Einzelbeispiel (SE&PM/JAVA), Sommersemester 2019

Bitte lesen Sie dieses Dokument aufmerksam und bis zum Ende durch, bevor Sie mit der Arbeit beginnen. Nur so können Sie sicherstellen, dass Sie die gesamte Angabe verstanden haben!

Um an der Gruppenphase der Lehrveranstaltung Software Engineering & Projektmanagement teilnehmen zu können, muss das Einzelbeispiel **selbständig** erfolgreich gelöst werden.

Zu verwendende Technologien

Programmiersprache	Java OpenJDK 11
Java-Framework	Spring Boot 2.1.x
Datenbank	H2 1.4.x
Logging	Apache Commons Logging 1.2.x
Test-Framework	JUnit 4.x.x
Build & Dependency Management	Maven 3.5.x
Versionierung	Git 2.x.x

Als Entwicklungsumgebung empfehlen wir IntelliJ IDEA - Community Edition 2018.x.x[1]. Sämtliche Tools stehen im Informatiklabor zur Verfügung und unsere TutorInnen bieten hierfür Unterstützung an.

Betreuung durch unsere Tutor/innen während der Eingangsphase

Bei Fragen und Unklarheiten während der Eingangsphase stehen Ihnen unsere TutorInnen per TUWEL Diskussionsforum zur Verfügung. Zusätzlich gibt es betreute Laborzeiten, zu denen unsere TutorInnen im Informatiklabor anwesend sind, um vor Ort bei der Problemlösung behilflich zu sein. ***Wichtig:*** *Je genauer Sie Ihre Fragen formulieren, desto besser kann Ihnen geholfen werden.*

Labor Fragestunden: Termine und Anwesenheitszeiten finden Sie im TUWEL.

1 Erwartete Vorkenntnisse

Fachliche und methodische Kompetenzen:

- Objektorientierte Analyse, Design und Programmierung
- Grundlagen der Unified Modeling Language (UML)
- Grundkenntnisse aus Algorithmen und Datenstrukturen
- Grundkenntnisse zu Datenbanksystemen

Kognitive und praktische Kompetenzen:

- Eine praxisrelevante Programmiersprache und -werkzeuge (z.B. Java oder C++) anwenden
- Eine IDE und Quellcodeverwaltung anwenden

[1] https://www.jetbrains.com/idea/download/

Inhaltsverzeichnis

2 Angabe

Sie sind als Softwareentwickler/in in einem kleinen österreichischen Unternehmen tätig. Ihr Unternehmen wurde beauftragt ein REST-Backend für *Wendy's Rennpferde* zur Verwaltung und Durchführung von Pferderennsimulationen zu entwickeln. Die Entwicklung einer graphischen Oberfläche ist im Projektauftrag nicht enthalten.
Die Anforderungsanalyse des Gesamtsystems wurde bereits durchgeführt und die Wünsche des Kunden in Stories erfasst. Ihr/e technische/r Architekt/in und Ihr/e Qualitätsmanager/in haben zusätzliche Anforderungen an die Umsetzung der Software formuliert. Außerdem wurde bereits die REST-API, die den Zugriff auf das Backend ermöglicht, definiert. Es wird von Ihnen erwartet, dass sich die REST-API exakt so, wie in der API Spezifikation beschrieben, verhält. Userstories, die von einer fehlerhaften Implementierung betroffen sind, werden nicht abgenommen. Um sicher zu gehen, dass die minimalen Anforderungen an das Backend erfüllt sind, hat Ihnen der Kunde vorab eine Testsuite, zum Testen der REST-API, zur Verfügung gestellt.
In der Aufgabenstellung wird zwischen Userstories und Techstories unterschieden. Userstories beschreiben konkrete Anwendungsfälle der zu erstellenden Software aus Sicht eines bestimmten Stakeholders. Techstories enthalten Anforderungen an die Implementierung sowie weitere Implementierungsdetails.

2.1 Domänenmodell

Um die Domäne etwas zu veranschaulichen, wurde bereits ein Domänenmodell erstellt.

2.2 Dokumentation

Im Rahmen der Stories werden Sie Code-Dokumentation erstellen.

Wichtig: *Bitte drucken Sie die Code-Dokumentation nicht aus!*

Außerdem müssen Sie eine Stundenliste führen, in der Sie Datum und Dauer sowie die Story an der Sie arbeiten festhalten. Diese müssen Sie ausgedruckt zum Abgabegespräch mitnehmen. Vermerken Sie zudem Ihren Namen und Ihre Matrikelnummer darauf.

2.3 Implementierungsreihenfolge

2.3.1 Userstories

Am besten implementieren Sie vertikal nach Userstories, alle Tests und Schichten einer Userstory sollten fertig sein, bevor Sie mit der nächsten beginnen. Achten Sie immer darauf, dass die API Spezifikation eingehalten wird. Natürlich können Sie später auch bei bereits implementierten Userstories nacharbeiten. Dieses Vorgehen hilft Ihnen dabei möglichst viele der Userstories abzuschließen.

2.3.2 Techstories

Techstories haben Einfluss auf die Implementierung aller Userstories. Sie sind daher laufend während der gesamten Entwicklung zu berücksichtigen.

2.4 Implementierung

Ihr/e technische/r Architekt/in hat zusätzlich zu den Anforderungen verschiedene Hilfestellungen im Abschnitt Implementierung für Sie zusammengestellt um Ihnen den Aufbau des Pro-

gramms zu erleichtern.

2.5 Userstories (80 Storypoints)

Userstories stellen Anforderungen eines Stakeholders an das Gesamtsystem dar. Jede Userstory muss dabei in allen Schichten des Backends (REST, Businesslogik, Persistenz) implementiert werden.

Wichtig: *die Aufschlüsselung der Userstories auf Stakeholder dient ausschließlich dem leichteren Verständnis. Sie sollen kein Mehrbenutzersystem entwickeln. Weiters wird in der Einzelphase auch auf Login bzw. Authentifizierung verzichtet.*

2.5.1 Pferdebesitzer/in

ID: 01	**Titel:** Als Pferdebesitzer/in möchte ich neue Pferde im System anlegen können.

- Jedes angelegte Pferd muss in einer Datenbank gespeichert werden.
- Zu jedem Pferd müssen dabei folgende Werte gespeichert werden:
  - Name (verpflichtend)
  - Pferderasse (optional)
  - Zeitpunkt der Erstellung (automatisch)
- Relevante HTTP Anfragen:
  - POST Horse

Storypoints: 6

ID: 02	**Titel:** Als Pferdebesitzer/in möchte ich die Minimal- und Maximalgeschwindigkeit eines Pferdes im System erfassen können.

- Jedes Pferd muss eine Minimal- und Maximalgeschwindigkeit haben. Wobei gilt: min. und max. Geschwindigkeit müssen zwischen 40 und 60 km/h liegen.
- Relevante HTTP Anfragen:
  - POST Horse

Storypoints: 2

ID: 03	**Titel:** Als Pferdebesitzer/in möchte ich Pferde bearbeiten können.

- Beim Bearbeiten von Pferden müssen alle Werte geändert werden können.
- Beim Bearbeiten von Pferden dürfen sich bereits erstellte Simulationen nicht ändern.
- Bei jedem Pferd muss vermerkt sein, wann es zuletzt bearbeitet wurde.
- Die Zeitpunkte der Erstellung sowie der letzten Bearbeitung dürfen von BenutzerInnen nicht verändert werden können.
- Relevante HTTP Anfragen:
  - PUT Horse

Storypoints: 3

ID: 04	**Titel:** Als Pferdebesitzer/in möchte ich Pferde aus dem System löschen können.

- Das Löschen von Pferden darf nicht rückgängig gemacht werden können. Möchte man ein Pferd erneut im System haben, muss es neu angelegt werden.
- Beim Löschen von Pferden dürfen sich bereits erstellte Simulationen nicht ändern.
- Relevante HTTP Anfragen:
  - DELETE Horse

Storypoints: 6

ID: 05	**Titel:** Als Pferdebesitzer/in möchte ich Pferde suchen können.

- Pferde sollen nach folgenden Parametern gesucht werden können:
  - Nach der ID
  - Nach einem Teil des Namens
  - Nach einem Teil der Rasse
  - Nach einer maximalen Minimalgeschwindigkeit
  - Nach einer minimalen Maximalgeschwindigkeit
- Dabei können die Parameter beliebig kombiniert werden.
- Relevante HTTP Anfragen:
  - GET One Horse by Id
  - GET All Horses
  - GET All Horses filtered

Storypoints: 3

2.5.2 Rennbahnbetreiber/in

ID: 06	**Titel:** Als Rennbahnbetreiber/in möchte ich neue Jockeys im System anlegen können.

- Jeder angelegte Jockey muss in einer Datenbank gespeichert werden.
- Zu jedem Jockey müssen dabei folgende Werte gespeichert werden:
  - Name (verpflichtend)
  - Zeitpunkt der Erstellung (automatisch)
- Relevante HTTP Anfragen:
  - POST Jockey

Storypoints: 6

ID: 07	**Titel:** Als Rennbahnbetreiber/in möchte ich das Können eines Jockeys im System erfassen können.

