Von DDD zu BDD: Methoden, Werkzeuge, Fallstudien

Bachelorarbeit

Studiengang Informatik
OST - Ostschweizer Fachhochschulen
Campus Rapperswil-Jona

Frühjahrssemester 2021

Autoren:	Timothée Moos, Saskia Stillhart
Betreuer:	Prof. Dr. Olaf Zimmermann
Experte:	Dr. Daniel Lübke
Gegenleserin:	Prof. Dr. Nathalie Weiler

Aufgabenstellung

Aufgabenstellung Bachelorarbeit
Timothée Moos, Saskia Stillhart

Von DDD zu BDD: Methoden, Werkzeuge, Fallstudien

1. Auftraggeber und Betreuer

Diese Bachelorarbeit wird in Zusammenarbeit mit dem Institut für Software (IFS) durchgeführt.

Betreuer (HSR/OST):

Prof. Dr. Olaf Zimmermann, Institut für Software, olaf.zimmermann@ost.ch

2. Ausgangslage und Ziele

Für die Domänenmodellierung und das Software-Design werden heute vermehrt die taktischen und strategischen Patterns aus dem Domain-Driven Design (DDD) eingesetzt. Komplementäre Praktiken sind das Behavior-Driven Development (BDD) mit dem Anspruch "Specification by Example" auch zur systematischen Identifikation von Testcases; eine hybride Arbeitstechnik ist das Example Mapping.

Für DDD können Online-Whiteboards wie miro und spezifische Modellierungswerkzeuge wie Context Mapper (DDD) eingesetzt werden; BDD wird z.B. von den Open Source Tools JBehave und Cucumber unterstützt. Für das Example Mapping haben sich noch keine dedizierten Werkzeuge durchgesetzt.

In der Literatur fehlen aktuell gesicherte Erkenntnisse ("Best Practices"), wie die genannten Praktiken sinnvoll kombiniert werden können. Test Cases werden meist manuell mit Hilfe von Basiswerkzeugen für Entwickler wir JUnit erstellt. Die Schnittstelle zwischen den Werkzeugen für DDD und BDD wurden bisher nicht ausreichend wissenschaftlich untersucht; es fehlen Integrationskonzepte und deren Umsetzung in Modellierungswerkzeugen oder eigenständigen Tools.

Diese Arbeit soll anhand von Praxisbeispielen den Stand der Technik erheben, eine integrierte Arbeitspraktik konzipieren und diese prototypisch in einem existierenden oder neuen Werkzeug unterstützen. Dabei sollen insbesondere die folgenden Fragen beantwortet werden:

- Welche Tests sind pro DDD-Pattern bzw. Konzept sinnvoll identifizier- und ableitbar? Sind diese generierbar als Input für BDD-Tools, z.B. über bereits existierende DSLs (DDD, BDD)?
- Welche Spezifikations- und Testdaten werden benötigt, um die Tests aussagekräftig zu machen (Bsp. z.B. für die Szenarien Normalbetrieb, Edge Cases, Fehlerfälle bzw. "Sunny Day", "Rainy Day", "Stormy Day")? Wie wird mit Optionalität sowie Mengen und anderen Datenstrukturen umgegangen? Welche Modellierungstiefe im DDD-Input ist erforderlich, um diese Daten generieren zu können?
- Wie lassen sich Test-Orakel bilden und Teile des System Under Test (SUT) dabei mocken?

Die Werkzeugplattform wird in den ersten Projektwochen zusammen mit den Anforderungen gemeinsam festgelegt (eine Option ist dabei die Erweiterung oder Nutzung von Context Mapper, z.B. über Apache Freemarker Templates).

3. Liefergegenstände und Erfolgsfaktoren

Die folgenden Liefergegenstände (Deliverables) sollen erstellt werden:

- Analysebericht zum Stand der Technik, z.B. mit den folgenden Inhalten: Recherche zu Best Practices für Unit- und Integrationstests, Kriterien für Methodenwahl und Methodenvergleich, Stärken und Schwächen der im Markt vorhandenen Werkzeuge (Bsp. Einstiegshürden und andere Herausforderungen bei der Verwendung von BDD-Tools und der Praktik Example Mapping, die bisher einer breiteren Nutzung entgegenwirken).
- Spezifikations- und Testgenerierungskonzepte "DDD-to-BDD", deren Funktionsumfang in den ersten Projektwochen in der Anforderungsanalyse gemeinsam festgelegt wird.
- Methodenintegrationsvorschlag DDD/BDD/Example Mapping, der z.B. im DPR-Activity-Format dokumentiert werden kann (https://github.com/socadk/design-practice-repository)
- Tool-Prototyp, der die vorgeschlagene Methodenintegration umsetzt, z.B. Testdaten und BDD-Toolinput (Cucumber und/oder JBehave, Gherkin DSL) aus DDD-Modellen generieren (ggfs. in miro oder im Context Mapper)
- User-Tutorial mit Beispielen
- Technischer Bericht

Die Praxisbeispiele werden a) vom Betreuer zur Verfügung gestellt, b) online recherchiert und c) selbst erarbeitet. Die Anwendungslandschaft Lakeside Mutual z.B. liegt bereits als DDD-Modell vor und wird bisher mit JUnit und curl getestet (https://github.com/Microservice-API-Patterns/LakesideMutual). BDD und Example Mapping kommen aber noch nicht zum Einsatz.

