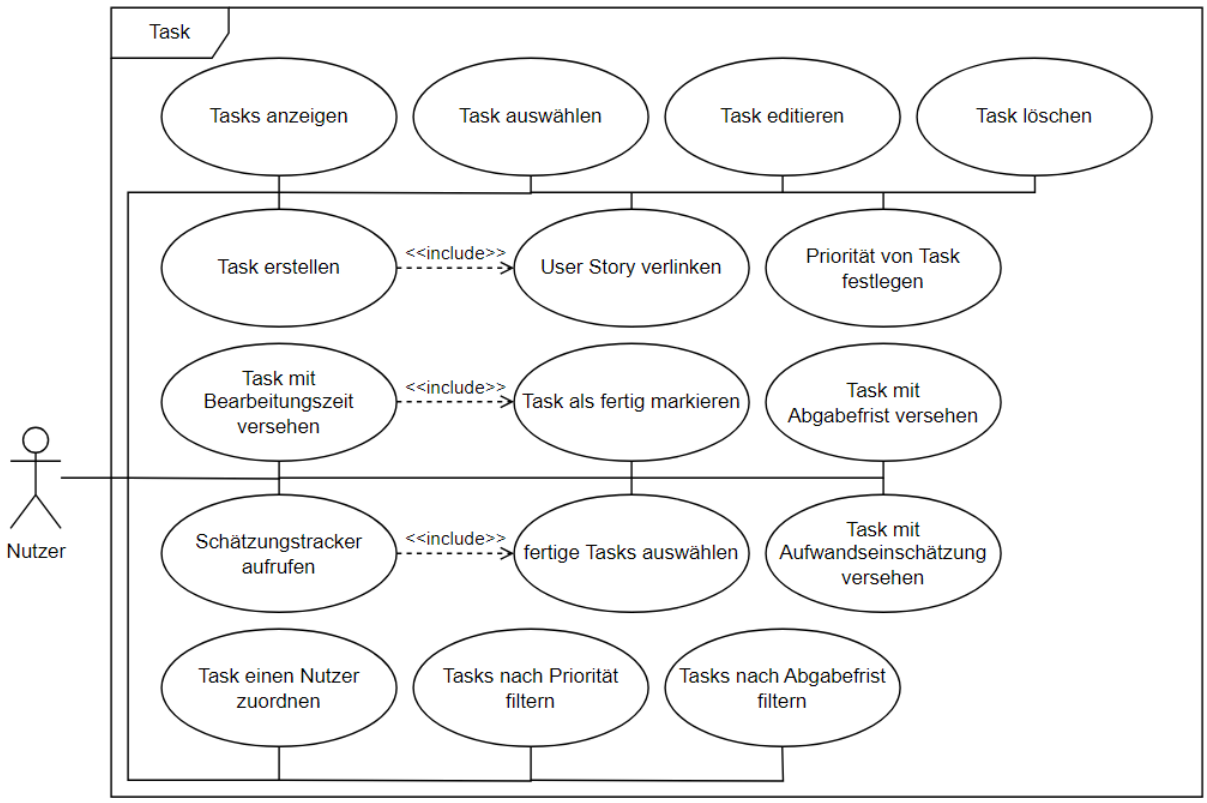
T11 Als Nutzer kann ich Tasks nach deren Fertigstellung mit einer Bearbeitungszeit versehen, um auszudrücken, wie lange die Bearbeitung der Task dauerte.

T12 Als Nutzer kann ich einen Schätzungstracker aufrufen, welcher für beliebige, fertige Tasks eine Visualisierung ausgibt, wie sich die Aufwandseinschätzungen von den Bearbeitungszeiten unterscheiden, um die Aufwandseinschätzungen bewerten zu können.

T13 Als Nutzer bekomme ich eine Benachrichtigung, sobald die Abgabefrist einer Task, der ich zugeordnet bin, bei der doppelten Zeit der Aufwandseinschätzung liegt, um mich darauf hinzuweisen.

T14 Als Nutzer kann ich Tasks nach der Priorität (Urgent > High > Normal > Low) filtern, um nur Tasks ausgewählter Prioritäten anzuzeigen.

T15 Als Nutzer kann ich Tasks nach deren Abgabefrist filtern, um nur Tasks, deren Abgabefristen in einem ausgewählten Intervall liegen, zu finden.



1.6 [PP] Planning Poker

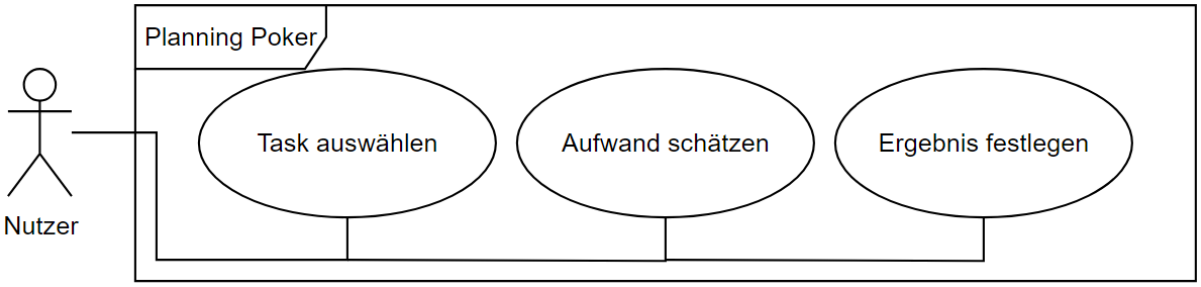
Beim Planning Poker wird der benötigte Aufwand einer Task von Nutzern beraten und geschätzt. Somit lässt sich die Aufwandseinschätzung interaktiv gestalten.

PP1 Als Nutzer kann ich auf Planning Poker zugreifen, um dort mit anderen Nutzern eine Aufwandseinschätzung für Tasks durchzuführen.

PP2 Als Nutzer kann ich beim Planning Poker eine Task auswählen, um diese zur Einschätzung vorzuschlagen.

PP3 Als Nutzer kann ich beim Planning Poker eine Einschätzung abgeben, um den Aufwand einer Task einzuschätzen.

PP4 Als Nutzer kann ich beim Planning Poker ein Ergebnis festlegen, um dieses als Aufwandseinschätzung der aktuellen Task festzulegen.



1.7 [TL] Task-Liste

Eine Task-Liste gruppiert Tasks bezüglich einer Eigenschaft.

TL1 Als Nutzer kann ich mir alle Task-Listen anzeigen lassen, um eine Übersicht über diese zu bekommen.

TL2 Als Nutzer kann ich eine Task-Liste ausklappen, um ihre Tasks einzublenden.

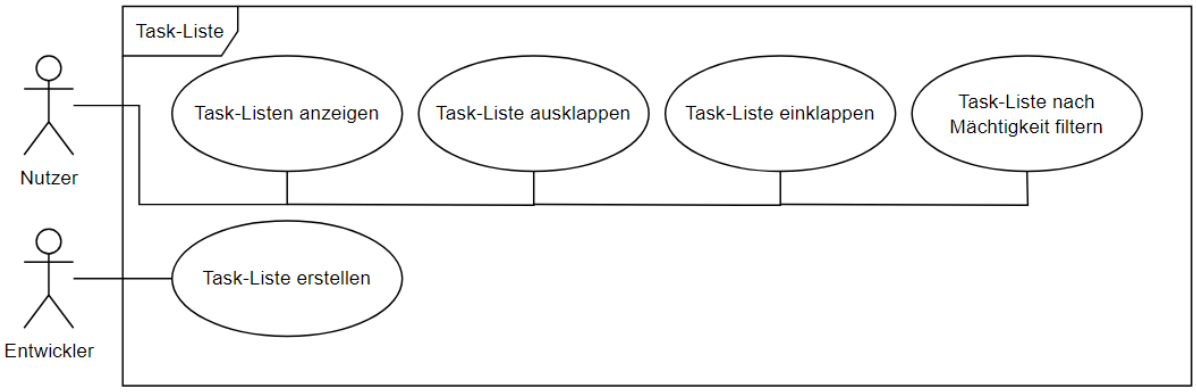
TL3 Als Nutzer kann ich eine Task-Liste einklappen, um ihre Tasks auszublenden.

TL4 Als Nutzer kann ich zum Projekt eine Task-Liste mit den Tasks des Projekts finden, um einen Überblick über die Tasks des Projekts zu bekommen.

TL5 Als Nutzer kann ich zum Sprint eine Task-Liste mit den Tasks des Sprints finden, um einen Überblick über die Tasks des Sprints zu bekommen.