- Das Können ist ein beliebiger Wert zwischen +/-4,9E-324 und +/-1,7E+308 (Double in Java)
- Relevante HTTP Anfragen:
  - POST Jockey

Storypoints: 2

ID: 08	**Titel:** Als Rennbahnbetreiber/in möchte ich Jockeys bearbeiten können.

- Beim Bearbeiten von Jockeys müssen alle Werte geändert werden können.
- Beim Bearbeiten von Jockeys dürfen sich bereits erstellte Simulationen nicht ändern.
- Bei jedem Jockey muss vermerkt sein, wann er zuletzt bearbeitet wurde.
- Die Zeitpunkte der Erstellung sowie der letzten Bearbeitung dürfen von BenutzerInnen nicht verändert werden können.
- Relevante HTTP Anfragen:
  - PUT Jockey

Storypoints: 3

ID: 09	**Titel:** Als Rennbahnbetreiber/in möchte ich Jockeys aus dem System löschen können.

- Das Löschen von Jockeys darf nicht rückgängig gemacht werden können. Möchte man einen Jockey erneut im System haben, muss er neu angelegt werden.
- Beim Löschen von Jockeys dürfen bereits erstellte Simulationen nicht geändert werden.
- Relevante HTTP Anfragen:
  - DELETE Jockey

Storypoints: 6

ID: 10	**Titel:** Als Rennbahnbetreiber/in möchte ich Jockeys suchen können.

- Jockeys sollen nach folgenden Parametern gesucht werden können:
  - Nach der ID
  - Nach einem Teil des Namens
  - Nach dem minimalen Können
- Dabei können die Parameter beliebig kombiniert werden.
- Relevante HTTP Anfragen:
  - GET One Jockey by Id
  - GET All Jockeys
  - GET All Jockeys filtered

Storypoints: 3

ID: 11	**Titel:** Als Rennbahnbetreiber/in möchte ich eine Rennsimulation erstellen können

- Jede Rennsimulation besteht aus dem Zeitpunkt der Erstellung, den Simulationsdaten und einem frei wählbaren Namen.
- Relevante HTTP Anfragen:
  - POST Simulation

Storypoints: 5

ID: 12	**Titel:** Als Rennbahnbetreiber/in möchte ich Rennsimulation suchen können.

- Rennsimulation sollen nach folgendem Parameter gesucht werden können:
  - Nach einem Teil des Namens
- Relevante HTTP Anfragen:
  - GET All Simulations
  - GET All Simulations filtered

Storypoints: 2

ID: 13	**Titel:** Als Rennbahnbetreiber/in möchte ich eine Rennsimulation durchführen.

- Sobald die Simulation gestartet wird, wird für jede am Rennen teilnehmende Pferd-Jockey-Kombination die Durchschnittsgeschwindigkeit berechnet.
- Die Pferd-Jockey-Kombination mit der höchsten Durchschnittsgeschwindigkeit ist somit der Sieger des Rennens.
- Eine einmal durchgeführte Rennsimulation soll in der Datenbank gespeichert werden und lässt sich nach der Durchführung nicht mehr ändern.
- Relevante HTTP Anfragen:
  - POST Simulation

Storypoints: 6

ID: 14	**Titel:** Als Rennbahnbetreiber/in möchte ich das Ergebnis der Rennsimulation abfragen können.

- Das Ergebnis der Rennsimulation soll in Form einer nach Rang sortierten Liste ausgegeben werden.
- Jeder Listeneintrag soll dabei den Namen des Pferdes sowie des Jockeys, die Geschwindigkeit des Pferdes p, den Glücksfaktor g, das Können des Jockeys k' sowie die finale Durchschnittsgeschwindigkeit enthalten.
- Relevante HTTP Anfragen:
  - GET One Simulation by Id

Storypoints: 7

2.5.3 Jockey

ID: 15	**Titel:** Als Jockey möchte ich an einem Rennen teilnehmen können.

- Bei einem Rennen können beliebig viele Pferd-Jockey-Kombinationen teilnehmen.
- Ein Pferd bzw. ein Jockey können nicht mehrmals an einem Rennen teilnehmen.
- Relevante HTTP Anfragen:
  - POST Simulation

Storypoints: 8

ID: 16	**Titel:** Als Jockey möchte ich die Durchschnittsgeschwindigkeit meines Pferdes in einer Rennsimulation berechnen können.

- Die Durchschnittsgeschwindigkeit einer Pferd-Jockey-Kombination berechnet sich aus den Daten des Pferdes (Maximal- und Minimalgeschwindigkeit eines Pferdes), dem Können des Jockeys und einem gewissen Glücksfaktor.
- Die Formel zur Berechnung der Durchschnittsgeschwindigkeit d sieht folgendermaßen aus:
  - p_{min}: p_{min} ist die für das Pferd festgelegt minimale Geschwindigkeit. Wobei gilt: $p_{\text{min}} \geq 40$.
  - p_{max}: p_{max} ist die für das Pferd festgelegt maximale Geschwindigkeit. Wobei gilt: $p_{\text{max}} \leq 60$.
  - $k : k \in \mathbb{R}$ Können des Jockeys zwischen +/-4,9E-324 und +/-1,7E+308.
  - $g : g \in \mathbb{R},\ g \geq 0.95,\ g \leq 1.05$ Glücksfaktor, Zahl zwischen 0.95 und 1.05.
  - $p : p \in \mathbb{R},\ p = (g - 0.95) * (p_{\text{max}} - p_{\text{min}})/(1.05 - 0.95) + p_{\text{min}}$
  - $k' : k' \in \mathbb{R},\ k' = 1 + (0.15 * 1/pi * arctan(1/5 * k))$
  - $d : d \in \mathbb{R},\ d = p * k' * g$
- Die Zwischenergebnisse von p und k', sowie das Endergebnis d, sollen auf 4 Dezimalstellen genau nach kaufmännischen Regeln gerundet werden.
  - Ist die Ziffer an der fünften Dezimalstelle eine 0, 1, 2, 3 oder 4, dann wird abgerundet.
  - Ist die Ziffer an der fünften Dezimalstelle eine 5, 6, 7, 8 oder 9, dann wird aufgerundet.
- Relevante HTTP Anfragen:
  - POST Simulation

Storypoints: 12

2.6 Techstories (80 Storypoints)

Techstories stellen Anforderungen eines Stakeholders an die Qualität der Umsetzung des Produkts dar. Jede Techstory muss dabei in allen Schichten und allen Userstories beachtet werden.

2.6.1 Qualitätsmanager/in

ID: 17	**Titel:** Als Qualitätsmanager/in möchte ich, dass das Programm Logfiles schreibt.

- Das Logfile muss `wendys-rennpferde-<datum>.log` heißen.
- Für jeden Tag muss ein eigenes Logfile erzeugt werden.
- Das Logfile muss alle auftretenden Fehler in Form einer Errormessage enthalten.
- Das Logfile muss alle vom User durchgeführten Aktionen als Infomessage enthalten.
- Das Logfile muss alle fürs Debugging relevanten Informationen als Debugmessage enthalten.
- Das Programm darf ausschließlich über das Logging Framework Ausgaben auf die Standardausgabe (stdout) sowie die Standardfehlerausgabe (stderr) schreiben.

Storypoints: 7

ID: 18	**Titel:** Als Qualitätsmanager/in möchte ich, dass das Programm in Englisch geschrieben und gut dokumentiert ist.

- Der Quellcode (Klassennamen, Variablennamen sowie Methodennamen) muss auf Englisch verfasst sein.
- Für alle Interfaces und Deklarationen in Interfaces muss JavaDoc existieren.
- Die JavaDoc enthält Informationen zu Übergabeparametern.
- Die JavaDoc enthält Informationen zu möglichen Fehlerfällen und Exceptions.
- Die JavaDoc enthält Informationen zu Rückgabewerten.
- Die JavaDoc ist auf Englisch geschrieben.

Storypoints: 7

ID: 19	**Titel:** Als Qualitätsmanager/in möchte ich, dass Kernfunktionalitäten des Programms getestet sind.

- Es müssen mind. vier Tests für die Persistenz Schicht und für die REST Schicht erstellt werden.
- Es müssen vier Tests für die Service Schicht erstellt werden, die die Story 16 testen.
- Es müssen sowohl Positivtests (Normalfall) als auch Negativtests (Fehlerfall) erstellt werden.
- In Summe sollen 12 neue Tests geschrieben werden.

Storypoints: 8

ID: 20	**Titel:** Als Qualitätsmanager/in möchte ich, dass alle Eingaben validiert werden.
• Eingabeparamter (Zahlenwerte oder Textwerte) müssen den Userstories entsprechend validiert werden. • Bei der Validierung muss darauf geachtet werden, ob Parameter verpflichtend oder optional sind. • Es muss zumindest in der Service Schicht validiert werden.	
Storypoints: 7	

ID: 21	**Titel:** Als Qualitätsmanager/in möchte ich, dass alle vom Kunden bereitgestellten Tests durchlaufen.
• Der Kunde hat eine Testsuite an integrativen Tests zusammengestellt, welche die REST-API testen. • Sämtliche Tests dieser Testsuite müssen erfolgreich durchlaufen!	
Storypoints: ∞	

2.6.2 Technische/r Architekt/in

ID: 22	**Titel:** Als Technische/r Architekt/in erwarte ich einen sauberen Git Workflow.
• Es müssen regelmäßig Commits erstellt werden. • Ein Commit gehört immer zu genau einer Userstory und/oder Techstory. • Jeder Commit enthält eine aussagekräftige Commitmessage.	
Storypoints: 7	

ID: 23	**Titel:** Als Technische/r Architekt/in erwarte ich die Verwendung eines Build und Dependency Managements.