Wichtige nichtfunktionale Anforderungen, die im Projektverlauf genauer erarbeitet, konkretisiert und dann gemeinsam festgelegt werden, sind:

- Robuster, benutzerfreundlicher Umgang mit unzureichender DDD-Modellqualität und -tiefe.
- Domänen-Fachexperten ohne Erfahrung in DDD und BDD sollen nach Einarbeitung mit Hilfe von Beispielen und Tutorials innerhalb von 5 bis 10 Minuten Specification by Example bzw. Example Mapping betreiben und valide Test Cases für BDD generieren können.
- Die generierten Test-Stubs sollen eine gute Testabdeckung und eine hohe Aussagekraft aufweisen (Bsp. Normalbetrieb, Edge Cases, Fehlerfälle); der Vorteil der Testcasegenerierung gegenüber einer manuellen Erstellung und -pflege soll sichtbar werden.
- Die Konzepte und der Toolprototyp sollen erweiterbar und modular im Hinblick auf neue Notationen und Praktiken sein.
- Als Implementierungssprache soll Java verwendet werden.
- Software-Engineering-Hygieneanforderungen wie Versionskontrolle, automatisiertes Testen, CI/CD sollen adäquat und sinnvoll eingesetzt werden.
- Als Lizenz soll Apache 2 verwendet werden analog zu Context-Mapper; externe direkte Abhängigkeiten müssen entsprechend kompatibel sein.

4. Unterstützung

Die erwartete und effektiv erhaltene Unterstützung wird durch die Studierenden protokolliert.

5. Zur Durchführung

Mit dem Betreuer finden in der Regel wöchentlich Besprechungen statt (Treffen an der OST oder Telefon- bzw. Webkonferenz). Zusätzliche Besprechungen sind nach Bedarf zu veranlassen.

Alle Besprechungen, bei denen eine Vorbereitung durch den Betreuer nötig ist, sind von den Studierenden mit einer Traktandenliste vorzubereiten. Beschlüsse sind in einem Protokoll zu dokumentieren.

Für die Durchführung der Arbeit ist ein Projektplan zu erstellen. Dabei ist auf einen kontinuierlichen und sichtbaren Arbeitsfortschritt zu achten. Arbeitszeiten sind zu dokumentieren.

Die Spezifikation der Anforderungen geschieht durch die Studierenden in Absprache mit dem Betreuer. Bei Disputen entscheidet der Betreuer in Rücksprache mit den Studierenden und dem Auftraggeber über die definitiv für die Bachelorarbeit relevanten Anforderungen.

Vorstudie, Anforderungsdokumentation und Architekturdokumentation sollten im Laufe des Projektes mittels Milestones mit dem Auftraggeber und dem Betreuer in einem stabilen Zustand abgenommen werden. Zu den abgegebenen Arbeitsresultaten wird ein vorläufiges Feedback abgegeben. Eine definitive Beurteilung erfolgt auf Grund der am Abgabetermin abgelieferten Dokumentation.

Die Rechte an den Ergebnissen der Bachelorarbeit werden in einer separaten Vereinbarung definiert (Bericht öffentlich).

6. Dokumentation

Über diese Arbeit ist eine Dokumentation gemäss den Richtlinien der Abteilung Informatik zu verfassen. Die zu erstellenden Dokumente sind im Projektplan festzuhalten. Alle Dokumente sind nachzuführen, d.h. sie sollten den Stand der Arbeit bei der Abgabe in konsistenter Form dokumentieren.

Bei der Projektdokumentation und Ihrer Abgabe sind die „Allgemeine Informationen zu Studien- und Bachelorarbeiten“ sowie die „Anleitung: Dokumentation Studien- und Bachelorarbeiten“ inklusive Anhängen zu beachten.

7. Termine

Siehe HSR/OST-Webseiten. Allfällige weitere Termine sind am Sekretariat der Abteilung Informatik zu erfragen und sollten entsprechend in einem Sitzungsprotokoll dokumentiert werden.

8. Beurteilung

Eine erfolgreiche Bachelorarbeit zählt 12 ECTS-Punkte pro Studierenden. Für 1 ECTS-Punkt ist eine Arbeitsleistung von ca. 25 bis 30 Stunden budgetiert. Siehe auch Modulbeschreibung der Bachelorarbeiten, http://studien.hsr.ch/allModules/24809_M_BAI14.html.

Gesichtspunkt	Gewicht
1. Organisation, Durchführung	1/6
2. Berichte (Abstract, Management Summary, technische u. persönliche Berichte) sowie Gliederung, Darstellung und Sprache der gesamten Dokumentation.	1/6
3. Inhalt*)	3/6
4. Mündliche Prüfung zur Bachelorarbeit	1/6

*) Die Unterteilung und Gewichtung von 3. Inhalt wird im Laufe dieser Arbeit festgelegt.

Im Übrigen gelten die Bestimmungen der Abt. Informatik zur Durchführung von Bachelorarbeiten.

Rapperswil, 24. 02. 2021

Prof. Dr. Olaf Zimmermann
Institut für Software (IFS)
University of Applied Sciences of Eastern Switzerland (OST)

Abstract

Diese Bachelorarbeit beschreibt die Umsetzung und Integration eines Werkzeugs, welches Domain Driven Design (DDD) mit Behaviour Driven Development (BDD) kombiniert. Im Fokus stehen dabei die Analyse, das Vorgehen und das Lösungskonzept zur Erarbeitung dieser Arbeitsmethode. Die gängigsten Werkzeuge im Bereich DDD und BDD werden im Rahmen der Bachelorarbeit anhand von Praxisbeispielen untersucht. Ziel dieser Arbeit ist es, zu untersuchen, wie diese beiden Methoden miteinander kombiniert werden können. Der entwickelte Prototyp zeigt, dass es möglich ist basierend auf einem DDD Modell, in Form eines Context Mapper Modells, Gherkin Tests zu generieren. Somit werden die beiden Methoden mittels Toolunterstützung kombiniert. Der Prototyp generiert Tests im Bereich Assoziationen, Validierungen von Attributen und Überprüfung von DDD Patterns. Zunächst wird ein Testgenerierungskonzept entwickelt, anhand dessen die Testgenerierung umgesetzt wird. Ergänzend zum Prototyp wird eine Anleitung zur Verfügung gestellt. Diese erklärt, wie die Einbindung des Gherkin-Outputs in den Toolstack durch die automatisierte Überprüfung der Gherkin-Tests gelingt. Hierfür wird Cucumber in ein Maven oder Gradle Projekt integriert. Es wird gezeigt, dass die Einbindung der Gherkintests in ein Softwareprojekt nicht triviale Implementationsschritte beinhaltet, welche die Testautomatisierung anspruchsvoll und zeitaufwändig gestaltet. Im Ausblick wird beschrieben wie komplexere Beispielprojekte eine Entwicklung von Best Practice im Bereich DDD zu BDD vorantreiben könnten.