TL6 Als Nutzer kann ich Task-Listen nach der Mächtigkeit filtern, um nur Task-Listen, deren Mächtigkeiten in einem ausgewählten Intervall liegen, zu finden.

TL7 Als Entwickler kann ich (leicht) eine neue Task-Liste erstellen, um das System an neue Anforderungen anzupassen.



1.8 [TB] Task-Board

Ein Task-Board bildet Task-Listen ab. Auf ihm können diese verwaltet werden.

TB1 Als Nutzer kann ich ein Task-Board mit der Task-Liste des Projekts und den Task-Listen jedes Sprints aufrufen, um das Projekt verwalten zu können.

TB2 Als Nutzer kann ich ein Task-Board mit der Task-Liste eines Sprints und den zum Sprint spezifischen Task-Listen 'Tasks in Bearbeitung', 'Tasks unter Review', 'Tasks unter Test' und 'fertiggestellte Tasks' aufrufen, um den Sprint verwalten zu können.

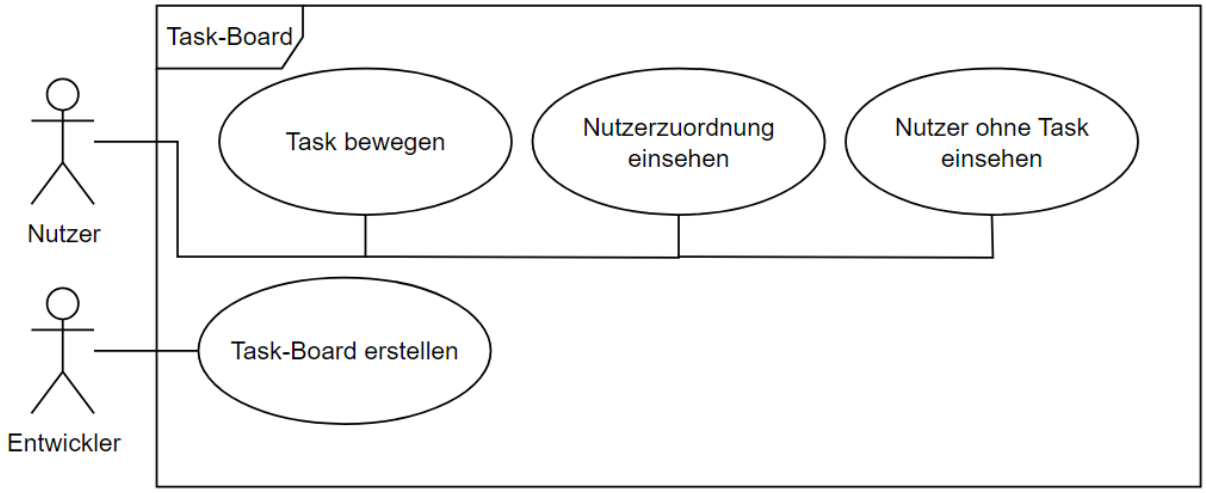
TB3 Als Nutzer kann ich auf einem Task-Board Tasks zwischen den angezeigten Task-Listen verschieben, um die Prozesse rund um die Bearbeitung der Tasks visuell anschaulich durchführen zu können.

TB4 Als Nutzer kann ich auf einem Task-Board gleichzeitig mit anderen Nutzern arbeiten, ohne dass Konflikte entstehen, um parallel arbeiten zu können.

TB5 Als Nutzer kann ich auf einem Task-Board einsehen, welcher Task welche Nutzer zugeordnet sind, um die Verteilung von Tasks zu erleichtern.

TB6 Als Nutzer kann ich auf dem Task-Board einsehen, welche Nutzer aktuell keiner Task zugeordnet sind, um die Verteilung von Tasks zu erleichtern.

TB7 Als Entwickler kann ich (leicht) ein neues Task-Board erstellen, um das System an neue Anforderungen anzupassen.



1.9 [K] Kalender

Ein Kalender bildet zeitlich-abhängige Ereignisse eines Nutzers ab. Auf ihm können Ereignisse organisiert und geplant werden.

K1 Als Nutzer kann ich auf einen Kalender zugreifen, um eine Übersicht über zeitlich-festgelegte Ereignisse zu bekommen.

K2 Als Nutzer wird jede Task mit Abgabefrist, der ich zugeordnet bin, anhand ihrer Abgabefrist automatisch meinem Kalender hinzugefügt, um einen Überblick und eine Erinnerung zu ermöglichen.

K3 Als Nutzer kann ich Meetings mit einem Namen, einem Zeitraum, einem Ort, einer Art und einer Auflistung der betroffenen Nutzer beziehungsweise Rollen erstellen, um Meetings zu planen.

K4 Als Nutzer wird jedes Meeting, dem ich zugeordnet bin, anhand seines jeweiligen Zeitraums automatisch meinem Kalender hinzugefügt, um einen Überblick und eine Erinnerung zu ermöglichen.

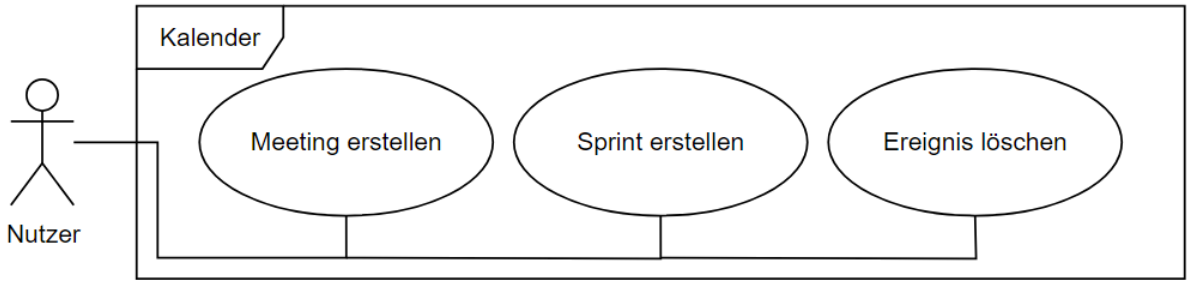
K5 Als Nutzer kann ich Sprints mit einem Namen, einem Zeitraum und einer Auflistung der betroffenen Nutzer beziehungsweise Rollen erstellen, um Sprints zu planen.

K6 Als Nutzer wird jeder Sprint, dem ich zugeordnet bin, anhand seines jeweiligen Zeitraums automatisch meinem Kalender hinzugefügt, um einen Überblick und eine Erinnerung zu ermöglichen.

K7 Als Nutzer kann ich Ereignisse aus meinem Kalender für mich löschen, um nicht mehr daran erinnert zu werden.

K8 Als Nutzer bekomme ich eine Benachrichtigung, sobald ein Ereignis meinem Kalender hinzugefügt wird, um kein neues Ereignis zu verpassen.

K9 Als Nutzer bekomme ich eine Benachrichtigung, sobald ein Ereignis auf meinem Kalender in zwei Stunden ansteht, um mich daran zu erinnern.



1.10 [S] System

Das System ist die Zusammenfassung aller Software-Komponenten, welche zum Projekt gehören.

S1 Als Nutzer kann ich innerhalb von 2 Sekunden neue Angaben aller anderen Nutzer im System sehen, um immer auf dem aktuellsten Stand zu sein.

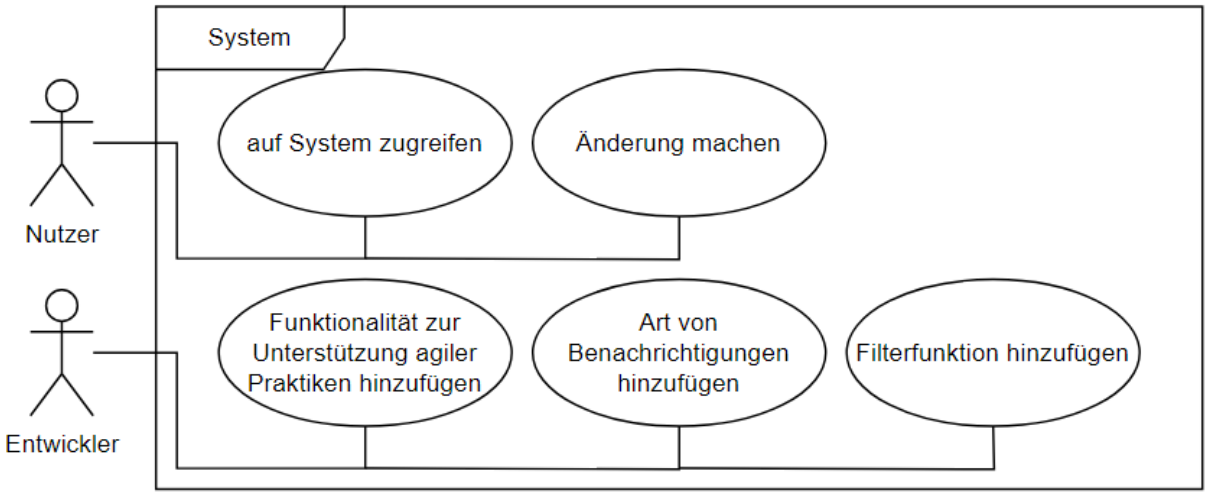
S2 Als Nutzer kann ich Zugriff auf das System bekommen, obwohl auch andere Nutzer schon Zugriff haben, um parallel arbeiten zu können.

