Einzelbeispiel (SE&PM/JAVA), Sommersemester 2018

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, welches **selbständig** zu entwickeln ist, erfolgreich gelöst werden.

Zu verwendende Technologien

Programmiersprache	Java 9
UI-Framework	JavaFX 9
Datenbank	H2 1.4.x
Logging	SLF4J & Logback 1.7.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 2017.x.x[1] in Kombination mit dem JavaFX SceneBuilder 9.x.x[2]. Beide Tools stehen im Informatiklabor zur Verfügung und unsere Tutor/innen bieten hierfür Unterstützung an.

Betreuung durch unsere Tutor/innen während der Eingangsphase

Bei Fragen und Unklarheiten während der Eingangsphase stehen Ihnen unsere Tutor/innen per TUWEL Diskussionsforum zur Verfügung. Zusätzlich gibt es betreute Laborzeiten, zu denen unsere Tutor/innen im Informatiklabor anwesend sind, um vor Ort bei der Problemlösung behilflich zu sein.

Wichtig: *Je genauer Sie hier Ihre Fragen formulieren, desto besser kann Ihnen geholfen werden.*

Labor Fragestunden: Termine und Anwesenheitszeiten finden Sie im TUWEL.

TUWEL: Auf unserer eLearning Plattform stellen wir ein moderiertes Diskussionsforum zur Verfügung, das von unseren Tutor/innen regelmäßig betreut wird.

1 Voraussetzungen

Wir gehen davon aus, dass Sie die Inhalte der Lehrveranstaltungen **Einführung in die Programmierung 1 & 2**, **Algorithmen und Datenstrukturen**, **Objektorientierte Modellierung** und **Objektorientierte Programmiertechniken** sowie **Datenbanksysteme** verstehen und auch praktisch anwenden können.

[1] https://www.jetbrains.com/idea/download/
[2] https://gluonhq.com/open-source/scene-builder/

Inhaltsverzeichnis

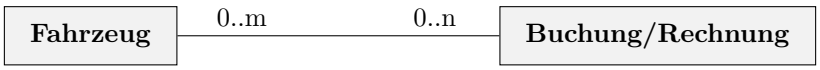
2 Angabe

Sie sind als Softwareentwickler/in in einem kleinen österreichischen Unternehmen tätig. Ihr Unternehmen wurde beauftragt, eine Softwarelösung für einen Fahrzeugverleih zu entwickeln.
Eine andere Abteilung hat bereits eine Anforderungsanalyse durchgeführt und die Wünsche des Kunden in Stories gefasst. Ihr/e technische/r Architekt/in und Ihr/e Qualitätsmanager/in haben zusätzliche Anforderungen an die Umsetzung der Software formuliert.
Um die Komplexität der Aufgabenstellung in Grenzen zu halten, müssen Sie keine Rücksicht auf die gewerberechtlichen und steuerrechtlichen Rahmenbedingungen der Software nehmen, sofern dies nicht explizit in einer Story erwähnt ist.

In der Aufgabenstellung wird zwischen Userstories und Techstories unterschieden. Userstories stellen dabei konkrete Anwendungsfälle in der zu erstellenden Software, aus Sicht eines bestimmten Steakeholders, dar. Techstories enthalten Anforderungen an die Implementierung sowie weitere Implementierungsdetails.

2.1 Domänenmodell

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



Wichtig: *Zur Vereinfachung betrachten wir Rechnungen als abgeschlossene Buchungen!*

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 auf ihr zudem Ihren Namen und Ihre Matrikelnummer.

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. Natürlich können Sie später auch bei bereits implementierten Userstories nacharbeiten. Dieses Vorgehen hilft Ihnen dabei möglichst viele der Userstories abzuschließen.

2.3.2 Techstories

Lesen Sie alle Techstories aufmerksam durch bevor Sie mit der Implementierung beginnen.
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 Programms zu erleichtern.

2.5 Userstories (80 Storypoints)

Userstories stellen Anforderungen eines Stakeholders an das Produkt und den Produktflow dar. Jede Userstory muss dabei in allen Schichten (Userinterface, Businesslogik, Persistenz) implementiert werden.

Wichtig: *Die Aufschlüsselung der Userstories auf Stakeholder dient ausschließlich dem leichteren Verständnis. Sie sollen kein Mehrbenutzersystem entwickeln.*

2.5.1 Betreiber/in

ID: 01	**Titel:** Als Betreiber/in möchte ich neue Fahrzeuge (Autos, Motorräder, Fahrräder, Segways, usw.) im System anlegen.

- Jedes angelegte Fahrzeug muss in einer Datenbank gespeichert werden.
- Zu jedem Fahrzeug müssen dabei folgende Werte gespeichert werden:
  - Bezeichnung (verpflichtend)
  - Baujahr (verpflichtend)
  - Beschreibung (optional)
  - Sitzplätze (optional)
  - Kennzeichen (verpflichtend, wenn ein Führerschein erforderlich ist)
  - Antriebsart: motorisiert oder muskelkraft (verpflichtend)
  - Leistung in kW (verpflichtend, wenn motorisiert)
  - Grundpreis pro Stunde (verpflichtend)
  - Zeitpunkt der Erstellung (automatisch)

Storypoints: 6

ID: 02	**Titel:** Als Betreiber/in möchte ich zu jedem Fahrzeug ein Bild abspeichern können.