Management Summary

Ausgangslage

Domain Driven Design (DDD) ist eine weit verbreitete Praktik in der Softwaremodellierung. Komplementär dazu wird Behavior-Driven Development für "Specification by Example"und die systematische Spezifikation von Testcases angewendet. Wie die beiden Techniken miteinander kombiniert werden können, ist noch nicht ausreichend untersucht und dokumentiert. In dieser Arbeit soll anhand von Praxisbeispielen der Stand der Techniken erhoben und eine integrierte Arbeitsmethode konzipiert werden. Diese Arbeitsmethode soll von einem neuen oder bereits existierenden Werkzeug unterstützt werden und als Einstiegshilfe für BDD dienen.

Vorgehen, Technologie

Als erstes analysierten wir die vorhanden DDD- und BDD-Tools. Im Bereich DDD waren dies Context Mapper und Sculptor Generator, zwei Modellierungs-Frameworks. Im BDD Umfeld untersuchten wir zwei Werkzeug Unterstützungen für Java, Cucumber und JBehave. Ebenfalls recherchierten wir zu weiteren Praktiken, die im BDD-Umfeld genutzt werden wie zum Beispiel das Example Mapping. Die Recherche nach einem geeigneten Werkzeug, welches direkt für die kombinierte Arbeitsmethode verwendet werden könnte, verlief erfolglos. Deshalb haben wir uns entschieden, einen eigenen Prototyp, DDD-2-BDD, zu entwickeln. Dieser nutzt Context Mapper Modelle als Input und generiert als Ouput Testszenarien im Gherkin-Syntax. BDD-Tools arbeiten mit Textdokumenten, die durch Schlüsselwörter zu ausführbaren Spezifikationen werden. Gherkin wird von vielen BDD-Frameworks unterstützt. Für den Context Mapper haben wir uns entschieden, weil das Projektteam bereits Erfahrungen damit gesammelt hatte, eine bereits erprobte Java Library zur Verarbeitung des Models zur Verfügung steht und es die Modellierung umfangreicher Modells ermöglicht. Der Prototyp ist als Webapplikation umgesetzt mit Spring Boot und Thymeleaf. Abbildung 0.1 zeigt mit einem Screenshot der Applikation auf, wie die Selektion der zu generierenden Tests funktioniert und in welcher Form die generierten Tests zur Verfügung gestellt werden.