- Abhängigkeiten müssen, mittels Maven verwaltet werden.
- Das Backend muss über Maven gebaut, gestartet und getestet werden können.
- Das Backend muss mit dem Befehl:
  `mvnw clean compile`
  erfolgreich gebaut werden können.
- Das Backend muss mit dem Befehl:
  `mvnw clean test`
  erfolgreich getestet werden können.
- Das Backend muss mit dem Befehl:
  `mvnw clean compile spring-boot:run`
  gestartet werden können.
- Für das Backend muss mit dem Befehl:
  `mvnw clean package`
  ein ausführbares jar File erstellt werden können.

Storypoints: 7

ID: 24	**Titel:** Als Technische/r Architekt/in erwarte ich eine saubere Umsetzung der Datenbank.

- Die Datenbank muss im embedded Mode verwendet werden.
- Die Datenbankverbindung darf, während das Programm läuft, nur einmal aufgebaut werden.
- Die Datenbankverbindung muss beim Beenden des Programms sauber geschlossen werden.
- Zur Verwaltung der Datebankverbindung muss das Singleton[2] Pattern eingesetzt werden.
- SQL Statements müssen als Prepared Statement ausgeführt werden.
- Primärschlüssel müssen laufende Nummern sein, die von der Datenbank erstellt werden.
- Um Relationen abbilden zu können müssen sinnvolle Fremdschlüssel gewählt werden.

Storypoints: 8

ID: 25	Titel: Als Technische/r Architekt/in erwarte ich eine gute Umsetzung der Programmarchitektur.

- Das Programm muss eine saubere Schichtentrennung aufweisen.
- Die Austauschbarkeit der Schichten muss gegeben sein.
- Zur Trennung der Schichten muss das Interface[3] Pattern verwendet werden.
- Außerdem dürfen andere Komponenten nie direkt instanziiert werden, sondern müssen mittels Dependency Injection[4] injiziert werden.
- Verwenden von Parameter Objects für die Definition von Suchkriterien und Filtern.
- Verwenden von Exceptions zum Transportieren von Fehlern.

Storypoints: 13

ID: 26	Titel: Als Technische/r Architekt/in erwarte ich vorausschauenden Umgang mit Ressourcen.

- Sofern möglich, müssen Instanzen wiederverwendet werden. Zum Beispiel Instanzen von Data Access Objects und Service Klassen.
- Wiederverwendung von Validatoren für Data Transfer Objects.
- Schließen von I/O-Streams, sofern vorhanden.
- Suchen müssen aus Performancegründen in der Datenbank vorgenommen werden.

Storypoints: 6

ID: 27	Titel: Als Technische/r Architekt/in erwarte ich eine durchdachte Fehlerbehandlung.

- Je nach Art des Fehlers wird eine passende Exception gewählt.
- Exceptions werden ausschließlich zum Transportieren von Fehlern verwendet.
- Fehler müssen im Log aufscheinen.
- Fehler müssen entweder behandelt oder weitgereicht werden.
- Das Programm darf zu keinem Zeitpunkt einfach abstürzen.

Storypoints: 10

[2] https://sourcemaking.com/design_patterns/singleton
[3] https://docs.oracle.com/javase/tutorial/java/concepts/interface.html
[4] https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

2.7 REST-API Spezifikation (siehe Anhang)

Die REST-API beschreibt, wie mittels HTTP Anfragen auf das Backend zugegriffen werden kann. Halten Sie sich beim Implementieren exakt an die API Spezifikation. Für alle Anfragen an das Backend gibt es eine genaue Beschreibung der Eingabeparameter, falls welche benötigt werden. Die darauf folgenden Antworten können Sie aus dem jeweiligen Antwortfeld (HTTP Response) entnehmen. Achten Sie darauf, dass die Felder Ihrer DTOs genauso wie in den HTTP Anfragen bzw. Antworten benannt werden, damit Spring Boot ihre DTOs in JSON Objekte umwandeln kann und umgekehrt. Des Weiteren ist für jede HTTP Antwort ein Status Code definiert. Im Fehlerfall, das heißt Status Codes 4xx und 5xx, muss lediglich der in der API Spezifikation definiert Status Code vom Backend retourniert werden. Der Text, den die HTTP Antwort enthält muss daher nicht ident mit der API Spezifikation sein. Der Text muss den Fehler sinnvoll beschreiben.

3 Implementierung

Der folgende Abschnitt der Angabe stellt einen Leitfaden zur Implementierung dar, er soll Ihnen helfen die richtige Herangehensweise an die Erstellung Ihres Programms zu wählen.

3.1 Erste Schritte

Im ersten Schritt sollten Sie Java OpenJDK 11 herunterladen und installieren. Öffnen Sie das Terminal und führen Sie den Befehl `java -version` aus um zu überprüfen, ob ihr Betriebssystem auch wirklich Java OpenJDK 11 verwendet. Danach sollten Sie Ihre Entwicklungsumgebung einrichten. Nachdem Sie IntelliJ IDEA - Community Edition installiert haben, laden Sie das Template aus TUWEL herunter. Entpacken Sie das Template in einen Ordner ihrer Wahl und öffnen Sie das Projekt in Ihre Entwicklungsumgebung. Öffnen Sie das Projekt indem Sie das pom.xml auswählen.

Das zur Verfügung gestellte Template beinhaltet bereits einen einfachen Durchstich durch alle Schichten, von der Persistenz über die Service bis zur REST Schicht. Außerdem enthält das Template die integrativen Tests des Kunden. Bevor Sie die Applikation starten, sollten Sie die Klasse `java/.../persistence/util/SchemaAndTestDataGenerator.java` ausführen, welche das benötigte Datenbankschema anlegt und initiale Testdaten einspielt. Um die Applikation zu starten, führen Sie den Befehl `mvnw spring-boot:run` im Ordner des Templates aus. Alternativ kann die Klasse `java/.../WendysRennpferdeApplication.java` über Ihre IDE gestartet werden. Die Applikation besteht aus einem Webserver welcher über die URL http://localhost:8080 erreichbar ist. Nach dem Senden einer GET-Anfrage mittels Browser oder Postman[5] an die URL http://localhost:8080/api/v1/horses/1 bekommen Sie das Pferd mit der ID 1 von Ihrem Backend System zurück.
Zusätzlich zu diesem einfachen Durchstich ist in dem Template bereits das Build und Dependency Management Tooling mittels Maven aufgesetzt.

Nachdem Sie das Projekt nun erfolgreich geöffnet haben sollten Sie als erstes die Platzhalter für ihre Matrikelnummer <e01234567> durch Ihre eigene Matrikelnummer ersetzen. Danach können Sie mit der Implementierung beginnen.

3.1.1 Aufbau des Template

Das zur Verfügung gestellte Template folgt einer streng vorgegeben Orderstruktur, diese Sollte von Ihnen nicht geändert werden. Bestimmte Dateien und Ordner sollten ebenfalls weder umbenannt noch verändert werden. Genauere Informationen dazu finden Sie in der nachfolgenden Liste.

Folgende Ordner und Dateien sind im Template enthalten:

`.mvn/` beinhaltet das Build und Dependencymanagement Tool. Dieser Ordner sollte nicht verändert werden.
`src/main/java` beinhaltet den Quellcode Ihrer Anwendung.
`src/main/resources` beinhaltet alle Ressourcen auf die Ihr Programm zugreifen muss, wie z.B. die Spring Konfigurationsdatei application.yml.
`src/test/java` beinhaltet den Quellcode Ihrer Tests.
`target/` wird automatisch vom Build und Dependencymanagement Tool erstellt und sollte nicht verändert werden.
`.editorconfig` enthält eine einfache Konfiguration für die Formatierung von Quellcode.
`.gitignore` enthält eine Liste an Dateien die nicht in git versioniert werden sollen.

[5] https://www.getpostman.com/downloads/

`mvnw` ist nur für Linux NutzerInnen relevant, es führt das Build und Dependencymanagement Tool aus. Die Datei sollte nicht verändert werden.

`mvnw.cmd` ist nur für Windows NutzerInnen relevant, es führt das Build und Dependencymanagement Tool aus. Die Datei sollte nicht verändert werden.

`pom.xml` enthält die Konfiguration für das Build und Dependencymanagement Tool.

`README.md` enthält Informationen über das Projekt.

3.2 Spring

Spring Boot ist ein Java Framework, dass verschiedene Best Practices und andere Frameworks, wie z.B. Apache Commons Logging, JUnit oder Jackson vereint. Des Weitern beinhaltet das Framework den Webserver Tomcat, der die Applikation hostet. Die Datei `resources/application.yml` enthält die Konfiguration des Frameworks. Zusätzlich können auch eigene Werte in dieser Datei angelegt werden um diese an einer Stelle im Programm zentral zu verwalten (Single Source of Truth). Die Klasse `java/.../config/Values.java` liest die Werte aus der Datei `resources/application.yml` aus und stellt diese über Getter-Methoden zur Verfügung.