S3 Als Nutzer kann ich jederzeit auf das System zugreifen, um dauerhaft arbeiten zu können.

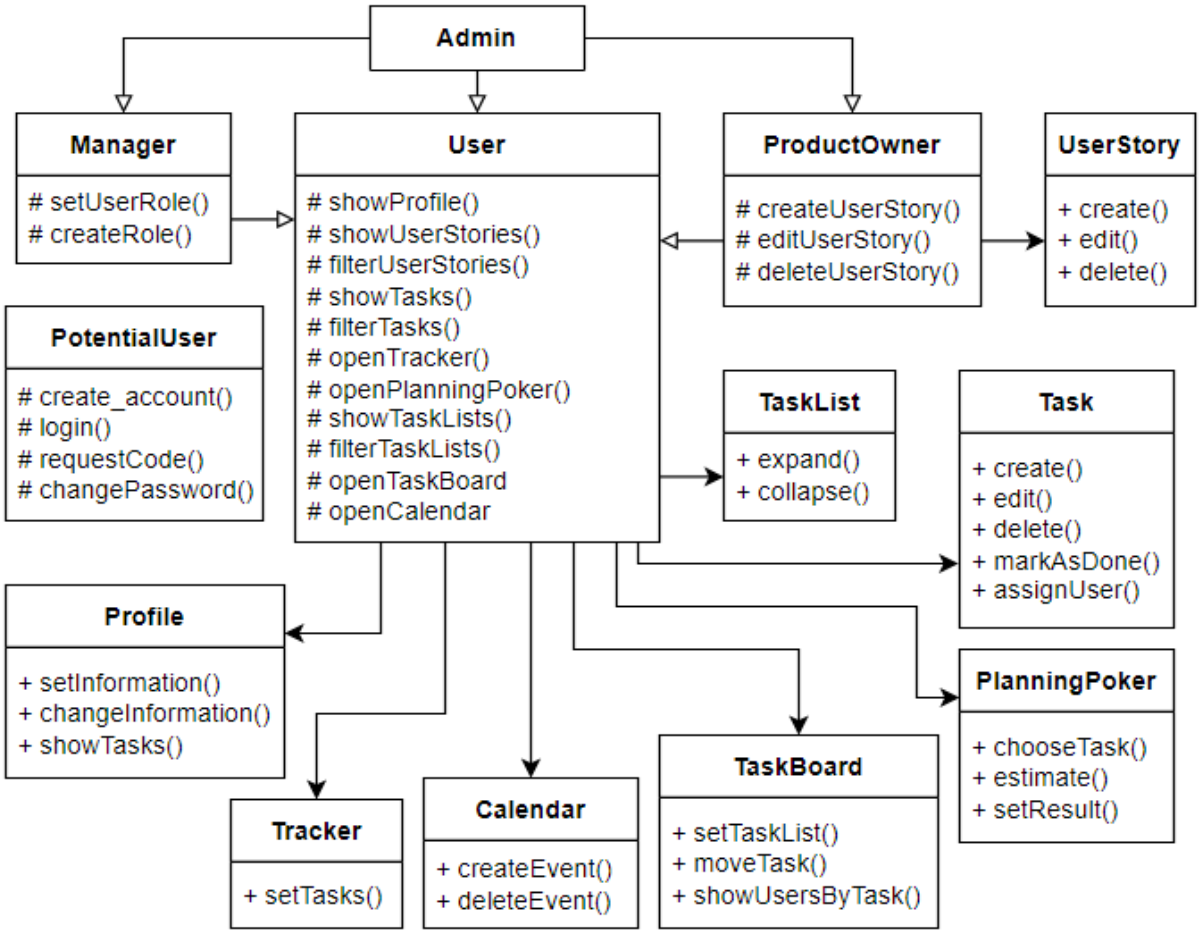
S4 Als Entwickler kann ich (leicht) neue Funktionalität zur Unterstützung weiterer agiler Praktiken hinzufügen, um das System an neue Anforderungen anzupassen.

S5 Als Entwickler kann ich (leicht) eine neue Art von Benachrichtigungen hinzufügen, um das System an neue Anforderungen anzupassen.

S6 Als Entwickler kann ich (leicht) eine neue Filterfunktion für im System filterbare Objekte hinzufügen, um das System an neue Anforderungen anzupassen.



1.11 Klassendiagramm zur Domäne des Projekts



1.12 Bezug zu den ursprünglichen Anforderungen

- Anforderung 1: A{1, 2, 3, 4, 5}
- Anforderung 2: U{1, 2, 3, 4, 5, 6}
- Anforderung 3: T{1, 2, 3, 4, 5, 6, 7}
- Anforderung 4: TL{1, 2, 3, 4, 5}, TB{2, 3}
- Anforderung 5: P{1, 2, 3, 4}
- Anforderung 6: T{8}, TB{1, 2, 5, 6}
- Anforderung 7: T{8}
- Anforderung 8: T{10, 11, 12}
- Anforderung 9: U{7}, T{14, 15}, TL{6}
- Anforderung 10: T{9, 10, 13}
- Anforderung 11: PP{1, 2, 3, 4}, K{3, 4, 5, 6}, TL{5}, TB{2}
- Anforderung 12: S{1}

- Anforderung 13: TB{4}
- Anforderung 14: S{2}
- Anforderung 15: S{3}
- Anforderung 16: S{1}
- Anforderung 17: TL{7}, TB{7}, S{4, 5, 6}
- neue Anforderungen: P{5}, K{1, 2, 7, 8, 9}

  zu P5: Die erfüllten Tasks eines Nutzers auf seinem Profil ansehen zu können, sorgt für einen schnell erreichbaren Überblick über den bisherigen Beitrag eines Nutzers, wodurch Verantwortlichkeiten und Aufgabenverteilungen besser gefunden werden können.

  zu K1: Ein Kalender hilft zur Organisation von Sprints und Meetings, weshalb er als Anforderung eingeführt wurde.

  zu K2: Tasks anhand ihrer Abgabefristen in den Kalender einzutragen, hilft dem Nutzer, um diese herum zu planen.

  zu K7: Das Löschen eines Ereignisses ist nötig, um Ereignisse sinnvoll verwalten zu können.

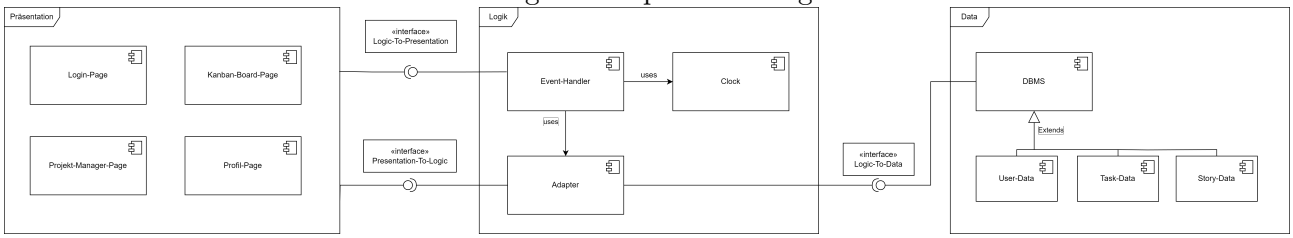
  zu K8: Die Benachrichtigung hilft dem Nutzer, Ereignisse zu berücksichtigen.

  zu K9: Die Benachrichtigung hilft dem Nutzer, Ereignisse nicht zu verpassen.