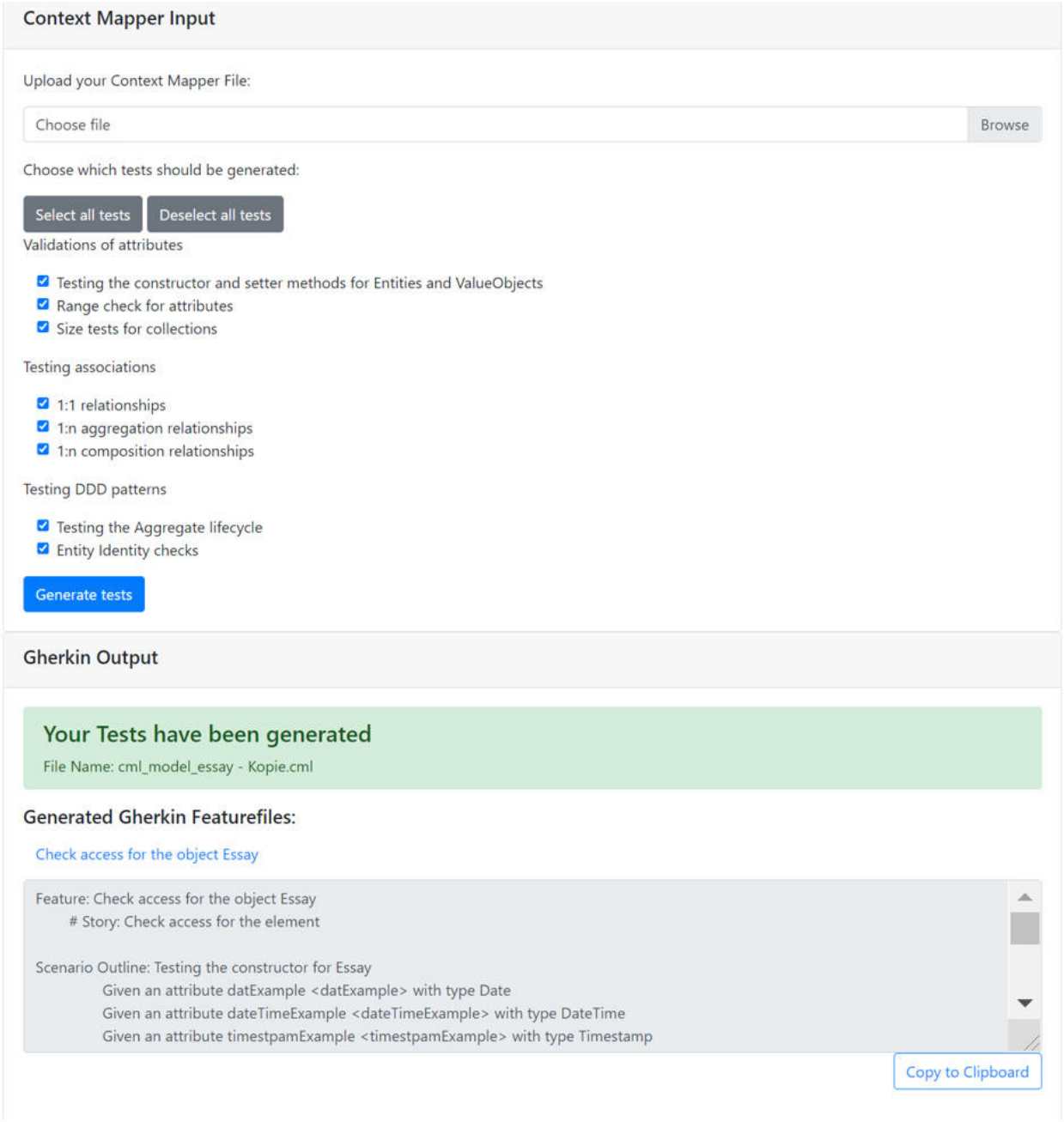

Abbildung 0.1: Screenshot User Interface

Ergebnisse

Zusätzlich zum DDD-2-BDD Toolprototypen entstand eine Dokumentation der Testgeneration. Es werden in drei verschiedenen Bereichen Testgenerierungen angeboten: Validierungen von Attributen, Testen von Beziehungen und Überprüfung von DDD-Patterns. Der Ansatz, aus dem Domänenmodel Gherkin-Features zu generieren, bietet den Vorteil, dass bereits modellierte Informationen wiederverwendet werden und so eine Zeitersparnis erreicht wird. Ein Nachteil dieses Ansatzes ist es jedoch, dass ein hoher Detailgrad im Modell-Input benötigt wird, um sinnvolle Tests generieren zu können. Zusätzlich zu den Generierungen bietet das Tool zwei Tutorials an. Die Tutorials erklären mithilfe von Beispielen, wie der Output generiert wird und wie er mit Cucumber verwendet werden kann, um die Tests zu automatisieren. Es wird beschrieben, wie ein Projekt aufgesetzt wird mit Maven oder Gradle, wie die Gherkin Files integriert werden und wie die sogenannten Step Definitions in Cucumber definiert werden können. Zusätzlich werden die Beispiele als Maven- und Gradle-Projekte zur Verfügung gestellt. In Abbildung 0.2 ist das Zusammenspiel zwischen Context Mapper, DDD-2-BDD, Gherkin Files, Cucumber Step Definitions und Java Code illustriert.

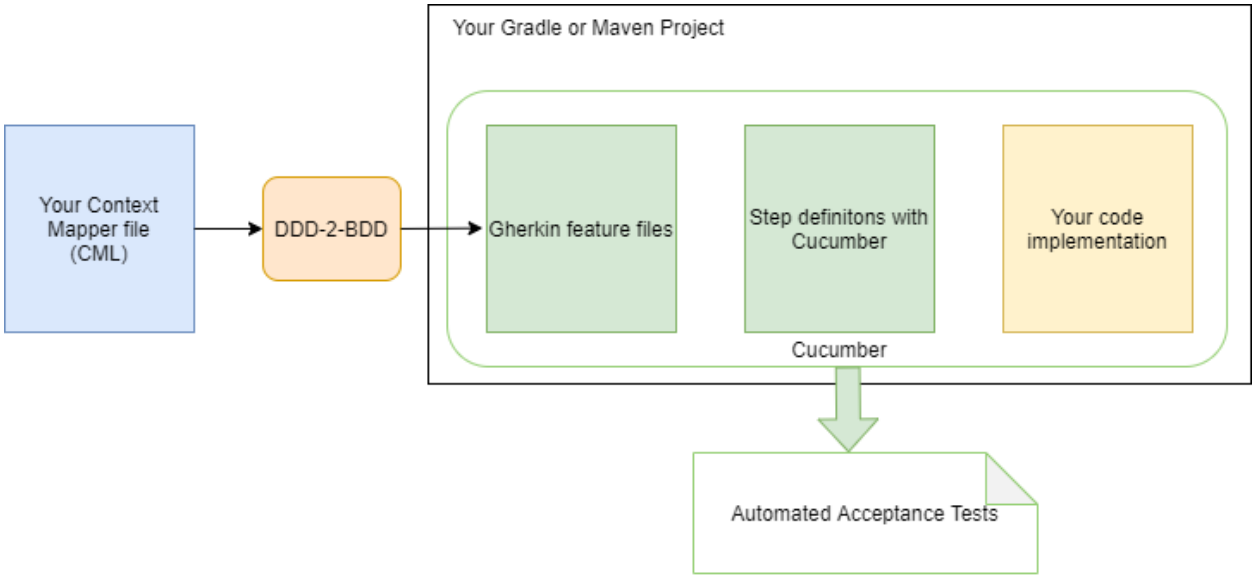

Abbildung 0.2: Überblick DDD-2-BDD Arbeitspraktik

Glossar

DDD	Domain Driven Design
CML	Context Mapper Language
BDD	Behaviour Driven Development
Gherkin	DSL für ausführbare Spezifikationen
Cucumber	Automatisierungstool für Gherkin Feature Files
JBehave	Automatisierungstool für Gherkin Feature Files
Example Mapping	Workshopform für die Spezifikation per Example
DDD-2-BDD	Toolprototyp für Testgenerierung

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

1.1 Ausgangslage

Als Ausgangslage dienen die Domänen Domain Driven Design (DDD) und Behaviour Driven Development (BDD). DDD ist eine Methode, um komplexe Domänen zu modellieren. Es basiert auf einem fundierten Verständnis der Domäne und der Entwicklung einer Ubiquitous Language. DDD definiert einen Katalog von Patterns[1]. BDD dient dazu die Lücke zwischen technischen und geschäftlichen Personen zu schliessen. Im BDD werden User Stories mithilfe von Beispielen genauer spezifiziert und dokumentiert, dies soll das Verständnis der beschriebenen Domäne verstärken [2]. In der Praxis werden DDD und BDD bereits kombiniert angewendet[3][4][5]. Es gibt jedoch keine Arbeitsmethode, die dies beschreibt. Der Einstieg in BDD ist sowohl durch die geringe Anzahl an öffentlich zugänglichen Tutorials und Beispielen als auch durch fehlende Best Practises erschwert.