- Zu jedem Fahrzeug muss ein Bild des Typs JPEG oder PNG abgespeichert werden können.
- Das Bild zum Fahrzeug ist optional.
- Bilder dürfen nicht in der Datenbank, sondern müssen in einem eigenen Ordner abgespeichert werden.
- Jedes Bild darf bis zu 5Mb groß sein.
- Jedes Bild muss mindestens 500x500 Pixel groß sein.
- Sofern ein Bild zum Fahrzeug gespeichert wurde, muss es in der Detailansicht zum Fahrzeug angezeigt werden.

Storypoints: 5

ID: 03	**Titel:** Als Betreiber/in möchte ich alle Fahrzeuge einer oder mehreren Führerscheinklassen zuordnen können.

- Jedes Fahrzeug kann einer oder mehreren Führerscheinklassen zugeordnet sein.
- Die verfügbaren Führerscheinklassen sind vom Gesetzgeber vorgegeben, müssen nicht änderbar sein und können fest im Programm hinterlegt sein.
- Folgende Führerscheinklassen müssen im System vorhanden sein:
  - A (Motorräder und Leichtmotorräder)
  - B (PKW und Leichtmotorräder)
  - C (LKW sowie PKW über 3,5 Tonnen)
- Alternativ kann auch vermerkt werden, dass kein Führerschein für das Fahrzeug benötigt wird (Segway, Fahrrad, usw.).

Storypoints: 6

ID: 04	**Titel:** Als Betreiber/in möchte ich Fahrzeuge bearbeiten können.

- Beim Bearbeiten von Fahrzeugen müssen alle Werte sowie das Bild geändert werden können.
- Beim Bearbeiten von Fahrzeugen dürfen sich bereits erstellte Rechnungen nicht ändern.
- Bei jedem Fahrzeug muss vermerkt sein, wann es zuletzt bearbeitet wurde.
- Die Zeitpunkte der Erstellung sowie der letzten Bearbeitung dürfen nicht verändert werden können.

Storypoints: 5

ID: 05	**Titel:** Als Betreiber/in möchte ich Fahrzeuge, die nicht mehr im Fuhrpark vorhanden sind, aus dem System löschen können.

- Das Löschen von Fahrzeugen darf nicht rückgängig gemacht werden können, sollte man das Fahrzeug wieder anbieten, dann wird es einfach neu angelegt.
- Beim Löschen von Fahrzeugen müssen mehrere Fahrzeuge gleichzeitig ausgewählt werden können.
- Beim Löschen von Fahrzeugen muss der/die Nutzer/in noch einmal rückgefragt werden, dabei wird er/sie informiert, welche Fahrzeuge gelöscht werden.
- Beim Löschen von Fahrzeugen dürfen sich bereits erstellte Rechnungen nicht ändern.

Storypoints: 5

ID: 06	**Titel:** Als Betreiber/in möchte ich statistische Informationen zu meinen Fahrzeugen und Buchungen anzeigen können.

- Um festzustellen, welche Fahrzeuge an welchen Tagen besonders gefragt sind, müssen die Verleihdaten statistisch aufbereitet werden.
- Die Statistik muss mittels passender Charts (BarChart, LineChart) grafisch aufbereitet werden.
- Folgende Statistiken müssen angezeigt werden:
  - Alle Umsätze über einen von Nutzer/innen frei einzugebenden Zeitraum (von, bis) als LineChart für von Nutzer/innen frei kombinierbare Führerscheinklassen (je eine Line pro gewählter Führerscheinklasse).
    Achten Sie darauf, Fahrzeuge für die kein Führerschein benötigt wird, auch als Möglichkeit aufzuführen.
  - Die Anzahl der Verleihvorgänge pro Wochentag (Mo, Di, Mi, ...) über einen von Nutzer/innen frei einzugebenden Zeitraum (von, bis) als BarChart für von Nutzer/innen frei kombinierbare Führerscheinklassen.
    Achten Sie darauf, Fahrzeuge für die kein Führerschein benötigt wird, auch als Möglichkeit aufzuführen.
- Da Buchungen oft im Voraus getätigt werden soll die Statistik alle Buchungen enthalten, unabhängig ob diese Bereits abgerechnet sind.

Storypoints: 10

2.5.2 Mitarbeiter/in

ID: 07	**Titel:** Als Mitarbeiter/in möchte ich Buchungen entgegennehmen können

- Buchungen sind „offene“ (noch nicht „abgeschlossene“ oder „stornierte“) Rechnungen.
- Jede Buchung enthält immer folgende Daten:
  - Name des Kunden
  - IBAN oder Kreditkartennummer des Kunden
  - Zeitraum der Buchung
  - Zeitpunkt der Erstellung
  - Gebuchte Fahrzeuge
  - Preis pro Fahrzeug
  - Gesamtpreis
- Bei Fahrzeugen für die ein Führerschein benötigt wird, muss für jedes Fahrzeug die Führerscheinnummer sowie das Erteilungsdatum der Lenkberechtigung des Fahrers / der Fahrerin gespeichert werden.
- Fahrzeuge, die ausschließlich mit einem Führerschein der Klasse A oder C gelenkt werden dürfen, können nur von Personen entliehen werden, die die Lenkberechtigung mindestens drei Jahre besitzen.
- Fahrzeuge können im selben Zeitraum nicht mehrfach gebucht werden.
- Vor dem Anlegen einer Buchung muss der/die Mitarbeiter/in gewarnt werden, falls keine kostenfreie Stornierung mehr möglich sein wird.
- Sowohl IBAN[3] als auch Kreditkartennummer[4] müssen korrekt validiert[5] werden.