2 Grobentwurf der Architektur

2.1 Geplantes Architekturmuster und geplante Hauptkomponenten

Abbildung 1: Komponentendiagramm



Für unsere **Architektur** haben wir eine Variation der 3-Schichten-Architektur gewählt (in Abbildung 1 erkennbar), die in eine Präsentations-, Logik- und Datenschicht gegliedert ist. Aus folgenden Gründen haben wir uns für die 3-Schichten-Architektur entschieden: Die Modularität vereinfacht die Entwicklung von Funktionalitäten, da sie auf die einzelnen Komponenten heruntergebrochen werden. Ebenfalls ist die Zerlegung in 3 Ebenen passend zur Gliederung des Teams in Frontend-Entwicklung (Präsentationsschicht), Backend-Entwicklung (Logikschicht) und Datenbank-Entwicklung (Datenschicht), was wiederum die Zuteilung von Tasks erleichtert. Die Einteilung in Schichten ermöglicht eine verbesserte Skalierbarkeit, wodurch nachträgliche Veränderungen, welche zu erwarten sind, einen geringeren Mehraufwand haben werden. Weiterhin begünstigt die Schichteneinteilung eine klare Seperation of Concerns. Da die einzelnen Schichten nur über eindeutig definierte Schnittstellen miteinander kommunizieren, herrscht eine geringe Abhängigkeit zwischen den Hauptkomponenten. Dies führt dazu, dass die Teilkomponenten untereinander die Berechnungen ausführen, sodass über die Schnittstellen jeweils die aktuell benötigten Daten übergeben werden. Dementsprechend folgt dies dem Prinzip High Cohesion, sowie Low Coupling. Aufgrund der definierten Schnittstellen herrscht auch das Prinzip des Information Hiding zwischen den einzelnen Schichten. Dies bedeutet, dass interne Abläufe einer Schicht der anderen Schichten nicht bekannt sind.

Wir wollen uns nicht für die klassische 3-Schichten-Architektur entscheiden, da diese keine Methodenaufrufe von den unteren Schichten an die Oberen erlaubt. Um die Anforderung für Benachrichtigung (Anforderung 10) und Update (Anforderung 12 und 16) umzusetzen, benötigt man die Funktionalitäten *Sync* für Updates und *Notification* für Benachrichtigung. Diese Funktionalitäten sind zeitlich bedingt und somit abhängig von der *Clock*, welche vom *Event-Handler* genutzt wird. Dies kann nicht in der Präsentationsschicht eingebettet werden, weil in dieser Schicht keine Berechnungen möglich sind. Deshalb müssen die Funktionalitäten in eine andere Schicht (in unserem Falle die Logikschicht) ausgelagert werden. Nun kann der *Event-Handler* über das *Logic-To-Presentation-Interface* die Website zu einem Aktualisieren oder Zeigen einer Benachrichtigung erzwingen. Solch ein Interface wäre in der üblichen 3-Schichten-Architektur nicht gestattet, weshalb wir eine Variation der 3-Schichten-Architektur verwenden.

Die **Präsentationsschicht** dient der Visualisierung der Daten in einer nutzerfreundlichen Weise und bietet dem Nutzer die Möglichkeit, mit dem System zu interagieren. Der Nutzer startet auf der *Login-Page*, auf der er sich anmelden oder registrieren kann (Anforderung 1). Bei erfolgreicher Anmeldung gelangt er auf die *Projekt-Manager-Page*. Dort werden alle Tasks und User-Storys des Projekts abgebildet. Es können die Tasks und User-Storys bearbeitet und neue User-Storys erstellt werden (Anforderung 2 und 3). Das heißt die *Projekt-Manager-Page* von hier aus ist die *Kanban-Board-Page* und die *Profil-Page* erreichbar. Auf der *Kanban-Board-Page* sind die Tasks für den Sprint strukturiert in Spalten, welche den Bearbeitungsstand der Tasks darstellen. Sie sind gegliedert in *in Bearbeitung*, *unter Review*, *unter Test* und *fertiggestellt* (Anforderung 4). Es können bei Bedarf weitere Spalten

zur detaillierteren Kategorisierung eingefügt werden. Die *Profil-Page* kann das Profil eines Nutzers eingesehen werden. Dort werden Informationen wie Name, Beschreibung und Rolle abgebildet. Zu Beginn wird das eigene Profil gezeigt, welches man bearbeiten kann, und über einen Reiter kann man die Profile der anderen Nutzer auswählen und betrachten (Anforderung 5). Damit die Aktion des Nutzers eine Auswirkung auf das System hat, haben wir das *Presentation-To-Logic-Interface* von der Präsentationsschicht zum *Adapter*.
Der *Adapter* ist Teil der **Logikschicht** und wandelt die Anfragen der Präsentationsschicht in die semantisch-äquivalente Methoden, die über das Interface *Logic-To-Data-Interface* an die Datenbank vermittelt werden. Neben dem *Adapter* ist der *Event-Handler* und die *Clock* in der Logikschicht eingebunden. Die *Clock* ist die interne Uhr des Systems und prüft, ob eine Notifikation für zeitlich-sensible Aktivitäten wie ein Team-Meeting oder eine Deadline geschickt werden müssen. Der *Event-Handler* befasst sich mit allen zeitlich-abhängigen Funktionalitäten - deshalb verwendet er die *Clock* - wie *Sync* und *Notification*. *Sync* erzwingt ein Updaten der Präsentationsschicht, um die Veränderungen der anderen Nutzer zu integrieren. Dies wird mithilfe der *Clock* alle 2 Sekunden durchgeführt (Anforderung 12 und 16).
Die **Datenschicht** besteht aus einem Datenbankmanagementsystem, kurz *DBMS*, welche die Daten der Datenbanken für die Nutzer, die Tasks und die Storys mit den CRUD-Operationen bearbeiten kann. Mithilfe des *Logic-To-Data-Interface* kann der *Adapter* Daten von der Datenbank abfragen, löschen, verändern und neue hinzufügen.

2.2 Interfaces zwischen diesen Komponenten

Abbildung 2: Presentation-To-Logic-Interface

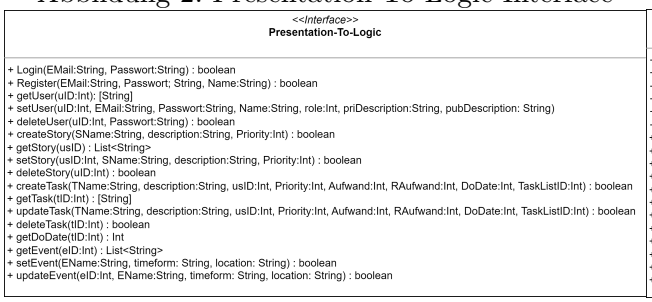
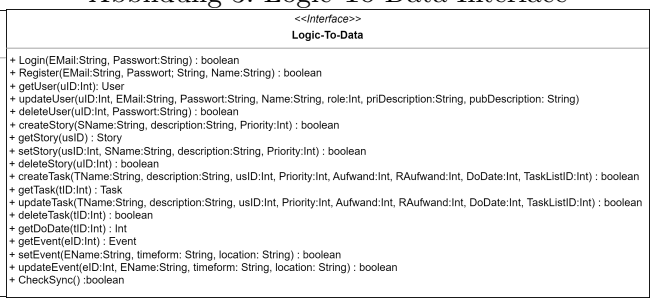


Abbildung 3: Logic-To-Data-Interface



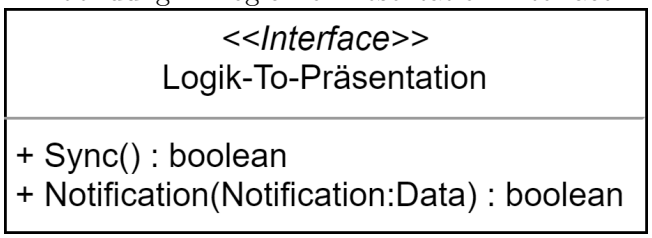
Die Interfaces enthalten allgemein CRUD-Funktionen, welche es dem Nutzer ermöglichen, Einfluss auf die in der Datenschicht gespeicherten Daten zu nehmen. Hierbei steht CRUD für die Funktionen Create, Read, Update und Delete. Diese beschreiben die Basisoperatoren für den Nutzer zur Bearbeitung von Daten. CRUD-Funktionen sind ebendeswegen sinnvoll, weil sie erst überhaupt die Kommunikation zwischen der Datenbank und der Webanwendung ermöglichen. In den folgenden Interfaces wird erklärt, wie die CRUD-Funktionen implementiert werden.