3.3 Build- und Dependencymanagement

Als Build- und Dependencymanagement Tool werden Sie Apache Maven einsetzen. Das Beispielprojekt enthält bereits eine mitgelieferte Version von Apache Maven, da Maven in Java programmiert ist und in der JVM läuft, müssen Sie nichts weiteres installieren.

Maven erfüllt zwei verschiedene Aufgaben. Zum einen wird es als **Buildtool** verwendet. Die Aufgabe eines Buildtools ist die verschiedenen Schritte des Buildprocesses zu automatisieren und damit reproduzierbar zu machen.
Dazu bietet Maven folgende, für Sie wichtigen, Befehle an.

`mvnw clean` löscht alle beim Kompilieren erstellten Dateien.

`mvnw compile` kompiliert das Programm.

`mvnw test` lässt alle erstellten Testfälle laufen.

`mvnw spring-boot:run` startet das Programm.

`mvnw clean package` erstellt ein ausführbares jar File, wenn alle Tests erfolgreich ausführbar sind.

`mvnw clean package -DskipTests` erstellt ein ausführbares jar File, ohne die Tests auszuführen.

Wichtig: Der Befehl `mvnw` kann nur im Hauptordner des Projekts ausgeführt werden. Unter Linux und Mac OS muss `./mvnw` mit führenden `./` eingegeben werden.

Die zweite wichtige Aufgabe von Maven ist das **Dependencymanagement**. Dependencymanagement oder auch Abhängigkeitsmanagement wird vor allem dann verwendet wenn das erstellte Programm Abhängigkeiten auf Libraries oder Frameworks hat. Im Fall des Einzelbeispiels sind das beispielweise Abhängigkeiten auf Spring Boot oder H2.
Maven wird über die `pom.xml`-Datei konfiguriert. Sie enthält die Abhängigkeiten sowie weitere Informationen über das zu erstellende Projekt. Bei den vorhandenen Projektinformationen ist für Sie vor Allem die `artifactId` relevant. Dieses müssen Sie entsprechend Ihrer Matrikelnummer verändern.

Für die Versionen werden dabei Platzhalter verwendet die unter `properties` definiert sind.

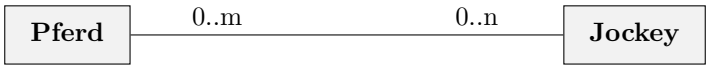
Maven unterscheidet bei den Abhängigkeiten zwischen vier verschiedenen `scopes`.

`compile` sagt aus, dass die Bibliothek bereits beim Kompilieren benötigt wird.

`test` sagt aus, dass die Bibliothek nur zum Testen benötigt wird.
`runtime` sagt aus, dass die Bibliothek nur zur Laufzeit benötigt wird.
`provided` sagt aus das die Bibliothek extern zur Verfügung gestellt wird.

3.4 Vom Domänenmodell zur Datenbank

Ein Domänenmodell stellt Ihre Sicht/Ihr Verständnis des Datenmodells in der realen Welt dar. Es ist daher keine eins zu eins Abbildung des Datenbankdesigns. Um die Domäne etwas zu veranschaulichen, wurde bereits ein Domänenmodell erstellt.



Wie Sie sehen besteht die Domäne aus zwei Entitäten die über eine $n : m$ Beziehung miteinander verbunden sind. Um eine $n : m$ Relation auch in der Datenbank abbilden zu können werden Sie eine weitere Tabelle benötigen. Diese ist ein technisches Implementierungsdetail und daher nicht Teil des Domänenmodells.

Überführen Sie die Entitäten in „CREATE TABLE ...“ Statements Ihrer Datenbank. Sie müssen und sollen nicht das gesamte Domänenmodell sofort in SQL umsetzen. Am besten ist es, wenn Sie Ihre SQL Dateien laufend erweitern und verbessern. Wichtig ist, dass Sie regelmäßig die Einhaltung aller Userstories und Techstories überprüfen.

3.5 Reihenfolge der Implementierung

Bei der Reihenfolge der Implementierung sollten Sie nach Kundenpriorität vorgehen. Es bringt Ihnen beispielsweise nichts, wenn Sie Simulationen auf Basis Ihrer Daten machen können wenn es nicht einmal die Möglichkeit gibt neue Daten in Ihr Programm einzuspielen. Als erstes wählen Sie eine Userstory mit möglichst hoher Kundenpriorität zum Beispiel Userstory 1.
Diese Userstory implementieren Sie dann durch alle Schichten: Persistenz, Service und REST.
Vergessen Sie dabei nicht auf die Einhaltung der Techstories.

Um die verschiedenen Schichten voneinander zu entkoppeln verwenden Sie Data Transfer Objects, Exceptions, Interfaces und Dependency Injections. Auch hier empfiehlt es sich wieder zwischen Interface und Implementierung zu trennen. Außerdem ist es wichtig, dass die REST Schicht nie direkt auf die Persistenzschicht zugreift und umgekehrt.

3.5.1 Persistenz

Die Persistenzschicht regelt den Zugriff auf Ihre Daten, dabei ist es egal ob die Daten im Filesystem oder in einer Datenbank gespeichert sind.
Erstellen Sie als erstes die benötigten DAOs (Data-Access-Objects) für Ihre Entitäten beziehungsweise erweitern Sie die bereits erstellten DAOs um die für die zu implementierende Userstory benötigte Funktionalität.

Die geläufigsten DAO-Operationen sind:

- create
- read/find/search
- update
- delete

Erstellen Sie immer nur die Methoden, die sie sicher brauchen werden.
Informieren Sie sich über den Sinn und Zweck von Data Access Objects und die Umsetzung des

DAO-Patterns. Stellen Sie sicher, dass Sie verstanden haben, warum das DAO-Pattern verwendet wird und warum es eine Trennung zwischen Interface und Implementierung gibt.

- Interfaces [6]
- DAO-Pattern [7]
- Data Transfer Objects [8]

Bevor Sie in Ihrem Programm zum ersten Mal eine Datenbankverbindung aufbauen können müssen Sie die Datenbanktreiber laden, siehe `java/.../persistence/util/DBConnectionManager.java`. Der hier abgebildete Code lädt zuerst die Datenbanktreiber und verbindet sich dann zu einer Datenbank namens „sepm“. Dabei werden der default Benutzername „sa“ und ein leerer Passwortstring verwendet.

Um das Datenbankschema Schrittweise zu erweitern, fügen Sie Ihre „CREATE TABLE ...“ Statements zur Datei `resources/sql/createSchema.sql` hinzu. Um die Datenbank mit Testdaten zu befüllen, fügen Sie Ihre „INSERT INTO ...“ Statements zur Datei `resources/sql/insertData.sql` hinzu. Die Klasse `java/.../persistence/util/SchemaAndTestDataGenerator.java` baut eine Datenbankverbindung auf und führt diese beiden SQL Dateien aus.

Sollte die Datenbank noch nicht existieren wird sie automatisch erstellt. In diesem Beispiel wird das Datenbankfile direkt im Homeverzeichnis des Users erstellt. Dabei werden folgende Dateien angelegt:
`~/sepm.mv.db`
`~/sepm.trace.db`

Wichtig: *Beachten Sie, dass Sie das Verwalten der Datenbankverbindungen entsprechend der Techstories umsetzen müssen.*

Mit dem Befehl:
`./mvnw exec:java -Dexec.mainClass="org.h2.tools.Server"-Dexec.args="-web␣-browser"`
können Sie die H2 WebUI starten. Damit können Sie sich Ihre Datenbank ansehen.

Wichtig: *es kann immer nur entweder ihr Programm oder die WebUI mit Ihrer Datenbank verbunden sein.*

3.5.2 Service

Die Serviceschicht stellt das Herzstück Ihrer Anwendung dar. Sie beinhaltet die komplette Businesslogik. In vielen Fällen kann es durchaus sein, dass die Serviceschicht nicht viel mehr macht als die Daten aus der REST Schicht an die Persistenzschicht durchzureichen. Allerdings ermöglicht sie es die Anwendung einfacher anzupassen und modularer zu gestalten.
In einigen Fällen wird die Serviceschicht auch mehr Aufgaben übernehmen.
Achten Sie bei der Erstellung Ihrer Serviceklassen darauf, dass Ihre Services nur kleine zusammenhängende Aufgaben erfüllen und vermeiden sehr große monolithische Serviceklassen.

3.5.3 REST

Nun fehlt noch die Implementierung der REST Schicht. Diese Schicht ist die Schnittstelle nach außen und ermöglicht es externen Services oder GUIs mit Ihrem Backend System zu kom-

[6] https://docs.oracle.com/javase/tutorial/java/concepts/interface.html
[7] https://www.oracle.com/technetwork/java/dataaccessobject-138824.html
[8] https://www.oracle.com/technetwork/java/transferobject-139757.html

munizieren. Halten sie sich beim Implementieren dieser Schicht ganz genau an die REST-API Spezifikation die in 2.7 definiert wurde.