Storypoints: 9

ID: 08	**Titel:** Als Mitarbeiter/in möchte ich Buchungen abschließen.

- Es dürfen nur „offene“ (noch nicht „abgeschlossene“ oder „stornierte“) Buchungen abgerechnet werden.
- Beim Abrechnen wechselt eine Bestellung vom Status „offen“ in den Status „abgerechnet“. Sie werden ab dem Zeitpunkt als Rechnung behandelt.
- Beim Abrechnen muss der Zeitpunkt der Abrechnung gespeichert werden.
- Ab dem Zeitpunkt der Abrechnung werden alle enthaltene Fahrzeuge für den Restzeitraum wieder buchbar.
- Rechnungen dürfen nicht gelöscht werden.
- Zu bereits abgerechneten Rechnungen dürfen keine weiteren Fahrzeuge hinzugefügt werden.

Storypoints: 6

[3] Beispiele für gültige IBANs: http://ibanvalidieren.de/beispiele.html
[4] Beispiele für gültige Kreditkartennummern: http://www.getcreditcardnumbers.com/
[5] Open Source Bibliothek zur Validierung von IBANs und Kreditkartennummern: Apache Commons Validator https://commons.apache.org/proper/commons-validator/

ID: 09	**Titel:** Als Mitarbeiter/in möchte ich Buchungen stornieren können

- Es dürfen nur „offene“ (noch nicht „abgeschlossene“ oder „stornierte“) Rechnungen storniert werden.
- Buchungen können bis zu einer Woche vor Beginn des Buchungszeitraums kostenlos storniert werden, in dem Fall wird die Buchung einfach gelöscht.
- Buchungen können bis zu 72 Stunden vor Beginn der Buchung mit einer Stornogebühr in der Höhe von 40% des Gesamtpreises storniert werden.
- Buchungen können bis 24 Stunden vor Beginn der Buchung mit einer Stornogebühr in der Höhe von 75% des Gesamtpreises storniert werden.
- Ab 24 Stunden vor Beginn der Buchung ist eine Stornierung nicht mehr möglich und die volle Leihgebühr wird fällig.
- Im Fall einer Stornierung werden alle enthaltene Fahrzeuge für den Zeitraum wieder buchbar.
- Buchungen werden bei der Stornierung bezahlt und wechseln damit in den Status „storniert“. Sie werden ab dem Zeitpunkt als Rechnung behandelt.
- Zu bereits stornierten Rechnungen dürfen keine weiteren Fahrzeuge hinzugefügt werden.

Storypoints: 6

2.5.3 Betreiber/in & Mitarbeiter/in

ID: 10	**Titel:** Als Betreiber/in & Mitarbeiter/in möchte ich alle Buchungen & Rechnungen anzeigen können.

- Es werden alle Buchungen & Rechnungen absteigend nach Zeitpunkt des Beginn des Buchungszeitraums sortiert angezeigt.
- Die Anzeige muss in Tabellenform erfolgen.
- Dabei muss für jede Buchung & Rechnung der Name des Kunden, der Buchungszeitraum, der Status („offen“, „abgeschlossen“ oder „storniert“) sowie der Gesamtpreis angezeigt werden.
- Vor dem Beginn des Buchungszeitraums kann jede Buchung ausgewählt, angezeigt, bearbeitet oder storniert werden.
- Nach dem Beginn des Buchungszeitraums kann jede Buchung ausgewählt, angezeigt und abgeschlossen werden.
- Die Detailsansicht der Buchung enthält alle Daten die beim Anlegen eingegeben wurden.
- Jede Rechnung („abgeschlossen“ oder „storniert“) kann angezeigt werden.
- Die Detailansicht einer Rechnung muss folgende Daten beinhalten:
  - Rechnungsnummer
  - Status („abgeschlossen“ oder „storniert“)
  - Zeitraum der Buchung
  - Zeitpunkt der Abrechnung
  - Gebuchte Fahrzeuge
  - Preis pro Fahrzeug
  - Rechnungssumme

Storypoints: 8

ID: 11	**Titel:** Als Betreiber/in & Mitarbeiter/in möchte ich nach Fahrzeugen suchen können.