Das *Presentation-To-Logic*-Interface wird von dem Adapter innerhalb der Logikschicht zur Verfügung gestellt, welches der Präsentationsschicht Funktionen zur Übergabe und Anfrage von Parametern an die Logikschicht bietet. Hierzu werden bei Eingabe des Users oder Veränderungen durch den User Datenwerte an die Logikschicht übertragen, sodass diese anschließend mit diesen weiterarbeiten kann. Für die gewünschten Veränderungen des Users dienen die CRUD-Operationen innerhalb des Interfaces, um direkt Tasks, User-Storys oder den Nutzer selbst zu erstellen, zu löschen oder zu bearbeiten (create(), update(), delete()). Hierbei sollen die Anforderung 2, Anforderung 3 als auch das Editieren des Profils in Anforderung 5 erfüllt werden. Ebenfalls existieren in dem Interface Get- und Set-Funktionen, welche die eingegebenen Veränderungen für User, Tasks und Events der Präsentationsschicht ent-

nehmen beziehungsweise auf einen anderen Wert setzen können. Diese sind notwendig, um für die Logikschicht Daten abzurufen und auf einen anderen Wert zu setzen. Hierbei beschreibt ein Event einen Kalendereintrag eines Users, welches Ereignisse wie Meetings und Abgaben beschreibt. Hierzu gehört ebenso die Get-Methode zu den Fristdaten (GetDoDate()), welche von der Präsentationsschicht an die Logik weitergeleitet werden. Diese sind nötig, um Daten aus der Präsentationsschicht an die Logikschicht zu senden. Zuallerletzt hat das Interface die Methoden Register und Login, welche es dem User ermöglichen, sich in der Webanwendung zu registrieren beziehungsweise anzumelden, wie es in Anforderung 1 erforderlich ist.

Das *Logic-To-Data*-Interface wird von der Datenschicht zu der Logikschicht gestellt. Das Interface wird benötigt, um eine Kommunikation mit der Datenschicht zu ermöglichen. Die Daten werden anschließend in der Datenbank gespeichert, verändert, oder an den Adapter in der Logikschicht übergeben. Dafür werden CRUD-Funktionen verwendet, mit welchen Tasks und User-Storys erstellt, bearbeitet und gesucht werden können. Hierzu gehören auch Get- und Set-Methoden, mit welchen für die Datenschicht die benötigten Datenwerte mit dem Get-Operator abgerufen werden, um diese in der Datenbank mit der Set-Methode zu verändern. Diese beziehen sich auf die Tasks, User-Storys, User, Events und Fristdatum. Eine zusätzliche Aufgabe des Interfaces ist es, die Synchronisation zwischen der Logik- und Datenschicht zu überprüfen, sodass diese stets aktuell bleibt. Hierfür wird die Funktion checkSync() implementiert, mit welcher die Logikschicht periodisch prüft, ob Daten in der Datenbank verändert wurden, beispielsweise durch neue Updates der Entwickler. Wenn Daten geändert wurden, so gibt die Datenschicht ein "True"-Responce zurück.

Abbildung 4: Logic-To-Presentation-Interface



Die Präsentationsschicht stellt durch das *Logic-To-Presentation-Interface* der Logikschicht, insbesondere dem *Event-Handler* Methoden zur Verfügung. Die Methoden *Sync()*, welches die Aktualität der Nutzersicht sicherstellt, und *Notification()*, das den Nutzer über zeitlich gebundene Aktivitäten informiert, decken die Anforderungen 10, 12 und 16 ab. Bevor mittels *Sync()* die Webpage zum Aktualisieren gezwungen wird, prüft der *Event-Handler* durch Polling der Datenbank (im 2 Sekunden Abstand), ob eine Veränderung seit der letzten Abfrage aufgetreten ist. Sobald die Abfrage als true - das heißt es gab eine Veränderung - ausgewertet wird, wird *Sync()* ausgeführt, woraufhin die Präsentationsschicht Information der aktuellen Nutzersicht mittels Getter-Methoden der *Presentation-To-Logic-Interface* abfragt. Die Methode *Notification()* wird aufgerufen, wenn der *Event-Handler* unter Nutzung des *Adapters*, getaktet mittels der *Clock* durch Polling der Datenbank erkennt, dass eine zeitlich bedingte Aktivität demnächst, heißt in 2 Stunden bis hin zu einem Tag, stattfindet. Sobald ein Termin eine zeitliche Grenze überschritten hat, wird der *Event-Handler* durch die *Clock* informiert, welcher Termin in welchem Zeitabstand passiert. Diese Information sind die Argumente der *Notification()*, die an die Präsentationsschicht geschickt wird, woraufhin auf der Webpage eine Push-Benachrichtigung auftaucht.

Grobentwurf Ablauf von bestimmten Ereignissen

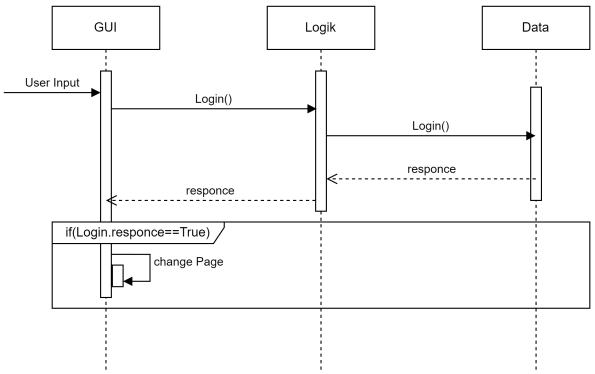


Abbildung 5: Sequenzdiagramm Login

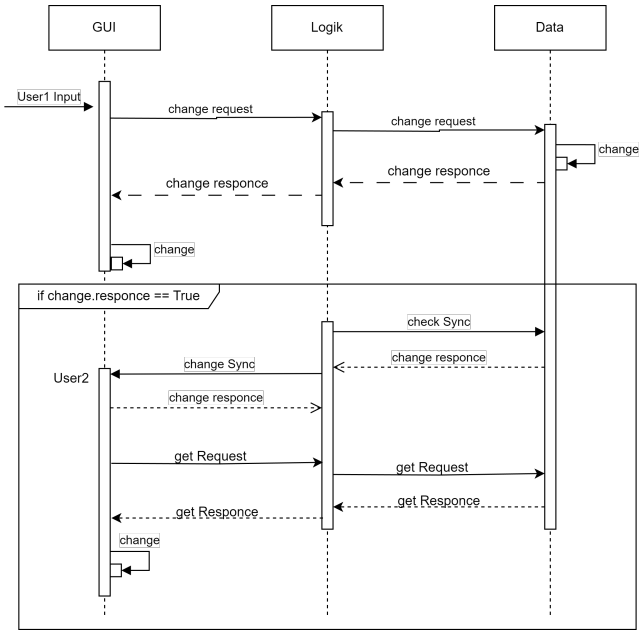


Abbildung 6: Sequenzdiagramm Synchronisation

Zur Beschreibung der ablaufenden Prozesse innerhalb der Webapplikationen wurden folgende drei Sequenzdiagramme gewählt, da diese fundamentalen Funktionen der Applikation darstellen. Hierbei wurde der Login-Prozess (Abbildung 5) repräsentativ für alle CRUD-Funktionen gewählt, da der grundlegende Ablauf dieser Funktionen in unserer Applikation gleich zu beschreiben ist. Der Synchronisations-Prozess (Abbildung 6) wurde gewählt, um die Anforderung 12 zu berücksichtigen und das Verhalten der Applikation für mehrere Nutzer zu modellieren. Und letztlich wurde der Notification-Prozess gewählt, um Anforderung 10 zu repräsentieren und zu beschreiben, wie anstehende Events in dieser Applikation abgearbeitet werden.

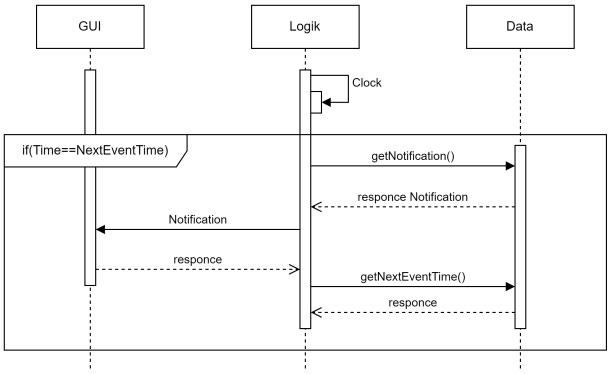
Wie im Sequenzdiagramm 5 zu sehen, übermitteln die CRUD-Funktionen Nutzer-Daten aus der Präsentationsschicht an die Logikschicht. Durch den *Adapter* in der Logikschicht wird dann die Anfrage des Nutzers umgeformt an die Datenschicht weitergegeben. Ist der Vorgang, also die Speicherung oder Überprüfung der Daten, beendet, so reagiert die Datenschicht mit einer Responce an die Logikschicht. Diese Responce entspricht dem im *Logic-To-Data-Interface* (Abbildung 3) beschriebenen Rückgabe-Datentypen. Die Responce wird daraufhin durch den *Adapter* zurück an die Präsentationsschicht übermittelt, in Form einer Rückgabe des intialen Methodenaufrufs der Präsentationsschicht, und führt dort zu Änderungen in der betrachteten Seite durch. In Bezug auf die Login-Funktion sowie create, update und delete Funktionen erfolgt die Änderung der Nutzersicht nur bei einer "True"-Responce, um Inkonsistenten zwischen den auf der Webpage angezeigten und in der Datenschicht gespeicherten Informationen zu vermeiden.