1.2 Problembeschreibung

Das Ziel dieser Arbeit ist es, im Bereich DDD und BDD den Stand der Techniken zu erheben und die beiden Domänen DDD und BDD miteinander zu kombinieren. Dabei soll eine kombinierte Arbeitsmethode konzipiert werden. Die Arbeitsmethode soll die Einstiegshürde zur Arbeit mit BDD senken für Personen, die mit DDD bereits vertraut sind. Diese Arbeitsmethode soll zusätzlich von einem neuen oder mit bereits bestehende Werkzeuge unterstützt werden. Anhand eines DDD Modells sollen Testfälle in einer Form generiert werden, die mit BDD Tools weiterverwendet werden können. Die Schwierigkeit zeigt sich darin, dass anhand eines statischen Modells dynamische Tests generiert werden sollen. Die technische Machbarkeit ist dadurch gegeben, dass bereits ein automatischer Testgenerator entwickelt wurde, welcher einen DDD-Input verarbeiten kann. Dieser Generator ist jedoch nicht verfügbar und wurde ausschliesslich für die Machbarkeitsanalyse entwickelt[6]. Eine weitere Herausforderung ist, dass die Tools im Bereich von BDD nur minimal dokumentiert sind und kaum komplexe Beispiele vorhanden sind. In der Dokumentation von Cucumber sind nur wenige und einfache Beispiele zu den einzelnen Cucumber Expressions[1] vorhanden.

1 https://cucumber.io/docs/cucumber/cucumber-expressions/

1.3 Übersicht

Der Bericht ist in sechs Teile gegliedert. Zu Beginn wird ein Einblick in die Recherche gegeben. Danach wird das Lösungskonzept erläutert und die Umsetzung beschrieben. Darauf folgen die Ergebnisse und die Diskussion der Ergebnisse. Zum Schluss wird ein Ausblick gegeben, wie die entwickelte Arbeitsmethode weiterentwickelt und verbessert werden kann. Die Software-Engineering Dokumente sind im Anhang zu finden, sowie ein ausführlicher Recherchebericht und die Benutzer Tutorials.

2 Analyse

Um den Stand der Technik zu erheben, haben wir diverse Tools im Bereich DDD und BDD mit eigenen Beispielen getestet und nach wissenschaftlichen Arbeiten dazu recherchiert. Im folgenden Abschnitt sind die elementarsten Ergebnisse der Analyse zusammengetragen. Der detaillierte Analysebericht ist im Anhang A zu finden.

2.1 Domain Driven Design (DDD)

Die Domain Driven Design Modellierung wird von zwei DSL Tools unterstützt, zum einen durch den Context Mapper[1] und zum anderen durch den Sculptor Generator[2]. Der Context Mapper integriert die Tactic DDD Syntax des Sculptor Generators und wurde an den OST Ostschweizer Fachhochulen entwickelt[7]. Neben der Verwendung dieser streng formatierten Tools kann DDD auch mit Onlinewhiteboards oder von Hand frei modelliert werden.

2.2 Behaviour Driven Development (BDD)

Die Behaviour Driven Development Toolingumgebung kann in zwei Bereiche eingeteilt werden. Den einen Bereich bilden die BDD Tools, die die automatisierte Ausführung von Testszenarien ermöglichen. Zu diesen Tools gehören Cucumber und JBehave, welche wir detailliert untersucht haben. Im anderen Bereich gibt es Tools, welche Workshopformate, wie Example Mapping und Feature Mapping, unterstützen. Diese Workshopformate werden genutzt, um die Akzeptanzkriterien der User Stories beziehungsweise der zu entwickelnden Software zu definieren.

2.3 DDD in Kombination mit BDD

Die kombinierte Verwendung von DDD und BDD wird aktuell von keinem Tool unterstützt. Die Recherche indiziert; dass es noch keine einheitliche Kombination der beiden Methoden vorhanden ist.

1 https://contextmapper.org/
2 http://sculptorgenerator.org/

2.4 Testing

Das Testen von Software kann in drei verschiedene Granularitätsbereiche aufgeteilt werden, wie in Abbildung 2.1 ersichtlich ist. Im untersten Bereich sind die Unittests, beziehungsweise Whitebox Tests, welche einzelne Komponenten isoliert testen. In der darüber liegenden Schicht sind die Service Tests angesiedelt. Diese werden je nach Definition auch Integration-Tests genannt. Integration-Tests überprüfen, ob mehrere Komponenten von einem System richtig miteinander interagieren. Über den Integrationstests liegen die UI-Tests oder auch End-to-End Tests. End-to-End Tests stellen sicher, dass das System als ganzes reibungslos funktioniert. Die Integrations, sowie auch End-to-End Tests sind Blackbox Tests[8].