- Um Fahrzeuge zu bearbeiten, zu löschen oder in Buchungen aufzunehmen, muss die Möglichkeit bestehen, nach den Fahrzeugen zu suchen.
- Fahrzeuge müssen dabei nach folgenden Kriterien gesucht und gefiltert werden können:
  - Führerscheinklasse/n
    (das Fahrzeug muss mit einer der Klassen oder ohne Führerschein lenkbar sein)
  - Grundpreis pro Stunde (mindestens)
  - Grundpreis pro Stunde (höchstens)
  - Zeitraum der Buchung
  - Bezeichnung
    (auch Teile davon, Groß- und Kleinschreibung muss ignoriert werden)
  - Antriebsart
  - Sitzplätze
- Die Kriterien müssen dabei frei kombinierbar sein.
- Sollte mehr als ein Kriterium ausgewählt werden, dürfen nur Fahrzeuge gefunden werden, die allen Kriterien entsprechen.

Storypoints: 7

ID: 12	**Titel:** Als Betreiber/in & Mitarbeiter/in möchte ich Ergebnisse der Fahrzeugsuche anzeigen können.

- Nach einer Suche müssen die Suchergebnisse tabellarisch aufbereitet werden.
- Aus der Tabelle müssen Fahrzeuge zum Bearbeiten, Löschen und zur Aufnahme in Buchungen ausgewählt werden können.
- In der Tabelle müssen folgende Daten dargestellt werden:
  - Bezeichnung
  - Führerscheinklassen
  - Antriebsart
  - Grundpreis pro Stunde
- Aus den Suchergebnissen muss man auch zur Detailansicht eines Fahrzeugs wechseln können.
- Die Detailansicht muss alle Fahrzeugdaten sowie das Bild des Fahrzeugs beinhalten.
- In der Detailansicht muss es die Möglichkeit geben, das Fahrzeug zu bearbeiten, zu löschen und in die Buchung aufzunehmen.

Storypoints: 7

2.6 Techstories (80 Storypoints)

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

2.6.1 Qualitätsmanager/in

ID: 13	**Titel:** Als Qualitätsmanager/in möchte ich, dass das Programm Logfiles schreibt.

- Der Logfile muss `sepm-<datum>.log` heißen.
- Für jeden Tag muss ein eigener Logfile erzeugt werden.
- Der Logfile muss alle auftretenden Fehler in Form einer Errormessage enthalten.
- Der Logfile muss alle vom User durchgeführten Aktionen als Infomessage enthalten.
- Der 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: 6

ID: 14	**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: 6

ID: 15	**Titel:** Als Qualitätsmanager/in möchte ich, dass Kernfunktionalitäten des Programms getestet sind.

- Sie müssen insgesamt acht Unittests erstellen. Für zwei verschiedene Userstories von zwei unterschiedlichen Stakeholdern jeweils vier Unittests.
- Es sollen keine Unittests für das Userinterface erstellt werden.
- Die Unittests müssen unterschiedliche Schichten (min. drei Tests pro Schicht) testen.
- Es müssen Test für den Normalfall und Fehlerfall (siehe 3.5.1) erstellt werden.

Storypoints: 8

ID: 16	**Titel:** Als Qualitätsmanager/in möchte ich, dass alle Eingaben validiert werden.

- Eingabefelder müssen, je nachdem ob es sich um Zahlenwerte, Datumswerte oder Textwerte handelt, entsprechende Validierungen aufweisen.
- Felder müssen entsprechend ihrem Zweck validieren, zum Beispiel keine negativen Preise oder Namen mit Mindestlänge/Maximallänge.
- Startdaten müssen immer vor Enddaten liegen.
- Es muss bei der Eingabe aller Daten klar ersichtlich sein, ob es sich um ein Pflichtfeld handelt oder nicht.

Storypoints: 8

2.6.2 Technische/r Architekt/in

ID: 17	**Titel:** Als Technische/r Architekt/in erwarte ich einen sauberen Git Workflow.

- Es müssen regelmäßig Commits erstellt werden.
- Ein Commit besteht immer aus einer zusammenhängenden Einheit.
- Ein Commit gehört immer zu genau einer Userstory und/oder Techstory.
- Jeder Commit enthält eine aussagekräftige Commitmessage sowie die Nummer der Userstory als Referenz.

Storypoints: 6

ID: 18	**Titel:** Als Technische/r Architekt/in erwarte ich die Verwendung eines Build und Dependency Managements.

- Abhängigkeiten müssen, sofern möglich, mittels Maven verwaltet werden.
- Das Programm muss über Maven gebaut, gestartet und getestet werden können.
- Das Programm muss mit dem Befehl:
  `mvnw clean compile`
  erfolgreich gebaut werden können.
- Das Programm muss mit dem Befehl:
  `mvnw clean test`
  erfolgreich getestet werden können.
- Das Programm muss mit dem Befehl:
  `mvnw clean compile exec:java`
  gestartet werden können.

Storypoints: 6

ID: 19	**Titel:** Als Technische/r Architekt/in erwarte ich eine saubere Umsetzung der Datenbank.

- Die Datenbank muss im embedded Modus 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 Datenbankverbindung muss das Singleton[6] Pattern eingesetzt werden.
- SQL Statements müssen als Prepared Statement ausgeführt werden.
- Primärschlüssel müssen laufende Nummern sein, die von der Datenbank erstellt werden. Sie dürfen außerdem an keiner Stelle im Programm angezeigt werden.
- Um Relationen abbilden zu können müssen sinnvolle Fremdschlüssel gewählt werden.