Der Synchronisation-Prozess von zwei oder mehr Nutzern wird durch das Sequenzdiagramm 6 beschrieben. Dieser Prozess zeigt die Folgen aller create, update und delete Funktionen (hier vereinfacht durch *change request* dargestellt) auf andere aktive Nutzer (im Diagramm *User2*) der Anwendung. Durch zu speichernde Inputs eines Nutzers wird ein *Change-Request* an den *Adapter* der Logikschicht gestellt. Nach Umformung dieses Requests durch den *Adapter* wird sie daraufhin an die Datenschicht weitergegeben, welche sie abarbeitet und die Änderungen permanent speichert. Die Datenschicht übermittelt daraufhin eine Change-Responce vom Typ "Boolean" an den *Adapter*. Die Responce wird

von dem *Adapter* zur Präsentationsschicht des Nutzers überreicht, sodass eine Änderung der angezeigten Daten hervorgerufen wird. Bei einem "False"-Responce wird dem ersten Nutzer eine Fehlermeldung angezeigt, da das Speichern der Daten fehlgeschlagen ist, um inkonsistente Informationen zwischen Präsentationsschicht und Datenschicht zu vermeiden. Anschließend wird eine "True"-Responce in der Datenschicht vermerkt. Dieselbe Responce sorgt beim ersten Nutzer für eine Änderung seiner eigenen Webansicht. Falls die *Change-Responce* "False" ist, wurde keine Veränderung in der Datenbank hervorgerufen und es wird auf der Nutzersicht ein Fehlercode eingeblendet.

Die Logikschicht aller weiteren Nutzer prüft mit *check Sync* auf Änderungen der Daten in der Datenschicht, repräsentiert durch der vorherigen "True"-*Change-Responce*. Wenn Änderungen in den Daten vorliegen, wird eine "Sync"-Request über das *Logic-To-Presentation-Interface* an die Präsentationsschicht des Nutzers übermittelt. Dies führt zu einem *get-Request* durch die Präsentationsschicht des Nutzers über das *Presentation-To-Logic-Interface* an den *Adapter* der Logikschicht. Nach Umformung des Requests durch den *Adapter* wird dieser an die Datenschicht über das *Logic-To-Data-Interface* übermittelt. Nach Bearbeitung des *get-Request* in der Datenbank, wird ein *get-Responce* an die Logikschicht und darauf zur Präsentationsschicht und die Veränderung wird auf die Webpage ausgeführt.

Abbildung 7: Sequenzdiagramm Notification



Die internen Benachrichtigungen werden in Abbildung 7 beschrieben und werden genutzt, um den Nutzer über vorstehende *DoDates* und anderen Events zu informieren (Anforderung 10). Hierbei wird die *Clock* angewendet, um periodisch zu prüfen, ob die aktuelle Zeit des Nutzers mit der nächsten anstehenden Event-Zeit, also *DoDates* und Kalendereinträge, übereinstimmt. Wenn die aktuelle Zeit sich der gespeicherten *EventTime* nähert, wird die entsprechende Nachricht aus der Datenschicht über das *Logic-To-Data-Interface* angefragt. Die Datenschicht übermittelt die Daten des zur *EventTime* gehörenden Events an den *Adapter*. Nach Umformung der Daten durch den *Adapter* werden die Daten als Nachricht an die Präsentationsschicht weitergegeben und dort dem Nutzer angezeigt. Die Responce der Präsentationsschicht ist dann der Auslöser einer weiteren Request der Logik über das *Logic-To-Data-Interface* an die Datenschicht. Hierbei wird die Zeit für das nächste Event angefragt und als Responce der Datenschicht an die Logik übermittelt und in der *Clock* gespeichert.

2.3 Datenstrukturen

Grundlegende Datenstrukturen des Systems

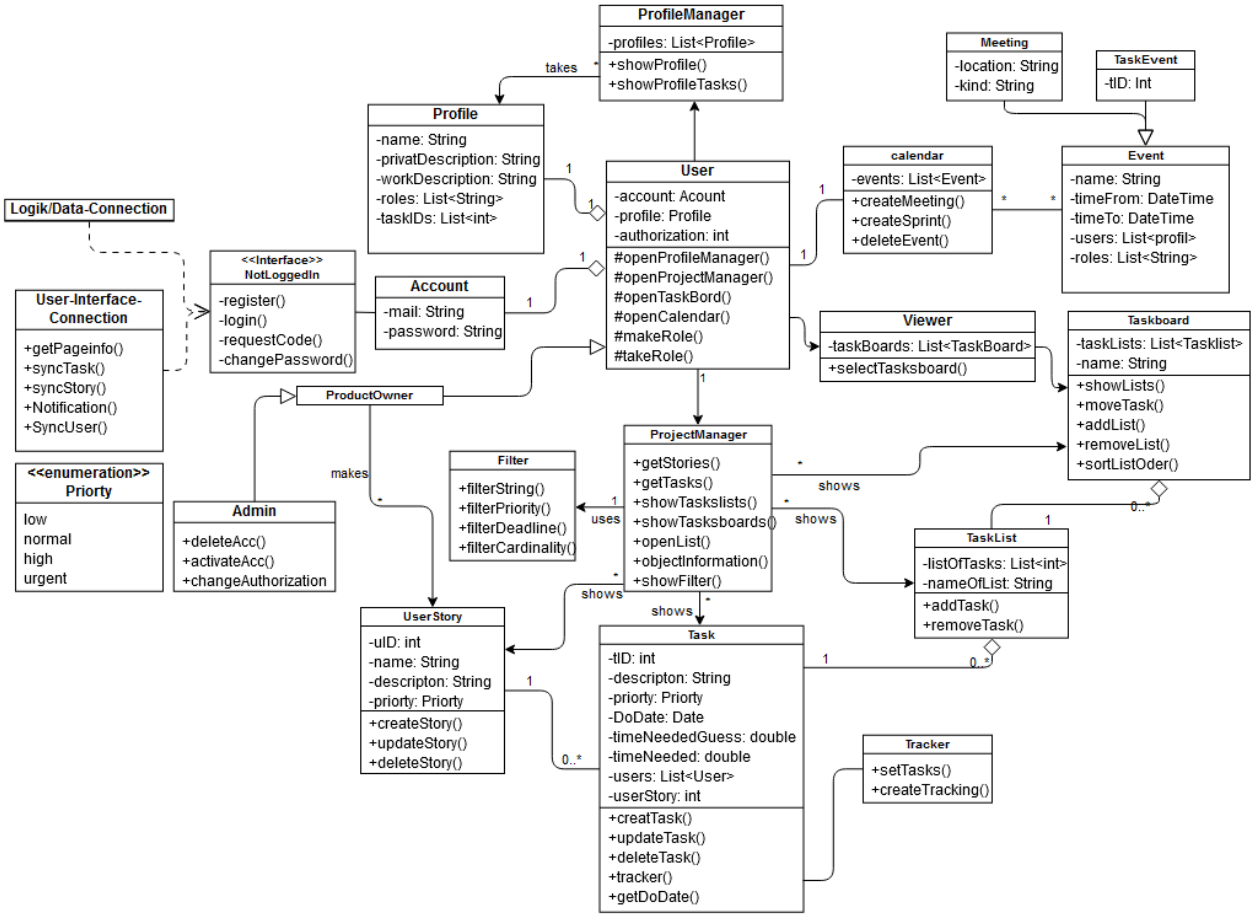


Abbildung 8: Klassendiagramm der grundlegenden Datenstrukturen

Wie dem Klassendiagramm 8 zu entnehmen ist, haben wir uns für eine zentrale Klasse, den *User*, entschieden, da diese den Großteil der Systemnutzung beansprucht. Der User unterteilt sich neben seinen Funktionalitäten in *Account*, *Profile* und der *Authorization*, somit trennen wir die sensiblen Daten des Accounts von den Daten, die im normalen System zur Verfügung stehen schon im Konzept. Dies fördert die Modularität und einfache Anpassbarkeit für den Account und seinen Daten.
Durch die Authorization wird gespeichert, welche Rechte der User im System besitzt. Dabei kann es sich um einen User, einen Product Owner, oder einen Admin handeln, wobei ein Product Owner eine Erweiterung von User und Admin eine Erweiterung vom Product Owner ist.
Wenn wir den Account nun näher betrachten, ist dies die Klasse, in der die E-Mail, welche einzigartig sein muss, und das Passwort gespeichert werden. Sie können von außen durch die Funktionen des *NotLoggedIn-Interface* kreiert und durch die Klasse Admin, welche vom User erbt, freigeben und gelöscht werden. Das *NotLoggedIn-Interface* wird durch die Schnittstelle zwischen der Logik- und der Datenschicht, sowie der Schnittstelle zwischen der Präsentations- und der Logikschicht implementiert. Somit werden die Methoden vorausgesetzt, jedoch können die unterschiedlichen Schnittstellen aufgrund der verschiedenen Anforderungen, diese anders umsetzen.
Durch den Admin stellen wir sicher, dass User nicht aus Versehen ihren Account löschen können. Wei-