3.6 Testen

Die erstellte Software zu testen gehört zu den wichtigsten Aktivitäten des Entwicklungsprozesses. Ziel ist es, Fehler so früh wie möglich zu finden, jeder Entwickler ist dabei auch Tester.
Um sicherzustellen, dass Fehler möglichst früh gefunden werden, wird der Testprozess direkt mit dem Entwicklungsprozess verwoben (zum Beispiel mittels Test Driven Development, hierbei ist die Erstellung der Tests sogar eine Vorbedingnung zum Erstellen der Implementierung).

Im Rahmen des Einzelbeispiels werden Sie ein Framework zur Testautomatisierung verwenden um Unittests zu stellen. Außerdem empfiehlt es sich im Laufe des Entwicklungsprozesses ins besondere die Kernfunktionalitäten Ihrer Anwendung immer wieder manuell zu testen.

3.6.1 Normalfall und Fehlerfall

Beim Testen wird zwischen Tests für den Normalfall und Tests für den Fehlerfall unterschieden. Ein Normalfall (NF) stellt einen Test einer gültigen Eingabe in das System dar. Hierbei wird geprüft ob sich das Programm bei einer richtigen Eingabe korrekt verhält. Ein Fehlerfall (FF) stellt einen Test einer ungültigen Eingabe in das System dar. Hierbei wird geprüft ob sich das Programm bei einer falschen Eingabe korrekt verhält. Ein typisches Beispiel eines Fehlerfalls wäre wenn der/die Nutzer/in ein Pferd mittels ID laden möchte und diese ID im System nicht existiert. In dem Fall muss eine entsprechende Exception ausgelöst werden mit deren Hilfe der Fehler behandelt beziehungsweise an den/die Nutzer/in kommuniziert werden kann.

Wichtig: *Vergessen Sie nicht, dass Sie sowohl für den Normalfall als auch den Fehlerfall Tests erstellen müssen.*

3.6.2 Manuelle Tests

Im Rahmen des Einzelbeispiels werden Sie auch manuelle Tests durchführen. Hierbei ist es wichtig, dass sie strukturiert vorgehen.
Gehen Sie dazu die bereits implementierten Userstories Punkt für Punkt durch und prüfen Sie die Einhaltung aller Punkte der Userstory sowie der Techstories. Am Einfachsten fällt das Testen wenn Sie sich dabei einen roten Faden zurecht legen. Ein Beispiel, in stark vereinfachter Form, wäre:

1. Pferd anlegen <Joe, Haflinger,40.1, 60.0>
2. Jockey anlegen <John, 76.5>
3. Simulation start <Joe, John>
4. Simulation anzeigen <Simulation1>
5. Pferd bearbeiten <Joe, Haflinger, 45., 55.0>
6. Simulation anzeigen <Simulation1>

3.6.3 Automatisierte Tests

Im Rahmen der Umsetzung bestimmter Stories werden Sie automatisierte Tests erstellen. Dafür verwenden Sie in der Einzelphase das Framework JUnit. Die mittels JUnit erstellten Tests sollten Sie regelmäßig über den Befehl `mvnw clean test` ausführen.

Wenn Sie im Rahmen der Softwareentwicklung Tests erstellen, machen Sie das häufig nach dem Prinzip des „Test-Driven-Development“ (TDD). Dabei erstellen Sie zuerst die Interfaces für die zu testenden Klassen. Danach erstellen Sie Tests sowie leere Implementierungen für die

Interfaces. Diese Tests sollen natürlich fehlschlagen, da es noch keine entsprechend vollständige Implementierung gibt. Im nächsten Schritt implementieren Sie das Interface, danach sollten die Tests fehlerfrei ausgeführt werden können.
Diese Vorgehensweise garantiert, dass jeder Programmcode einen Test besitzt und Sie nur den minimal notwendigen Programmcode erstellen, um der Spezifikation zu genügen. Hier zeigt sich wie wichtig es ist, vollständige und korrekte Dokumentation im Code zu haben. Nur aus einer vollständigen Interface-Dokumentation lassen sich ausreichend viele und gute Tests erstellen.

Wichtig: *Beachten Sie, dass Sie auf Grund der limitierten Zeit für die Einzelphase nur eine geringere Anzahl an Tests erstellen müssen und nicht jede Klasse getestet sein muss.*

JUnit führt beim Ausführen der Tests alle mit `@Test` annotierten Methoden aus. Die Reihenfolge der Testausführung ist nicht definiert, daher müssen Sie darauf achten, dass die verschiedenen Testfälle nicht voneinander abhängig sind.
Außerdem sollten Tests so simpel und selbsterklärend wie möglich geschrieben sein. Vermeiden Sie in Tests die Verwendung von Schleifen sowie kompliziertes Exceptionhandling. Achten Sie auch darauf, dass Ihre Testklassen und Methoden aussagekräftige Namen haben.

3.7 Versionskontrolle

Im Rahmen der Einzelphase verwenden Sie das Versionskontrollsystem Git um den Quelltext, Ihre Testdaten, Ihre Dokumente sowie Ihre Programmdateien in einem Repository zu verwalten. Git ist ein verteiltes Versionskontrollsystem, das heißt, dass Sie eine lokale Kopie des Repositories auf Ihrem Entwicklungssystem haben. In diesem machen Sie Ihre Commits, die Sie danach auf das zur Verfügung gestellte Repository pushen.

Das Versionskontrollsystem ist ein sehr mächtiges Werkzeug, insbesondere wenn Sie im Team arbeiten. In der Einzelphase benötigen Sie nur ein minimales Set an Befehlen:

```
1 git init
2 git status
3 git clone
```

```
4 git add
5 git commit
6 git push
7 git pull
```

Wichtig: *Beachten Sie, dass aus organisatorischen Gründen die Zugänge zum Versionskontrollsystem ein paar Tage verzögert zur Verfügung gestellt werden. Sie können und sollen schon davor mit Ihrer Implementierung beginnen.*

3.7.1 Initialisieren

Um Git in einem Projekt nutzen zu können müssen Sie das Projekt zuerst für Git initialisieren. Dazu wechseln Sie in den Ordner den Sie unter Versionskontrolle stellen wollen und geben folgenden Befehl ein:

```
1 $ cd sepm-individual-assignment/
2 $ git init
3 Initialized empty Git repository in /projects/sepm-individual-assignment/.git/
4 $ git status
5 On branch master
6
7 No commits yet
8
9 nothing to commit (create/copy files and use "git add" to track)
```

Mit dem Befehl `git status` können Sie sich den Status ihres Repositories anzeigen lassen.

Alternativ zur Initialisierung eines lokalen Ordners können Sie auch ein bereits initialisierten Ordner klonen. Dazu verwenden Sie den Befehl `git clone <url>`. Weitere Informationen finden Sie nach der Freischaltung des Repositories direkt in RESET.

3.7.2 Add & Commit

Um Änderungen permanent zu Ihrem lokalen Repository hinzuzufügen benötigen Sie zwei Befehle. Als erstes verwenden Sie den Befehl `git add .` um alle Änderungen an Ihrem Arbeitsverzeichnis zu markieren (Sie können mittels `git add <filename>` auch Änderungen an einzelnen Dateien markieren. Nach dem „added“ befinden sich die Änderungen im „staging“ Bereich. Um diese Änderungen permanent zu Ihrem Repository hinzuzufügen müssen Sie die Änderungen mit dem Befehl `git commit -m <message>` commiten.

Abbildung 1: Git Commit: https://xkcd.com/1296/

Beim Commiten ist es wichtig eine sinnvolle Commitmessage zu wählen die relevante Informationen für die durchgeführte Änderung enthält.
Versuchen Sie Ihre Commitmessages auf Englisch zu halten und achten Sie dabei darauf, die Nachricht kurz und prägnant zu formulieren.
Jedes Set an Änderungen soll zudem genau einer Userstory und/oder Techstory zugeordnet sein. So erreichen Sie, dass die Commits übersichtlich bleiben und nur zusammenhängende Inhalte haben.

Beispiele für schlechte Commitmessages:

- „Neue Implementierung hinzugefügt“
- „Added changes to Horse, Jockey and Race Simulations“
- „Fix Bug“

Beispiele für gute Commitmessages:

- „Fixed bug where name is not stored correctly [story: 1]“
- „Added validation to horse [1, 20]“

3.7.3 Push & Pull

Nachdem Sie Änderungen zu Ihrem lokalen Repository hinzugefügt haben müssen Sie diese auf den Server pushen, das machen Sie mit dem Befehl `git push`. Damit Ihre Abgabe gewertet wird muss diese rechtzeitig auf den Server gepushed werden. Am besten Sie pushen Ihre Änderungen

so häufig wie möglich. So stellen Sie sicher das zur Deadline alle relevanten Daten auf unseren Servern vorhanden sind und beugen Datenverlust vor.
Git macht es Ihnen möglich von mehreren Rechnern aus am gleichen Code zu arbeiten. Dazu bietet Git, mittels dem Befehl `git pull`, die Möglichkeit alle Änderungen vom Server zu laden und so Ihr lokales Repository auf den aktuellen Stand zu bringen.