Storypoints: 10

ID: 20	**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[7] Pattern verwendet werden.
- Verwenden von Data-Transfer-Objects zur Kommunikation zwischen den Schichten.
- Verwenden von Data-Transfer-Objects für die Definition von Suchkriterien und Filtern.
- Verwenden von Exceptions zum Transportieren von Fehlern.

Storypoints: 16

ID: 21	**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 (zum Beispiel beim Laden/Speichern von Bildern).
- Suchen müssen aus Performancegründen in der Datenbank vorgenommen werden.

Storypoints: 6

[6] https://sourcemaking.com/design_patterns/singleton
[7] https://docs.oracle.com/javase/tutorial/java/concepts/interface.html

ID: 22	**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, sofern Sie nicht vom Programm selbst behoben werden können, dem/der Nutzer/in kommuniziert werden.
- Fehler müssen zumindest im Log aufscheinen.
- Fehler müssen entweder behandelt oder weitgereicht werden.
- Das Programm darf zu keinem Zeitpunkt abstürzen.
- Das Programm darf sich zu keinem Zeitpunkt in einem Zustand befinden der einen Neustart des Programms erfordert.

Storypoints: 8

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 Ihre Entwicklungsumgebung einrichten. Nachdem Sie IntelliJ IDEA - Community Edition installiert haben, laden Sie das Beispielprojekt aus TUWEL herunter. Entpacken Sie das Beispielprojekt in einen Ordner ihrer Wahl und importieren Sie das Projekt in Ihre Entwicklungsumgebung.

Das zur Verfügung gestellte Beispielprojekt beinhaltet bereits einen sehr einfachen Durchstich (eine Implementierung durch mehrere Schichten). Die Anwendung besteht aus einem Fenster in dem Sie einen Button drücken können. Nach dem Druck auf den Button startet eine, circa zwei Sekunden lange, Berechnung deren Ergebnis danach auf dem Bildschirm ausgegeben wird.
Zusätzlich zu diesem einfachen Durchstich ist in dem Beispielprojekt bereits das Build und Dependency Management Tooling mittels Maven aufgesetzt.

Nachdem Sie das Projekt nun erfolgreich importiert 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 Beispielprojekts

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

Folgende Ordner und Dateien sind im Beispielprojekt enthalten:

`.idea/` beinhaltet alle Projektdateien die von IntelliJ benötigt werden (Einstellungen usw.). Dieser Ordner sollte nicht verändert werden.
`.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, das sind insbesondere FXML Dateien.
`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.
`.gitattributes` enthält eine einfache Konfiguration für das Versionskontrollsystem git.
`.gitignore` enthält eine Liste an Dateien die nicht in git versioniert werden sollen.
`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.

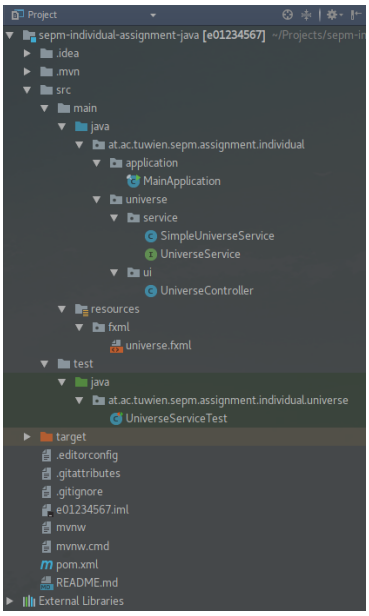


Abbildung 1: Projekt Struktur in IntelliJ

3.2 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 Buildprozesses 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 exec:java` startet das Programm.

Wichtig: Der Befehl `mvnw` kann nur im Hauptordner des Projekts ausgeführt werden. Unter Linux und OSX 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 Abhängigkeiten auf SLF4J & Logback und H2. Außerdem hat das Einzelbeispiel eine Abhängigkeit auf JUnit zur Erstellung und Ausführung von Tests.
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, siehe Listing 1.

```
1    <groupId>at.ac.tuwien.sepm.assignment.individual</groupId>
2    <artifactId>e01234567<!-- add your matriculation number here --></artifactId>
3    <version>1.0-SNAPSHOT</version>
```

Listing 1: Projektkonfiguration im pom.xml

Außer den Projektinformationen ist für Sie vor allem der `dependencies` Abschnitt relevant, siehe Listing 2. Hier wurden bereits die Benötigten Bibliotheken für Tests sowie Logging hinzugefügt. Die Bibliothek für die Datenbank H2 müssen Sie selbst hinzufügen. Dazu können Sie zur Suche von Abhängigkeiten *Maven Repository Search* [8] oder *Maven Central Search* [9] verwenden.

```
1        <!-- compile dependencies -->
2        <dependency>
3            <groupId>org.slf4j</groupId>
4            <artifactId>slf4j-api</artifactId>
5            <version>${slf4j.version}</version>
6        </dependency>
7        <!-- runtime dependencies -->
8        <dependency>
9            <groupId>ch.qos.logback</groupId>
10           <artifactId>logback-classic</artifactId>
11           <version>${logback.version}</version>
12           <scope>runtime</scope>
13       </dependency>
14       <dependency>
15           <groupId>ch.qos.logback</groupId>
16           <artifactId>logback-core</artifactId>
17           <version>${logback.version}</version>
18           <scope>runtime</scope>
19       </dependency>
20       <!-- test dependencies -->
21       <dependency>
22           <groupId>junit</groupId>
23           <artifactId>junit</artifactId>
24           <version>${junit.version}</version>
25           <scope>test</scope>
26       </dependency>
27   </dependencies>
```