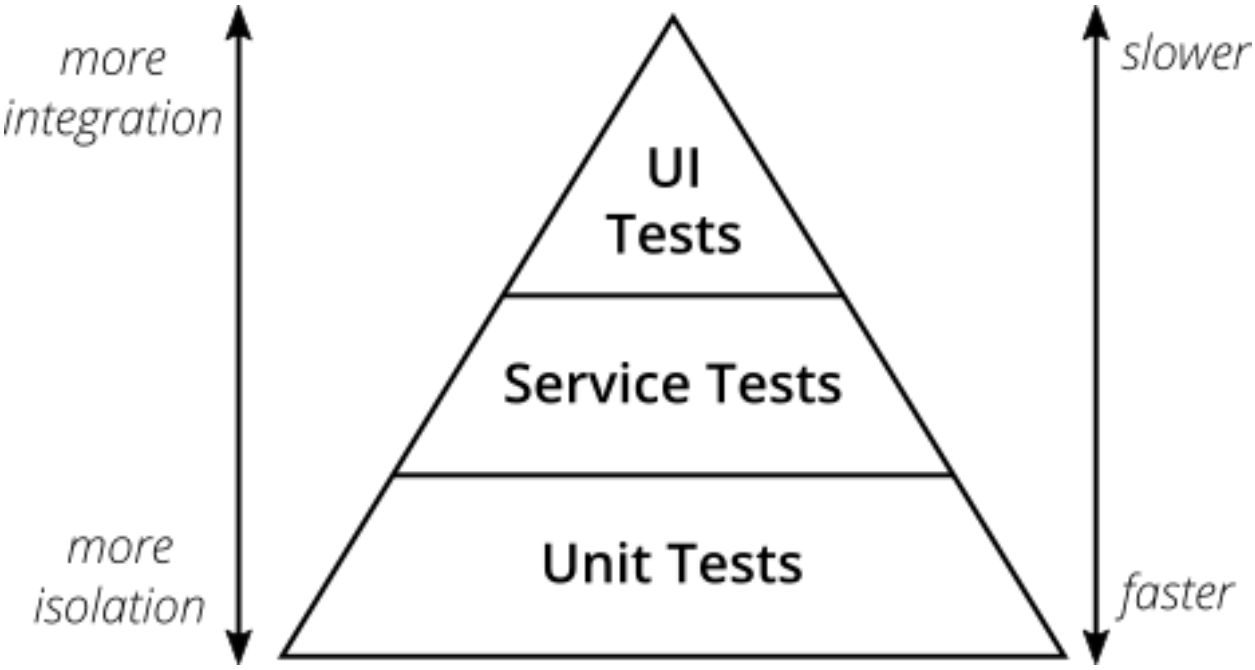

Abbildung 2.1: Testpyramide von Martin Fowler

BDD Tests können innerhalb aller drei Testbereiche sein. Sie sind jedoch für Unittest ungeeignet, da der Aufwand für das Resultat zu hoch ist. BDD Tests punkten im Bereich der Integration und End-to-End Tests, da die BDD Testspezifikationen, das Verhalten prägnant und ausdrucksstark formulieren[9].

3 Lösungskonzept

In den folgenden Abschnitten soll erläutert werden, welche Werkzeuge und Tools für die Generierung von Tests zum Einsatz kommen und wie die neue Arbeitsmethode umgesetzt wird.

3.1 Toolunterstützung für kombinierte Anwendung von DDD und BDD

Die Recherche nach einem geeigneten Werkzeug oder Tool, welches wir direkt für unsere Arbeitsmethode verwenden können, verlief erfolglos. Aus diesem Grund haben wir uns entschieden, einen eigenen Prototyp, DDD-2-BDD, zu entwickeln. Dieser verarbeitet einen DDD-Input und generiert einen dazugehörigen BDD-Output. Als DDD-Input wird ein Context Mapper (CML) File verwendet. Der BDD-Output wird in Form von Gherkin Feature Files generiert. Diese Gherkin Feature Files beinhalten Testszenarien in einem klar definierten Syntax. Die Entscheidungsfindung bezüglich des In- und Outputs sind im Anhang D.2 zu finden. Die generierten Gherkin Feature Files können mit einem Automatisierungstool, wie Cucumber oder JBehave, weiterverarbeitet werden.

Anforderungen

DDD-2-BDD soll anhand des Inputs automatisch Tests generieren. Ein Ziel ist, eine hohe Testabdeckung zu erreichen, sodass der Nutzer neben dem Vorteil der Zeitersparnis, auch eine gewisse Sicherheit im Vergleich zur manuellen Testerstellung hat. Zudem soll der Prototyp einfach zu nutzen und die Handhabung soll schnell erlernbar sein. Die detaillierten Anforderungen inklusive Use Cases sind im Anhang B zu finden. Des Weiteren haben sich drei Hauptanwender des Prototypens herauskristallisiert, diese sind in den Personas B.3 aufgeführt.

3.2 Testgenerierung

Gherkin benutzt Schlüsselwörter, um die Akzeptanzkriterien zu strukturieren und ausführbar zu machen. Die wichtigsten Schlüsselwörter sind Given, When und Then [10]. Um die erstreb-

ten Inhalte der Gherkin Feature Files des Outputs zu bestimmen, wurden Example Mapping Workshops für die einzelnen Use Cases durchgeführt. Diese fanden teilweise auch in Zusammenarbeit mit dem Betreuer statt. Die detaillierten Texte der einzelnen Statements wurden von einem der beiden Teammitglieder spezifiziert und jeweils vom anderen Teammitglied bei Bedarf angepasst und mit einem Beispiel überprüft. Es wurde besonders darauf geachtet, dass die Texte kurz, präzise und aussagekräftig sind. Zusätzlich musste bedacht werden, dass die einzelnen Statements unique sein sollten, ausser sie sind semantisch gleich, dann muss genau der gleiche String verwendet werden, damit die Step Defintions in der Verwendung mit Cucumber nicht doppelt programmiert werden müssen.

Kombiniertes Testen

Pro Szenario soll nur ein Fall getestet werden. Die dazugehörige Szenariobeschreibung soll möglichst eindeutig sein. Pro Gherkin Feature File wird nur eine Komponente getestet. Dies soll den Benutzern den Einstieg ins BDD erleichtern und vereinfacht das finden der Fehler beim Fehlschlagen der Testfälle. Der Entschluss gegen das kombinierte Testen resultiert aus der dadurch massiv erhöhten Komplexität des Prototyps. Der Entscheid gegen kombinatorisches Testen wurde untermauert durch die Schwierigkeit des Abschätzens, wie die einzelnen Komponenten anhand der Namen und Struktur im Context Mapper Model kombiniert werden können, ohne den Ursprung der Daten und den semantischen Sinn dahinter zu kennen.