3.8 Weiterführende Links & Literatur

- H2 1.4.x
  - https://www.h2database.com/html/main.html
- Apache Commons Logging 1.2.x
  - https://commons.apache.org/proper/commons-logging/
- JUnit 4.x.x
  - http://junit.org/junit4/
- Maven 3.5.x
  - https://maven.apache.org/
- Git 2.x.x
  - https://git-scm.com/
  - Deutscher Git Guide: https://rogerdudler.github.io/git-guide/
  - Git Cheatsheet: https://ndpsoftware.com/git-cheatsheet.html
- Design Patterns
  - https://sourcemaking.com/
  - http://java-design-patterns.com/
  - Singleton: https://sourcemaking.com/design_patterns/singleton
  - Interface: https://docs.oracle.com/javase/tutorial/java/concepts/interface.html
  - DAO: https://www.oracle.com/technetwork/java/dataaccessobject-138824.html
  - DTO: https://www.oracle.com/technetwork/java/transferobject-139757.html
- IntelliJ IDEA - Community Edition 2018.x.x
  - https://www.jetbrains.com/idea/download/
- Spring Boot
  - Spring Boot Documentation Project: https://spring.io/projects/spring-boot
- Maven
  - Maven Repository Search: https://mvnrepository.com/
  - Maven Central Search: https://search.maven.org/
- Vertiefendes Testen
  - Mocking: http://en.wikipedia.org/wiki/Mock_object
  - Google Testing Blog: http://googletesting.blogspot.co.at/
- Clean Code
  - 2009; Clean Code: A Handbook of Agile Software Craftsmanship
- Vorträge und Vorlesungen an der TU
  - 183.239/188.410; Software Engineering und Projektmanagement; VO
  - 180.764; Software-Qualitätssicherung; VU

4 Bewertung

Sie erhalten Punkte für die Implementierung jeder einzelner Userstory. Die Punkte, die Sie für jede Userstory erhalten können, entsprechen den definierten Storypoints für jede der Userstories. Die Gesamtpunkte für die Userstories summieren sich auf 80 Punkte. Die Userstories werden einzeln Punkt für Punkt abgenommen. Mangelhafte Userstories werden mit 0 Punkten beurteilt. Die Einhaltung der Techstories wird Punkt für Punkt abgenommen. Für die Nichterfüllung von Techstories erhalten Sie Punkteabzüge. Die maximalen Punkteabzüge einer Techstory entsprechen den definierten Storypoints für die jeweilige Techstory.
Die einzelnen Teilbereiche einer Userstory sind bezüglich der zu erreichenden beziehungsweise abzuziehenden Punkte einer Story nicht gleichverteilt!

ACHTUNG: Für einen positiven Abschluss der Einzelphase muss Story 21 zwingend erfüllt sein!

4.1 Bestehen der Einzelphase

Um an der Gruppenphase teilnehmen zu können, benötigen Sie in Summe aus dem Einstiegstest (bis zu 10 Punkte), Live Beispiel (bis zu 30 Punkte) und Einzelbeispiel (bis zu 80 Punkte) mindestens 60 Punkte. Außerdem benötigen Sie mindestens 40 Punkte auf das Einzelbeispiel.

4.1.1 Einstiegstest (max. 10 Punkte)

Für das Lösen des Einstiegstests haben Sie einen Versuch und 60 Minuten Zeit.
Der Einstiegstest wird automatisiert bewertet und sie müssen keine Mindestpunkte erreichen.

4.1.2 Live Beispiel (max. 30 Punkte)

Für das Lösen des Live Beispiels haben Sie 50 Minuten Zeit, während der Sie ein bestehendes Programm um Funktionalität erweitern müssen. Dabei sollen Sie zeigen, dass Sie die Technologien der Einzelphase beherrschen.
Das Live Beispiel wird automatisiert bewertet und Sie müssen keine Mindestpunkte erreichen.

4.1.3 Einzelbeispiel (max. 80 Punkte)

Sie müssen mindestens 40 Punkte erreichen.

4.2 Einfluss auf die Endnote

Für die Gruppenphase erhalten Sie vom Assistenten eine Note. Diese Note bestimmt ¾ Ihrer Endnote, ¼ Ihrer Endnote macht die Einzelphase aus. Der Notenschlüssel für die Einzelphase entspricht der Standardnotenverteilung.

Prozent	Punkte	Note
100,00 % - 88,00 %	120 - 105	S1
87,99 % - 75,00 %	104 - 90	U2
74,99 % - 63,00 %	89 - 75	B3
62,99 % - 50,00 %	74 - 60	G4

Zur Veranschaulichung noch zwei Beispiele: Wenn Sie in der Einzelphase ein B3 erreichen und in der Gruppenphase ein S1 ergibt das die Notensumme von $1,5$ und Sie bekommen als Gesamtnote ein S1. Wenn Sie in der Einzelphase eine U2 und in der Gruppenphase ein B3 erreichen ergibt das die Notensumme von $2,75$ und Sie bekommen als Gesamtnote ein B3. Gerundet wird dabei immer auf den nächsten Integer (> 0.5 wird aufgerundet und ≤ 0.5 wird abgerundet).

Wichtig: *Für eine positive Gesamtnote müssen Sie in der Einzelphase und in der Gruppenphase positiv sein.*

5 Abgabegespräch

5.1 Vorbedingungen

- Sie sind in RESET zu einem **Abgabegespräch angemeldet**.
- Ihr Projekt ist vollständig (Dokumente, Programmdateien, Testdaten und Quelltext; kein Sourcecode oder Binärdaten von Libraries die Sie über Dependencymanagement beziehen) im SCM vorhanden. **Nur Code und Dokumentation, die im SCM liegen, werden für die Bewertung herangezogen.**
- Die Abgaben finden ausschließlich auf den Laborrechnern statt!

5.2 Ablauf des Abgabegesprächs

1. Live-Beispiel (Programmieraufgabe)
   - **Selbständiges Lösen einer Programmieraufgabe** (max. 50min)
   - **Automatisierte Bewertung**
2. Abgabe des Einzelbeispiels
   - **Produktcheck**: Der/die Tutor/Tutorin lässt sich alle **Userstories** von Ihnen erklären und vorführen, außerdem prüft er/sie die Einhaltung der **Techstories** und führt selbständig Tests durch.
   - **Verständnisfragen**: Der/die Tutor/Tutorin überprüft Ihr Verständnis zum Produkt, zu den Technologien sowie zur Entwicklungsumgebung.

5.3 Wichtige Hinweise zur Abgabe

- **Plagiate werden ausnahmslos negativ beurteilt!**
- Änderungen **nach der Deadline** werden **nicht akzeptiert!**
- Das Programm muss lauffähig sein!
- Achten Sie auf die **Vollständigkeit** der Funktionalität Ihres Programms.
- Während der Abgabe müssen Sie den **Sourcecode** Ihres Programms an beliebigen Stellen jederzeit und ohne Vorbereitungszeit flüssig **erklären** können.
- Die **Datenbank** muss mit einer entsprechenden Anzahl an realistischen **Testdatensätzen** befüllt sein (**zumindest 10 Stück** pro Domänenobjekt, achten Sie ebenfalls darauf, dass Relationen in den Testdaten vorhanden sind).

5.4 Nach dem Abgabegespräch

Nach dem Abgabegespräch werden Sie einem Institut zugeordnet, dabei versuchen wir nach Möglichkeit, Ihren Institutswunsch zu berücksichtigen. Je nach Institut haben Sie nach der Zuordnung, **möglichst vor dem ersten Tutorenmeeting**, Folgendes zu tun:

- **QSE**: Erarbeiten Sie einen eigenen Projektvorschlag, den Sie präsentieren müssen.
- **INSO**: Machen Sie sich mit der Umsetzung des Beispielprojekts vertraut.

Wichtig: *Melden Sie sich auch* ***rechtzeitig für ein Abgabegespräch*** *an. Nutzen Sie die Hilfestellungen durch das* ***TUWEL Forum*** *und unsere* ***Tutorinnen und Tutoren****.*

Viel Spaß und Erfolg beim Einzelbeispiel aus der SE&PM Laborübung!