Listing 2: Abhängigkeiten im pom.xml

Für die Versionen werden dabei Platzhalter verwendet die unter `properties` definiert sind, siehe Listing 3.

```
1    <properties>
2        <!-- build properties -->
3        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
4        <maven.compiler.source>9</maven.compiler.source>
5        <maven.compiler.target>9</maven.compiler.target>
6        <exec.mainClass>at.ac.tuwien.sepm.assignment.individual.application.
             MainApplication</exec.mainClass>
7        <!-- compile dependencies -->
8        <slf4j.version>1.8.0-beta0</slf4j.version>
9        <!-- runtime dependencies -->
10       <logback.version>1.2.3</logback.version>
11       <!-- test dependencies -->
12       <junit.version>4.12</junit.version>
13       <!-- plugins -->
14       <maven-surefire-plugin.version>2.20.1</maven-surefire-plugin.version>
15   </properties>
```

Listing 3: Properties im pom.xml

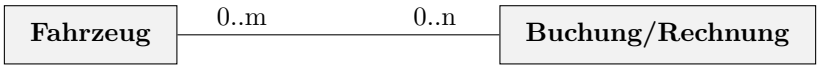
[8] https://mvnrepository.com/
[9] https://search.maven.org/

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.

3.3 Vom Domänenmodell zur Datenbank

Ein Domänenmodell stellt Ihre Sicht/Ihr Verständnis des Datenmodells in der realen Welt dar. Es ist daher wie wir gleich sehen werden 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 Hilfstabelle benötigen. Diese ist aber ein rein technisches Implementierungsdetail und daher nicht Teil des Domänenmodells.

Überführen Sie die Entitäten in „CREATE TABLE ...“ Statements Ihrer Datenbank. Erstellen Sie hierzu am besten eine Datei, zum Beispiel `database.sql` und fügen Sie dort die Statements zum Erstellen und Befüllen der Datenbanktabellen ein.
Sie müssen und sollen nicht das gesammte Domänenmodell sofort in SQL umsetzen. Am besten ist es wenn Sie Ihre erstellte Datei (`database.sql`) laufend erweitern und verbessern. Wichtig ist, dass Sie regelmäßig die Einhaltung aller Userstories und Techstories überprüfen.

3.4 Reihenfolge der Implementierung

Bei der Reihenfolge der Implementierung sollten Sie nach Kundenpriorität vorgehen. Es bringt Ihnen beispielsweise nichts, wenn Sie komplizierte statistische Auswertungen über Ihre Daten machen können wenn es nocht 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 Layer: Persistenz, Service und Userinterface. Vergessen Sie dabei nicht auf die Einhaltung der Techstories.

Um die verschiedenen Schichten von einander zu entkoppeln verwenden Sie DTOs (Data-Transfer-Objects), Exceptions und Interfaces. Auch hier empfiehlt es sich wieder zwischen Interface und Implementierung zu trennen. Außerdem ist es wichtig, dass das Userinterface nie direkt auf die Persistenzschicht zugreift und umgekehrt.

3.4.1 Persistenz

Die Persistenzschicht regelt den Zugriff auf Ihre Daten, dabei ist es egal ob die Daten im Filesystem, in einer Datenbank oder in einem Webservice 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 [10]
- Data-Access-Object Pattern [11]
- Data-Transfer-Objects [12]

Bevor Sie in Ihrem Programm zum ersten Mal eine Datenbankverbindung aufbauen können müssen Sie die Datenbanktreiber laden, siehe Listing 4. Der hier abgebildete Code lädt zuerst die Datenbanktreiber und verbindet sich dann zu einer Datenbank namens „sepm“ dabei wird der Benutzername „dbUser“ und das Passwort „dbPassword“ verwendet. Direkt nach dem Verbindungsaufbau wird eine SQL-Datei ausgeführt, die sich unter `resources/sql/createAndInsert.sql` befindet.

```
1  import java.sql.*;
2
3  public class Test {
4    public static void main(String[] a) throws Exception {
5      Class.forName("org.h2.Driver");
6      Connection connection = DriverManager.getConnection(
7        "jdbc:h2:~/sepm;INIT=RUNSCRIPT␣FROM␣'classpath:sql/createAndInsert.sql'",
8        "dbUser",
9        "dbPassword"
10       );
11     connection.close();
12   }
13 }
```

Listing 4: Datenbanktreiber laden und Verbindung aufbauen

Sollte die Datenbank noch nicht existieren wird Sie automatisch erstellt. In diesem Beispiel wird der Datenbankfile direkt im Homeverzeichnis der Nutzer/in erstellt. Dabei werden folgende Dateien angelegt:
`~/sepm.mv.db`
`~/sepm.trace.db`

Wichtig: *Vergessen Sie nicht, 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 sobald Sie h2 als dependency zu Ihrem Projekt hinzugefügt haben. Damit können Sie sich Ihre Datenbank ansehen.