3.3 Arbeitsmethode

Die Arbeitsmethode wird in Form von Tutorials zur Verfügung gestellt. Anhand einer Beispielapplikation wird der Prozess dargestellt, wie mit einem CML File die Testszenarien generiert werden und der Output in ein Maven oder Gradle Projekt eingebunden werden kann. Zudem wird gezeigt, wie die Überprüfung der in den Gherkin Files definierten Testszenarien mit Cucumber automatisiert werden kann. Der komplette Prozess des Tutorials ist in der Abbildung 3.1 zu sehen. Für die Überprüfung der Tutorials wird ein Usabilitytest durchgeführt.

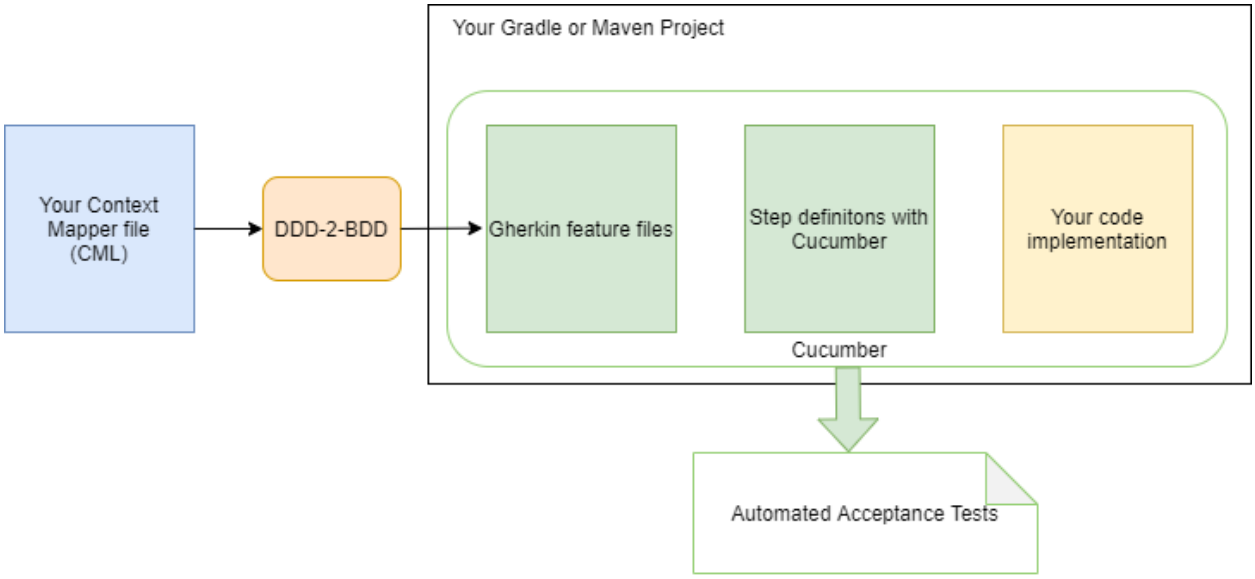

Abbildung 3.1: Überblick Arbeitsmethode

3.4 Example Mapping, Feature Mapping

Auf die Verwendung von Example Mapping und Feature Mapping wird bewusst verzichtet, weil bei diesen Techniken der Dialog im Fokus steht. Im Softwareprozess den Dialog anhand eines Tools oder Werkzeugs zu unterstützten, ist schwer umsetzbar und in diesem Fall nur bedingt sinnvoll. Da solche Tools möglichst frei in der Gestaltung sein sollten um die Benutzer nicht unnötig einzuschränken. Dadurch wäre es sehr aufwändig solche Tools in einen Toolstack einzubinden, welches den Rahmen dieser Arbeit sprengen würde. In der Studienarbeit miro2cml[1], der Vorgängerarbeit dieser Bachelorarbeit wurde ein Tool entwickelt das sich dieser Problematik widmet.

1 https://eprints.ost.ch/id/eprint/935/

4 Umsetzung

Im folgenden Abschnitt wird ein Einblick in die Software Architektur und das Software Design gegeben. Die ausführliche Dokumentation der eingesetzten Technologien und der Software Architektur und des Designs befindet sich im Anhang D. Die nächsten Abschnitte beschreiben, wie die Testgenerierung umgesetzt wird.

4.1 Einblick Software Architektur und Design

Die Abbildung 4.1 zeigt das Context-Diagram des DDD-2-BDD Prototyps. Der blaue Pfeil stellt den Benutzerinput dar, welcher aus einem CML File und der Auswahl der zu generierenden Tests besteht. Die DDD-2-BDD Applikation liest das erhaltene CML-File mithilfe der Context Mappers DSL[1] aus. Der grüne Pfeil stellt den Output, die Testszenarien in Form von Gherkin Feature Files, welche dem Benutzer basierend auf dem Input generiert und zurückgegeben werden, dar.