Deadline: Sonntag, 24. März 2019, 23:55h

[Anhang] REST API Specification

POST Horse

HTTP request

```
POST /api/v1/horses HTTP/1.1
Accept: application/json
Content-Type: application/json; charset=UTF-8
Content-Length: 106
Host: localhost:8080

{"id":null,"name":"Horse1","breed":"Breed1","minSpeed":45.0,"maxSpeed":55.0,"created":null,"updated":null}
```

HTTP response

```
HTTP/1.1 201 Created
Date: Mon, 25 Feb 2019 12:36:36 GMT
Content-Type: application/json;charset=UTF-8
Content-Length: 145
Transfer-Encoding: chunked

{"id":1,"name":"Horse1","breed":"Breed1","minSpeed":45.0,"maxSpeed":55.0,"created":"2019-02-
25T13:36:36.357","updated":"2019-02-25T13:36:36.357"}
```

Request fields

Path	Type	Description
`name`	`String`	The horses name
`breed`	`String`	The breed of the horse, which is optional
`minSpeed`	`Number`	The minimum speed of the horse, which must be at least 40.0 km/h and smaller than the max_speed
`maxSpeed`	`Number`	The maximum speed of the horse, which must be at most 60.0 km/h and greater than min_speed

Invalid Request

HTTP request

```
POST /api/v1/horses HTTP/1.1
Content-Length: 96
Accept: application/json
Content-Type: application/json; charset=UTF-8
Host: localhost:8080

{"id":null,"name":"","breed":null,"minSpeed":null,"maxSpeed":null,"created":null,"updated":null}
```

HTTP response

```
HTTP/1.1 400 Bad Request
Date: Mon, 25 Feb 2019 12:36:36 GMT
```

```
Content-Length: 159
Connection: close
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked

{"timestamp":"2019-02-25T12:36:36.710+0000","status":400,"error":"Bad Request","message":"Error during
saving horse: Name must be set","path":"/api/v1/horses"}
```

GET One Horse by Id

HTTP request

```
GET /api/v1/horses/1 HTTP/1.1
Accept: application/json
Host: localhost:8080
```

HTTP response

```
HTTP/1.1 200 OK
Date: Mon, 25 Feb 2019 12:36:36 GMT
Content-Type: application/json;charset=UTF-8
Content-Length: 145
Transfer-Encoding: chunked

{"id":1,"name":"Horse1","breed":"Breed1","minSpeed":45.0,"maxSpeed":55.0,"created":"2019-02-
25T13:36:36.357","updated":"2019-02-25T13:36:36.357"}
```

Path parameters

Table 1. /api/v1/horses/{id}

Parameter	Description
`id`	The horses id

Invalid Request

HTTP request

```
GET /api/v1/horses/10 HTTP/1.1
Accept: application/json
Host: localhost:8080
```

HTTP response

```
HTTP/1.1 404 Not Found
Date: Mon, 25 Feb 2019 12:36:36 GMT
Content-Length: 176
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked

{"timestamp":"2019-02-25T12:36:36.909+0000","status":404,"error":"Not Found","message":"Error during
reading horse: Could not find horse with id 10","path":"/api/v1/horses/10"}
```

Path parameters

Table 1.

/api/v1/horses/{id}

Parameter	Description
id	The horses id

GET All Horses

HTTP request

```
GET /api/v1/horses HTTP/1.1
Accept: application/json
Host: localhost:8080
```

HTTP response

```
HTTP/1.1 200 OK
Date: Mon, 25 Feb 2019 12:36:36 GMT
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Content-Length: 147

[{"id":1,"name":"Horse1","breed":"Breed1","minSpeed":45.0,"maxSpeed":55.0,"created":"2019-02-
25T13:36:36.357","updated":"2019-02-25T13:36:36.357"}]
```

GET All Horses filtered

HTTP request

```
GET /api/v1/horses?name=Horse&breed=Breed&minSpeed=40.0&maxSpeed=60.0 HTTP/1.1
Accept: application/json
Host: localhost:8080
```

HTTP response

```
HTTP/1.1 200 OK
Date: Mon, 25 Feb 2019 12:36:36 GMT
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Content-Length: 157

[{"id":1,"name":"HorseUpdate","breed":"BreedUpdate","minSpeed":41.0,"maxSpeed":59.0,"created":"2019-02-
25T13:36:36.357","updated":"2019-02-25T13:36:36.524"}]
```

Request parameters

Parameter	Description
name	Filters all horses where the name contains this string
breed	Filters all horses the breed contains this string

Parameter	Description
minSpeed	Filters all horses with greater or equal minSpeed
maxSpeed	Filters all horses with smaller or equal maxSpeed

PUT Horse

HTTP request

```
PUT /api/v1/horses/1 HTTP/1.1
Accept: application/json
Content-Type: application/json; charset=UTF-8
Content-Length: 116
Host: localhost:8080

{"id":null,"name":"HorseUpdate","breed":"BreedUpdate","minSpeed":41.0,"maxSpeed":59.0,"created":null,"updat
ed":null}
```

HTTP response

```
HTTP/1.1 200 OK
Date: Mon, 25 Feb 2019 12:36:36 GMT
Content-Type: application/json;charset=UTF-8
Content-Length: 134
Transfer-Encoding: chunked

{"id":1,"name":"HorseUpdate","breed":"BreedUpdate","minSpeed":41.0,"maxSpeed":59.0,"created":"2019-02-
25T13:36:36.357","updated":"2019-02-25T13:36:36.524"}
```

Request fields

Path	Type	Description
name	String	The horses name
breed	String	The breed of the horse, which is optional
minSpeed	Number	The minimum speed of the horse, which must be at least 40.0 km/h and smaller than the max_speed
maxSpeed	Number	The maximum speed of the horse, which must be at most 60.0 km/h and greater than min_speed

Only request fields, which are not null are handled by the backend system. Null values are allowed for all fields.

Path parameters

Table 1. /api/v1/horses/{id}

Parameter	Description
id	The horses id

Invalid Request

HTTP request

```
PUT /api/v1/horses/1 HTTP/1.1
Content-Length: 96
Accept: application/json
Content-Type: application/json; charset=UTF-8
Host: localhost:8080

{"id":null,"name":"","breed":null,"minSpeed":null,"maxSpeed":null,"created":null,"updated":null}
```

HTTP response

```
HTTP/1.1 400 Bad Request
Content-Length: 163
Date: Mon, 25 Feb 2019 12:36:36 GMT
Connection: close
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked

{"timestamp":"2019-02-25T12:36:36.792+0000","status":400,"error":"Bad Request","message":"Error during
updating horse: Name must be set","path":"/api/v1/horses/1"}
```

DELETE Horse

HTTP request

```
DELETE /api/v1/horses/2 HTTP/1.1
Accept: application/json
Host: localhost:8080
```

HTTP response

```
HTTP/1.1 200 OK
Date: Mon, 25 Feb 2019 12:36:36 GMT
```

Path parameters

Table 1. /api/v1/horses/{id}

Parameter	Description
id	The horses id

Invalid Request

HTTP request

```
DELETE /api/v1/horses/10 HTTP/1.1
Accept: application/json
Host: localhost:8080
```

HTTP response

```
HTTP/1.1 404 Not Found
Date: Mon, 25 Feb 2019 12:36:36 GMT
Content-Type: application/json;charset=UTF-8
Content-Length: 177
Transfer-Encoding: chunked

{"timestamp":"2019-02-25T12:36:36.964+0000","status":404,"error":"Not Found","message":"Error during
deleting horse: Could not find horse with id 10","path":"/api/v1/horses/10"}
```

Path parameters

Table 1. /api/v1/horses/{id}

Parameter	Description
id	The horses id

POST Jockey

HTTP request

```
POST /api/v1/jockeys HTTP/1.1
Accept: application/json
Content-Type: application/json; charset=UTF-8
Content-Length: 73
Host: localhost:8080

{"id":null,"name":"Jockey1","skill":7755.0,"created":null,"updated":null}
```

HTTP response

```
HTTP/1.1 201 Created
Date: Mon, 25 Feb 2019 12:36:35 GMT
Content-Type: application/json;charset=UTF-8
Content-Length: 112
Transfer-Encoding: chunked

{"id":1,"name":"Jockey1","skill":7755.0,"created":"2019-02-25T13:36:34.945","updated":"2019-02-
25T13:36:34.945"}
```

Request fields

Path	Type	Description
name	String	The jockeys name

Path	Type	Description
`skill`	`Number`	The skill of the jockey, which must be at between +/-4,9E-324 und +/-1,7E+308 (Java Double)

Invalid Request

HTTP request

```
POST /api/v1/jockeys HTTP/1.1
Accept: application/json
Content-Length: 64
Content-Type: application/json; charset=UTF-8
Host: localhost:8080

{"id":null,"name":"","skill":null,"created":null,"updated":null}
```

HTTP response

```
HTTP/1.1 400 Bad Request
Date: Mon, 25 Feb 2019 12:36:36 GMT
Content-Length: 161
Connection: close
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked

{"timestamp":"2019-02-25T12:36:36.275+0000","status":400,"error":"Bad Request","message":"Error during
saving jockey: Name must be set","path":"/api/v1/jockeys"}
```

GET One Jockey by Id

HTTP request

```
GET /api/v1/jockeys/1 HTTP/1.1
Accept: application/json
Host: localhost:8080
```

HTTP response

```
HTTP/1.1 200 OK
Date: Mon, 25 Feb 2019 12:36:35 GMT
Content-Type: application/json;charset=UTF-8
Content-Length: 112
Transfer-Encoding: chunked

{"id":1,"name":"Jockey1","skill":7755.0,"created":"2019-02-25T13:36:34.945","updated":"2019-02-
25T13:36:34.945"}
```

Path parameters

Table 1. /api/v1/jockeys/{id}

Parameter	Description

Parameter	Description
id	The jockeys id

Invalid Request

HTTP request

```
GET /api/v1/jockeys/10 HTTP/1.1
Accept: application/json
Host: localhost:8080
```

HTTP response

```
HTTP/1.1 404 Not Found
Date: Mon, 25 Feb 2019 12:36:35 GMT
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Content-Length: 179

{"timestamp":"2019-02-25T12:36:35.979+0000","status":404,"error":"Not Found","message":"Error during
reading jockey: Could not find jockey with id 10","path":"/api/v1/jockeys/10"}
```