[10] https://docs.oracle.com/javase/tutorial/java/concepts/interface.html
[11] https://www.oracle.com/technetwork/java/dataaccessobject-138824.html
[12] https://www.oracle.com/technetwork/java/transferobject-139757.html

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

Listing 5 zeigt ein mögliches Beispiel für einen SQL-Script zur Initialisierung und erstmaligen Befüllung einer Datenbank.

```
1  -- create table product if not exists
2  CREATE TABLE IF NOT EXISTS product (
3    -- use auto incrementing IDs as primary key
4    id BIGINT AUTO_INCREMENT PRIMARY KEY,
5    name VARCHAR(255),
6    description TEXT
7  );
8
9  -- insert initial test data
10 INSERT INTO product
11 -- select all entities to be inserted
12   SELECT * FROM (
13     -- the following select returns nothing and is just
14     -- for convenience to make the following lines look pretty
15     SELECT * FROM product WHERE FALSE
16     -- followed by each row which should be inserted in the table
17     -- IDs are hardcoded in the default insert to be able to insert relations
           afterwards
18     UNION SELECT 1, 'Tea', 'Best␣served␣in␣england␣at␣4pm.'
19     UNION SELECT 2, 'Coffee', 'A␣hot␣drink,␣which␣programmers␣need␣to␣function.'
20     UNION SELECT 3, 'Pizza', 'Fast␣and␣delicious␣italian␣food.'
21   )
22 -- run the insert only when the product table is empty
23 WHERE NOT EXISTS(SELECT * FROM product);
```

Listing 5: Beispielscript zum Initialisieren und Befüllen der Datenbank

3.4.2 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 UI zu validieren und an die Persistenzschicht durchzureichen. Allerdings ermöglicht sie es die Anwendung einfacher anzupassen und modularer zu gestalten.
Auch im Einzelbeispiel wird es in einigen Fällen notwendig sein, dass die Serviceschicht mehr Aufgaben übernimmt als nur die Validierung.
Achten Sie bei der Erstellung Ihrer Serviceklassen darauf, dass Ihre Services nur kleine zusammenhängende Aufgaben erfüllen und vermeiden sehr große monolitische Serviceklassen.

Das zur Verfügung gestellte Beispielprojekt beinhaltet bereits eine sehr einfach Serviceschicht für den „Universe Calculator“. In ihr wird eine „komplexe“ Berechnung durchgeführt.
In Listing 6 sehen Sie das definierte Interface. Es beinhaltet bereits alle Methodendefinitionen und Dokumentation. Versuchen Sie auch sinnvolle Namen für Ihre Interfaces zu wählen und vermeiden Sie beispielsweise ein vorangestelltes „I“.
Listing 7 zeigt die Implementierung des zuvor erstellten Interfaces. Achten Sie darauf für Ihre Implementierungen aussagekräftige Namen zu verwenden. Unsere sehr einfache Implementierung des `UniverseService` heißt daher `SimpleUniverseService`.

```java
package at.ac.tuwien.sepm.assignment.individual.universe.service;

/**
 * The <code>UniverseService</code> is capable to calculate the answer to
 * <blockquote>Ultimate Question of Life, the Universe, and Everything</blockquote>.
 * Depending on the implementation it might take a while.
 */
public interface UniverseService {

    /**
     * Calculate the answer to the ultimate question of life, the universe, and everything.
     *
     * @return the answer to the ultimate question of life, the universe, and everything
     */
    String calculateAnswer();

}
```

Listing 6: Interface des Universe Service

```java
package at.ac.tuwien.sepm.assignment.individual.universe.service;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.lang.invoke.MethodHandles;
import java.time.Duration;

import static java.time.temporal.ChronoUnit.SECONDS;

public class SimpleUniverseService implements UniverseService {

    private static final Logger LOG = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());
    private static final int SLEEP_SECONDS = 2;

    @Override
    public String calculateAnswer() {
        LOG.debug("called calculateAnswer");
        // sleep to simulate heavy load
        try {
            LOG.trace("Going to sleep for {} seconds", SLEEP_SECONDS);
            Thread.sleep(Duration.of(SLEEP_SECONDS, SECONDS).toMillis());
        } catch (InterruptedException e) {
            LOG.warn("Failed to sleep cause {}", e.getMessage());
        }
        return "42!";
    }

}
```

Listing 7: Implementierung des Universe Service

3.4.3 Userinterface

Wenn Sie bei diesem Punkt angelangt sind, ist die Userstory bereits bis auf die Implementierung der grafischen Benutzeroberfläche komplett fertig. Achten Sie darauf, dass alle Elemente Ihres UI, die Zugriff auf ein Service benötigen, auch eine Membervariable für das entsprechende Service haben und dieses von der erstellenden Komponente gesetzt bekommt.
Für eine besonders gute UX (User Experience) ist es von Vorteil wenn alle Eingaben zusätzlich zum Service auch direkt während der Eingabe im UI validiert werden und der User sofort eine Rückmeldung über die Validität der Daten erhält.