terhin muss der Admin einen erstellten Account freigeben, bevor ein User zu diesem erstellt werden kann, so können gezielt Nutzer dem System hinzugefügt werden. Der Account existiert somit etwas abseits vom System, wodurch eine Modularität und eine einfache Austauschbarkeit des Logins und den geforderten Daten gewährleistet werden kann.
Ein Nutzer, der nicht eingeloggt ist, kann sich über den Login in seinen Account anmelden, weiterhin kann er an der gleichen Stelle einen Sicherheitscode anfordern, welcher an die hinterlegte E-Mail-Adresse gesendet wird, um das Passwort zurückzusetzen. Durch diese Entscheidungen setzen wir die Anforderung 1. Ein User steht mit einem Account, sowie mit einem *Profile* über eine Aggregation in Beziehung. Dabei kann ein Account Objekt ohne User kreiert werden, sowie ein Objekt vom Typ *Profile* auch ohne User erstellt werden kann. Datenbanktechnisch ist jedoch ein User von den Accountdaten und ein *Profile* von den Userdaten abhängig. Wird der Account eines Users durch einen Admin gelöscht, so wird der Account, sowie der User und das *Profile* aus der Datenbank gelöscht und kann somit auch nicht mehr als Objekt vorhanden sein, es sei denn, dass der Datensatz erneut im System importiert wird. Ein *Profile* besteht aus einem einzigartigen Namen, einer Rolle, die Liste der IDs aller dem *Profile* zugewiesenen Tasks, sowie zwei Beschreibungen, welche kreativ durch den User gefüllt werden können. Somit wird die Anforderung 5 erfüllt. Durch die IDs der zugewiesenen Tasks ist es klar, welche Tasks zum *Profile* gehören, wobei man den Vorteil behält, dass das eigentliche *Profile* Objekt nur so groß, wie nötig ist.
Eine Rolle ist dabei einfacher String, welche zum derzeitigen Zeitpunkt noch nicht klarer eingegrenzt werden kann und im Laufe des Projektes auf eine Liste an Standard-Werten wachsen soll. Eine Rolle soll als eine bestimmte Zuweisung, welche die eigentlichen User schnell in Teams beziehungsweise Themen sortieren kann, fungieren. Dabei kann ein User mehrere Rollen haben und diese selber kreieren, sowie sich selber zuordnen. Dabei kann jedoch keine Rolle erstellt werden, die durch die *Authorization* abgebildet ist. Zu diesem Zeitpunkt können also keine Rollen mit den Titeln: *Admin*, *Product Owner* und *User* erstellt werden. Somit soll es durch die Rolle weiterhin möglich sein ein User seinen Teams bzw. mehreren Themen oder Spezialisierungen ohne viel Aufwand zuzuweisen.
Die Beschreibungen sind durch den Profil-Manager, welcher einen User durch die Klasse *ProfileManager* angezeigt werden kann, auch für andere User einzusehen. Sie sind der Ort für zusätzliche Informationen, welche so leicht im System hinterlegt werden können, wie zum Beispiel der derzeitige Arbeitsstatus, Spezialisierungen oder besonderes Wissen in bestimmten Themengebieten. Eine Beschreibung ist dabei mehr für persönliche Informationen, welche durch das Attribut *privatDescription* abgebildet wird, während die Andere für projektbezogene Informationen, welche hingegen im Attribut *workDescription* gespeichert wird, ausgelegt ist.
Die Klasse *ProfileManager* stellt weiterhin nicht nur die Funktion zum Einsehen der verschiedenen Profile im System bereit, sondern auch die Möglichkeit sich die zugeordneten Task zu den einzelnen Profilen anzeigen zu lassen. Dies fördert die Transparenz des Projektes und verspricht Kommunikationswege zu kürzen, bzw. unnötige zu vermeiden, da so die Beschreibungen, sowie die Rollen der unterschiedlichen Nutzer, einzusehen sind. Mit diesen Informationen können direkt die entsprechenden Experten der Themengebiete, beziehungsweise der Teams, kontaktiert werden, ohne dass sich groß anderweitig erkundigt werden muss.
Der User vererbt seine Attribute, sowie seine Funktionalität an die Klasse *ProductOwner*. Die Klasse *Admin* kann Accounts freigeben, wodurch zum Account ein User erstellt werden kann, sowie Accounts löschen, das wie zuvor erwähnt zur datenbankseitige Vernichtung der Daten des gesamten Nutzers führt. So wird gewährleistet, dass das System nicht durch neue Nutzer überflutet wird, sowie das Entfernen von Nutzern leicht und unkompliziert zur Verfügung gestellt wird. Die Klasse *ProductOwner* hingegen kann Objekte der Klasse *UserStory* erstellen, zerstören, sowie angepassen.
User-Storys sind die Grundsteine des Systems. Die Klasse *UserStory* umfasst daher eine einzigartige ID, sowie einen einzigartigen Namen, durch die sie klar zu referenzieren sind, eine Beschreibung, in der die eigentliche User-Story steht, sowie eine Priorität, die ihre Wichtigkeit für das derzeitige Pro-