Path parameters

Table 1. /api/v1/jockeys/{id}

Parameter	Description
id	The jockeys id

GET All Jockeys

HTTP request

```
GET /api/v1/jockeys HTTP/1.1
Accept: application/json
Host: localhost:8080
```

HTTP response

```
HTTP/1.1 200 OK
Date: Mon, 25 Feb 2019 12:36:35 GMT
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Content-Length: 114

[{"id":1,"name":"Jockey1","skill":7755.0,"created":"2019-02-25T13:36:34.945","updated":"2019-02-
25T13:36:34.945"}]
```

GET All Jockeys filtered

HTTP request

```
GET /api/v1/jockeys?name=Jockey&skill=3000.0 HTTP/1.1
Accept: application/json
Host: localhost:8080
```

HTTP response

```
HTTP/1.1 200 OK
Date: Mon, 25 Feb 2019 12:36:36 GMT
Content-Length: 119
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked

[{"id":1,"name":"JockeyUpdate","skill":3441.0,"created":"2019-02-25T13:36:34.945","updated":"2019-02-
25T13:36:35.651"}]
```

Request parameters

Parameter	Description
`name`	Filters all jockeys where the name contains this string
`skill`	Filters all jockeys with greater or equal skill

PUT Jockey

HTTP request

```
PUT /api/v1/jockeys/1 HTTP/1.1
Accept: application/json
Content-Type: application/json; charset=UTF-8
Content-Length: 78
Host: localhost:8080

{"id":null,"name":"JockeyUpdate","skill":3441.0,"created":null,"updated":null}
```

HTTP response

```
HTTP/1.1 200 OK
Content-Length: 96
Date: Mon, 25 Feb 2019 12:36:35 GMT
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked

{"id":1,"name":"JockeyUpdate","skill":3441.0,"created":"2019-02-25T13:36:34.945","updated":"2019-02-
25T13:36:35.651"}
```

Request fields

Path	Type	Description
`name`	`String`	The jockeys name
`skill`	`Number`	The skill of the jockey, which must be at between +/-4,9E-324 und +/-1,7E+308 (Java Double)

Only request fields, which are not null are handled by the backend system. Null values are allowed for all fields.

Path parameters

Table 1. /api/v1/jockeys/{id}

Parameter	Description
id	The jockeys id

Invalid Request

HTTP request

```
PUT /api/v1/jockeys/1 HTTP/1.1
Accept: application/json
Content-Length: 64
Content-Type: application/json; charset=UTF-8
Host: localhost:8080

{"id":null,"name":"","skill":null,"created":null,"updated":null}
```

HTTP response

```
HTTP/1.1 400 Bad Request
Content-Length: 165
Date: Mon, 25 Feb 2019 12:36:35 GMT
Connection: close
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked

{"timestamp":"2019-02-25T12:36:35.917+0000","status":400,"error":"Bad Request","message":"Error during
updating jockey: Name must be set","path":"/api/v1/jockeys/1"}
```

DELETE Jockey

HTTP request

```
DELETE /api/v1/jockeys/2 HTTP/1.1
Accept: application/json
Host: localhost:8080
```

HTTP response

```
HTTP/1.1 200 OK
Date: Mon, 25 Feb 2019 12:36:35 GMT
```

Path parameters

Table 1. /api/v1/jockeys/{id}

Parameter	Description

Parameter	Description
id	The jockeys id

Invalid Request

HTTP request

```
DELETE /api/v1/jockeys/10 HTTP/1.1
Accept: application/json
Host: localhost:8080
```

HTTP response

```
HTTP/1.1 404 Not Found
Content-Length: 180
Date: Mon, 25 Feb 2019 12:36:35 GMT
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked

{"timestamp":"2019-02-25T12:36:36.047+0000","status":404,"error":"Not Found","message":"Error during
deleting jockey: Could not find jockey with id 10","path":"/api/v1/jockeys/10"}
```

Path parameters

Table 1. /api/v1/jockeys/{id}

Parameter	Description
id	The jockeys id

POST Simulation

HTTP request

```
POST /api/v1/simulations HTTP/1.1
Accept: application/json
Content-Type: application/json; charset=UTF-8
Content-Length: 183
Host: localhost:8080

{"name":"Simulation1","simulationParticipants":[{"horseId":4,"jockeyId":4,"luckFactor":1.0},
{"horseId":5,"jockeyId":5,"luckFactor":0.95},{"horseId":6,"jockeyId":6,"luckFactor":1.05}]}
```

HTTP response

```
HTTP/1.1 201 Created
Date: Mon, 25 Feb 2019 12:36:37 GMT
Content-Length: 483
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked

{"id":1,"name":"Simulation1","created":"2019-02-25T13:36:37.203","horseJockeyCombinations":
[{"id":1,"rank":1,"horseName":"Horse3","jockeyName":"Jockey3","avgSpeed":57.75,"horseSpeed":55.0,"skill":1.
0,"luckFactor":1.05},
{"id":2,"rank":2,"horseName":"Horse1","jockeyName":"Jockey1","avgSpeed":53.455,"horseSpeed":50.0,"skill":1.
0691,"luckFactor":1.0},
```

{"id":3,"rank":3,"horseName":"Horse2","jockeyName":"Jockey2","avgSpeed":43.9382,"horseSpeed":49.99,"skill":0.9252,"luckFactor":0.95}]}

Request fields

Path	Type	Description
name	String	The simulations name
simulationParticipants.[]	Array	An array of participants
simulationParticipants.[].horseId	Number	Id of the participating horse
simulationParticipants.[].jockeyId	Number	Id of the participating jockey
simulationParticipants.[].luckFactor	Number	Factor, which specifies the luck of the participant

Invalid Request

HTTP request

```
POST /api/v1/simulations HTTP/1.1
Content-Length: 41
Accept: application/json
Content-Type: application/json; charset=UTF-8
Host: localhost:8080

{"name":"","simulationParticipants":null}
```

HTTP response

```
HTTP/1.1 400 Bad Request
Content-Length: 173
Date: Mon, 25 Feb 2019 12:36:37 GMT
Connection: close
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked

{"timestamp":"2019-02-25T12:36:37.388+0000","status":400,"error":"Bad Request","message":"Error during performing simulation: Name must be set","path":"/api/v1/simulations"}
```

GET One Simulation by Id

HTTP request

```
GET /api/v1/simulations/1 HTTP/1.1
Accept: application/json
Host: localhost:8080
```

HTTP response

```
HTTP/1.1 200 OK
Date: Mon, 25 Feb 2019 12:36:37 GMT
Content-Length: 483
Content-Type: application/json;charset=UTF-8
```

```
Transfer-Encoding: chunked

{"id":1,"name":"Simulation1","created":"2019-02-25T13:36:37.203","horseJockeyCombinations":
[{"id":1,"rank":1,"horseName":"Horse3","jockeyName":"Jockey3","avgSpeed":57.75,"horseSpeed":55.0,"skill":1.
0,"luckFactor":1.05},
{"id":2,"rank":2,"horseName":"Horse1","jockeyName":"Jockey1","avgSpeed":53.455,"horseSpeed":50.0,"skill":1.
0691,"luckFactor":1.0},
{"id":3,"rank":3,"horseName":"Horse2","jockeyName":"Jockey2","avgSpeed":43.9382,"horseSpeed":49.99,"skill":
0.9252,"luckFactor":0.95}]}
```

Path parameters

Table 1. /api/v1/simulations/{id}

Parameter	Description
`id`	The simulation's id

Invalid Request

HTTP request

```
GET /api/v1/simulations/-1 HTTP/1.1
Accept: application/json
Host: localhost:8080
```

HTTP response

```
HTTP/1.1 404 Not Found
Date: Mon, 25 Feb 2019 12:36:37 GMT
Content-Length: 191
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked

{"timestamp":"2019-02-25T12:36:37.514+0000","status":404,"error":"Not Found","message":"Error during
reading simulation: Could not find simulation with id -1","path":"/api/v1/simulations/-1"}
```

Path parameters

Table 1. /api/v1/simulations/{id}

Parameter	Description
`id`	The simulation's id

GET All Simulations

HTTP request

```
GET /api/v1/simulations HTTP/1.1
Accept: application/json
Host: localhost:8080
```

HTTP response

```
HTTP/1.1 200 OK
Date: Mon, 25 Feb 2019 12:36:37 GMT
Content-Type: application/json;charset=UTF-8
Content-Length: 188
Transfer-Encoding: chunked

[{"id":1,"name":"Simulation1","created":"2019-02-25T13:36:37.203","horseJockeyCombinations":null},
{"id":2,"name":"Race1","created":"2019-02-25T13:36:37.32","horseJockeyCombinations":null}]
```

GET All Simulations filtered

HTTP request

```
GET /api/v1/simulations?name=Simulation HTTP/1.1
Accept: application/json
Host: localhost:8080
```

HTTP response

```
HTTP/1.1 200 OK
Date: Mon, 25 Feb 2019 12:36:37 GMT
Content-Length: 98
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked

[{"id":1,"name":"Simulation1","created":"2019-02-25T13:36:37.203","horseJockeyCombinations":null}]
```

Request parameters

Parameter	Description
name	Filters all simulations where the name contains this string