Wir empfehlen an geeigneten Stellen das Hinzufügen von Kontextmenüs. So können Sie zum Beispiel dem/der Nutzer/in die Möglichkeit geben Zeilen zu markieren und mittels Rechtsklick zu bearbeiten und zu löschen.

Für das Erstellen der UI sollen Sie den grafischen UI Designer JavaFX SceneBuilder verwenden.

3.5 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 Vorbedingung 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.

3.5.1 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 Bild auswählt, dass auf Grund einer fehlenden Berechtigung nicht geladen werden kann. Ein anderers Beispiel für einen Fehlerfall wäre es, wenn versucht wird ein Bild zu laden, das die zugelassene Größe überschreitet. In beiden Fällen muss eine entsprechende Exception ausgelöst werden mit deren Hilfe der Fehler behandelt beziehungsweise an den/die Nutzer/in kommunziert werden kann.

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

3.5.2 Manuelle Tests

Manuelle Tests werden im Rahmen des Einzelbeispiels den Großteil Ihrer Testtätigkeit einnehmen. Hierbei ist es wichtig, dass sie strukturiert vorgehen und die Tests in regelmäßigen Abständen wiederholen.
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. Produkt anlegen <Salat, € 1.50>
2. Produkt verkaufen <Salat>
3. Produkt bearbeiten <Salat, € 2.00>
4. Rechnung anzeigen <Salat, € 1.50>

3.5.3 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: *Vergessen Sie nicht, 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.*

Listing 8 zeigt einen einfachen Test für das UniverseService. 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 von einander abhängig sind.
Außerdem sollten Tests so simpel und selbsterklärend wie möglich geschrieben sein. Vermeiden Sie in Tests die Verwendung von Schleifen und sowie kompliziertem Exceptionhandling. Achten Sie auch darauf, dass Ihre Testklassen und Methoden aussagekräftige Namen haben.

```java
package at.ac.tuwien.sepm.assignment.individual.universe;

import at.ac.tuwien.sepm.assignment.individual.universe.service.
    SimpleUniverseService;
import at.ac.tuwien.sepm.assignment.individual.universe.service.UniverseService;
import org.junit.Assert;
import org.junit.Test;

import static org.hamcrest.core.Is.is;

public class UniverseServiceTest {

    private final UniverseService universeService = new SimpleUniverseService();

    @Test
    public void testSimpleUniverseService() {
        Assert.assertThat(universeService.calculateAnswer(), is("42!"));
    }

}
```

Listing 8: Test für das UniverseService

3.6 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 Repositorys auf Ihrem Entwicklungssystem haben. In diesem machen Sie Ihre Commits die Sie danach zum zur Verfügung gestellten 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
```

3.6.1 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 folgeden 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 Repositorys 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 Repositorys direkt in RESET.

3.6.2 Add & Commit

Um Ändungen 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.
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.

Abbildung 2: Git Commit: https://xkcd.com/1296/

Beispiele für schlechte Commitmessages:

- „Neue Implementierung hinzugefügt“
- „Added changes to Product, Invoice and Tables“
- „Fix Bug“

Beispiele für gute Commitmessages:

- „Fixed bug where image is not stored correctly [story: 2]“
- „Added validation #1, #16“

3.6.3 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 gepusht werden. Am besten pushen Sie Ihre Änderungen so häufig wie möglich. So stellen Sie sicher, dass 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.

3.7 Weiterführende Links & Literatur

- H2 1.4.x
  - https://www.h2database.com/html/main.html
- SLF4J & Logback 1.7.x
  - https://www.slf4j.org/
- 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 2017.x.x
  - https://www.jetbrains.com/idea/download/
- JavaFX SceneBuilder 9.x.x
  - https://gluonhq.com/open-source/scene-builder/
- JavaFX
  - JavaFX Documentation Project: https://fxdocs.github.io/docs/
  - JavaFX Tutorial: http://code.makery.ch/library/javafx-8-tutorial/
  - Dialoge: http://code.makery.ch/blog/javafx-dialogs-official/
- 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. Sollte eine Schicht bei einer Userstory fehlen, wird diese nicht abgenommen und Sie erhalten keine Punkte für die betreffende Userstory.
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!

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 %	75 - 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 VCS (Version Control System) vorhanden. **Nur Code und Dokumentation, die im VCS liegen, werden für die Bewertung herangezogen.**
- Bitte präsentieren Sie Ihre Abgabe nach Möglichkeit auf Ihrem eigenen Laptop!

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**.
   - **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.
- Das **grafische Interface** Ihrer Implementierung muss **nicht händisch programmiert** werden, wir raten Ihnen zur Verwendung des JavaFX SceneBuilder.
- 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 einer Forschungsbereich zugeordnet, dabei versuchen wir nach Möglichkeit, Ihren Wunsch zu berücksichtigen. Je nach Forschungsbereich 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, 08. April 2018, 23:55