jekt ausdrückt. Durch die Implementation von *ProductOwner* und *UserStory* kann die Anforderung 2gewährleistet werden.
Auf Grundlage dieser User-Storys werden Tasks erstellt, welche durch die Klasse *Task* mit einer einzigartigen ID, eine Beschreibung, eine Priorität, ein *DoDate*, sowie eine Aufwandschätzung (*timeNeededGuess*), den reellen Aufwand (*timeNeeded*), eine Verknüpfung zur zugehörigen User-Story, sowie eine Zuordnung von Usern, welche derzeit diese Task bearbeiten, repräsentiert werden. Durch die Liste der User wird die Anforderung 6 erfüllt. Im gleichen Atemzug beschreibt die Liste der User an den Task auch eine Möglichkeit Pair-Programming zu kennzeichnen, wodurch die Anforderung 7 umgesetzt werden kann, während dessen die Tasks selber als eine Verfeinerung der User-Storys zu verstehen sind, welche die Anforderung 3 befriedigt. Die Priorität beschreibt nicht nur die Wichtigkeit der einzelnen Task, sondern auch die Position in der *TaskList*, in welche sich die Task befindet. Das *DoDate* ist der späteste Zeitpunkt, an welchem die Task aus ihrer aktuellen *TaskList*, in die nächste, dem *TaskBoard* zugeordnete, geschoben werden soll.
Die Klasse *Tracker* greift auf die unterschiedlichen Tasks zu, um durch diese eine Visualisierung zur allgemeinen Bearbeitungszeit aufgrund der Diskrepanz zwischen der Aufwandschätzung und dem reellen Aufwand von Tasks zu erstellen. Durch den Tracker eröffnet das System einen Ort zur einfachen Auswertung der Arbeit an den schon vorangegangenen Sprints, bzw. dem Projekt als Ganzen. Durch den Tracker wird die Anforderung 8, des Schätzungstracking nicht nur erfüllt, sondern auch noch erweitert. Die Tasks werden in den unterschiedlichen Listen auf dem Kanban-Board zusammengefasst. Diese werden durch die Klasse der *TaskList* dargestellt. Die Klasse enthält den Namen der Liste, sowie die Liste von Tasks, welche ihr zugeordnet sind. Die unterschiedlichen Objekte der *TaskList* können den gleichen Namen enthalten, jedoch können keine zwei *TaskListen*, welche den gleichen Namen speichern, in ein und dasselbe *TaskBoard* hinzugefügt werden. Ein *TaskBoard* repräsentiert das gesamte Kanban-Board und zeichnet sich durch die Liste der *TaskLists*, sowie einen einzigartigen Namen aus. Durch den einzigartigen Namen von *TaskBoards*, sowie *TaskListen*, innerhalb eines *TaskBoards*, kann klar zwischen den unterschiedlichen Objekten differenziert werden, was die Verwechslungsgefahr in den unterschiedlichen Auswertungen verringert.
Durch das Zusammenspiel von Taskboard und Projektmanager kann die Anforderung 4 realisiert werden.
Der *ProjektManager* ist eine Klasse, welche die Funktionalitäten einer Übersicht über *TaskBoards*, *TaskLists*, Tasks und User-Storys implementiert. Die Abstraktion des ProjektManagers vom User soll Übersichtlichkeit, sowie Schlichtheit im Code und in die Architektur bringen, da so ein zentraler Punkt dafür entsteht. In diesem wird die einfache Suche nach bestimmten Kriterien, sowie Tasks zur Verfügung gestellt. Dazu wird aus den gleichen Gründen, eine ausgelagerte Klasse *Filter* kreiert, welche sich insbesondere mit der Sortierung und Filterung der einzelnen Übersichts-Objekte beschäftigt und so das Suchen und Filtern für den Nutzer, wie durch die Anforderung 9 ermöglicht. So kann ein Nutzer die unterschiedlichen Tasks, die durch diese Klasse gestellten Funktionen, filtern, sowie bestimmte suchen.
Um den zeitlichen Rahmen genauer erfassen zu können, haben wir uns für eine Klasse *Calendar* entschieden, welche die unterschiedlichen Events, als Liste, hält. Ein Event soll somit einen Eintrag in dem Kalender repräsentieren. Hier unterscheiden wir in Meetings, Sprints, sowie Deadlines für Tasks. Die Klasse *Event* definiert Events als einen Namen, ein Anfangszeitpunkt und einen Endzeitpunkt, sowie einer Liste von Usern und einer Liste an Rollen, die an dem Event teilnehmen sollen. Ein *Meeting* erweitern somit das Event um einen Ort, sowie eine Art des Meetings, während hingegen ein *TaskEvent*, das Event um eine zugehörigen Task Identifier erweitert. Sprints können dabei durch die Klasse Events selbst repräsentiert werden.
Dabei werden die Events gespeichert, um die unterschiedlichen Termine im Kalender zu vertreten. Weiterhin kann durch die explizite Speicherung der Events, das System, auch die Gestaltung der individuellen Kalender der unterschiedlichen User ermöglichen.

So werden die Events, welche zu bestimmten Usern, beziehungsweise Rollen von Profilen der User, zugeordnet sind, direkt dem Kalender der bestimmten User zugeordnet. Durch das Speichern von diesem Zusammenhang wird dem User weiterhin ermöglicht, den Kalender individuell anzupassen und etwaige Termine aus dem Kalender zu entfernen, ohne dass dies die anderen User beeinflusst. Des Weiteren finden sich diese Termine auch nach dem Neuladen des Kalenders nicht mehr in diesem vor. Die Funktionalitäten der *Logik-Connection* und der *Data-Connection-Interface* sind in den einzelnen Klassen umgesetzt und können durch diese Interfaces nach außen weitergegeben werden. Somit fungieren die Interfaces als Schnittstellen zwischen den verschiedenen Architekturebenen und stellen die Funktionalitäten den anderen Ebenen bereit. Die Komponenten des Grobentwurfs sind ebenfalls in den Klassen integriert und setzen damit die angedachten Charakteristiken um. Die Anforderungen der einzelnen User-Storys sind in ihrer Kategorie klar zu den einzelnen Komponenten und Klassen des Systems eingegrenzt.

Speicherung der Datenstrukturen vom System

Für die Speicherung der Daten wird eine SQLite Datenbank verwendet. Diese ist sehr eingeschränkt in ihren Datentypen, daher verwenden wir hauptsächlich nur die Datentypen Integer, sowie Text. Den Grobentwurf der Datenbank wird durch das Datenbankmodell, Abbildung 9, gut abgebildet. Dabei wird die Struktur der Klassen vernachlässigt und auf eine einfache Struktur zur Speicherung gesetzt. Die Daten werden aufbereitet, um dann in den beschriebenen Klassen dem System zur Verfügung zur stehen.

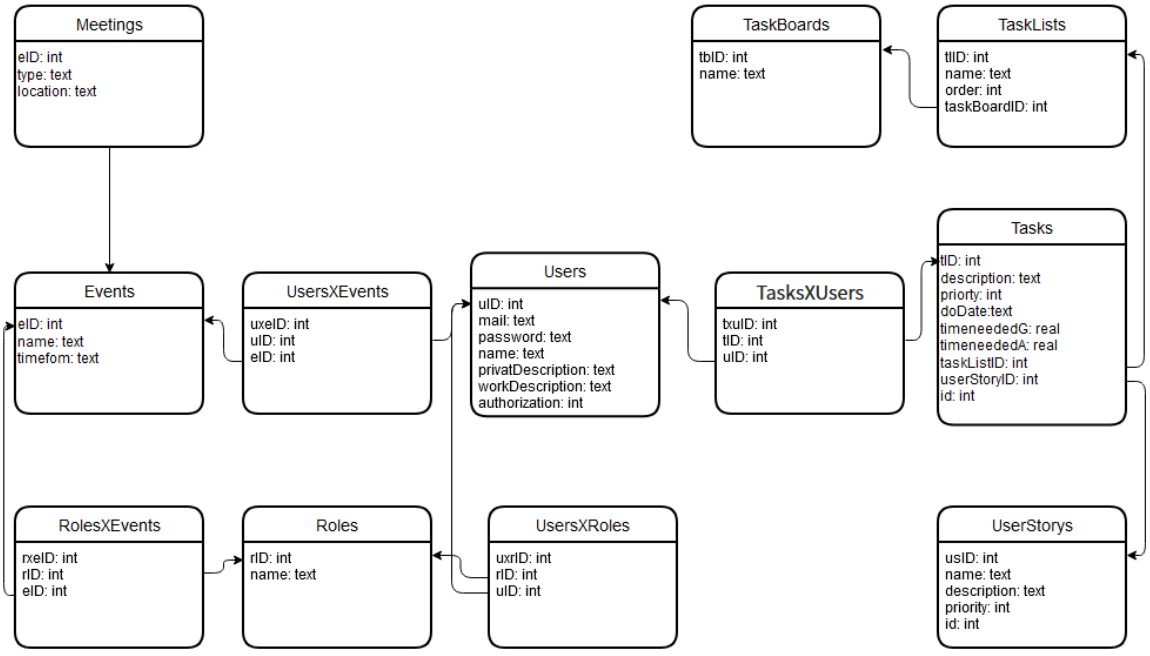


Abbildung 9: Grobentwurf zum Datenbankmodell

3 Technologien

Grundlegend ist geplant, das Projekt im Backend mit Java umzusetzen, da bereits einige Programmiererfahrung durch das Studium vorhanden ist. Des Weiteren ist die automatische Garbage-Collection vom Vorteil, da im Vergleich zu C beispielsweise der Speicher nicht manuell freigegeben werden muss und der Code somit automatisch optimiert wird. Außerdem kann Java plattformübergreifend genutzt werden, wodurch der Server nicht auf einem Betriebssystem beschränkt ist.

Für die Implementation der Applikation planen wir uns mit Servlet zu nutzen, da es eine weit verbreitete Implementierungsform für auf Java basierende Webapplikationen ist. Darüber hinaus erleichtert die Servlet-Struktur und Servlet-Pakete eine vereinfachte Behandlung von HTTP-Requests und Responces. Dazu wird Tomcat genutzt, da es uns erlaubt Webanwendungen der Servlet-Struktur auszuführen.

Für das Frontend nutzen wir HTML und JavaScript. Um uns aber die Arbeit mit diesen zu erleichtern, nutzen wir das Webframework Foundation. Es stellt bereits eine automatische Anpassung der Seite auf jeweilige Fenstergrößen verschiedener Geräte zur Verfügung und bietet einige Designvorlagen für Objekte und Komponenten, die optional verwendet werden können.

Und letztlich zum Speichern unserer Daten wollen wir Hibernate als Schnittstelle zwischen Backend und Datenbank verwenden, da es als ORM die Arbeit mit Datenbanken stark vereinfachen und eine große Anzahl an Datenbankimplementierungen unterstützt.