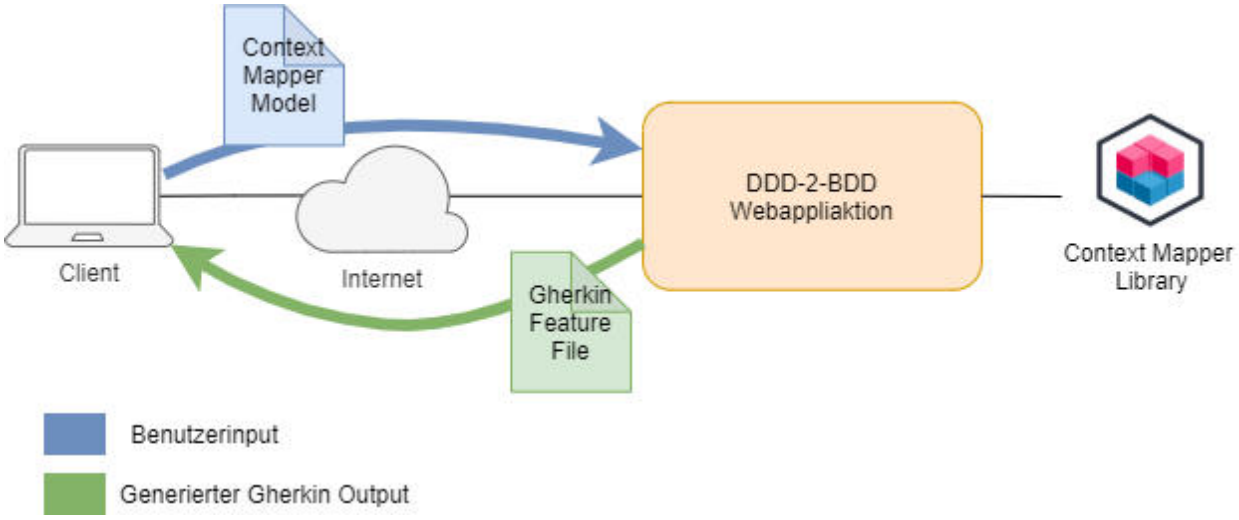

Abbildung 4.1: Context Diagramm

1 https://www.javadoc.io/doc/org.contextmapper/context-mapper-dsl/6.4.0/index.html

4.2 Umsetzung der Testgenerierung

Die zu generierenden Tests gemäss den Use Cases **??** können in drei Hauptgruppen mit unterschiedlichen Testzwecken unterteilt werden. Diese werden in den folgenden Abschnitten genauer beschrieben. Wie die Tests generiert und welche Anforderungen an den Input gestellt werden ist als Teil des Benutzertutorials im Anhang G zu finden.

4.2.1 Validierung von Attributeigenschaften

Bei der Validierung der Attribute geht es darum zu überprüfen, dass die Implementation dem entworfenem Modell entspricht. Der Fokus liegt hierbei auf den Attributen von Entities und ValueObjects. Es werden drei Features in diesem Bereich unterstützt, die näher beschrieben werden sollen.

4.2.2 Testing the constructor and setter methods

Die folgenden vier Use Cases werden mit diesem Feature abgedeckt:

As a Software Developer

- I want to assert that an immutable Attribute is immutable
- I want to assert that a changeable Attribute is changeable
- I want to assert that a required Attribute is defined in the constructor
- I want to assert that a nullable Attribute is nullable

so that I can ensure wheter the attributes have been implemented correctly.

Um die Attribute auf “immutabililty“, “changeablility“, “requireabillity“ und “nulllability“ zu testen, werden die Konstruktoren und die Setter Methoden überprüft. Die Testgenerierung basiert auf den Eigenschaften des CML Models, weswegen diese im folgenden Abschnitt kurz erläutert werden. Attribute in einer Entity sind in CML per Default “changeable“, mit dem Schlüsselwort “!changeable“ können sie “immutable“ gesetzt werden. Wenn Attribute das Schlüsselwort “required“ haben müssen sie vom Konstruktor initialisiert werden. Die Attribute in einem ValueObject sind per Default “immutable“ und werden vom Konstruktor gesetzt. Ein ValueObject kann jedoch auch “!immutable“ gesetzt werden, dann werden die Attributeigenschaften analog zur Entity gesetzt[11]. Im Konstruktor müssen alle Elemente enthalten sein, die entweder als “immutable“ oder “required“ definiert sind. Das Modell erwartet Setter Methoden für Attribute die “changeable“ sind. Der Konstruktortest arbeitet mit Defaultwerten bei Datentypen, die von der Cucumber JVM unterstützt werden. Bei den

übrigen Datentypen werden Platzhaltertexte verwendet. Für Testszenarien zur Überprüfung von Konstruktoren kommen Data Tables zum Einsatz, die von den Benutzer*innen mit eigenen Daten befüllt werden können. In den Testszenarien zur Überprüfung der Setter Methoden werden bei jedem Element, welches "changeable" ist überprüft, ob die Werte richtig gesetzt werden. Zusätzlich dazu wird überprüft, ob das Setzen von "null" funktioniert, wenn das Attribute "nullable" ist und wenn das Attribute nicht "nullable" ist, ob dann eine Exception geworfen wird. Bei Setter Methoden wird ebenfalls mit Defaultwerten gearbeitet. Das Setzen von Defaultwerten weist auf eine Schwäche dieses Ansatzes hin. So muss der Benutzer die Testszenarien noch ergänzen, um einen semantisch korrekten und sinnvollen Test zu erhalten. Wie Usability Tests gezeigt haben, ist es nicht sinnvoll randomisierte Werte anstelle von statischen Defaultwerten zu verwenden. In gewissen Fällen müssen die Werte in den Testszenarien trotzdem manuell angepasst werden, was zu Verwirrung auf Seiten der Benutzer*innen führen kann. Des Weiteren fehlt das Testen von nicht vorhandenen Setter Methoden bei Attributen die "immutable" sind, dies führt zur Unvollständigkeit der Tests. Da eine Implementation in den Step Definitions, welche die Abwesenheit einer Setter Methode überprüft nicht möglich ist, werden für diesen Fall keine Tests generiert. Die Benutzer*innen haben die Möglichkeit anhand der Setter Tests manuell überprüfen, welche Elemente keine Setter haben sollen. Je nach Datentyp und Programmiersprache ist es nicht möglich, gewisse Werte auf null zu setzten, da dies bereits während der Kompilierung einen Fehler ergibt. Solche Tests werden jedoch trotzdem generiert im Wissen, das die Benutzer*innen auch nach der Generierung der Tests, diese weiter selektieren und verwerfen können. Mit einem Workaround können diese Tests trotzdem durchgeführt werden, siehe Benutzertutorial G.

Listing 4.1: Beispiel Gherkin File für Konstruktor und Setter

```
Scenario Outline: Testing the constructor for Essay
 Given an attribute identifier <identifier> with type EssayId
 When the constructor Essay is called with the values: <title>,
       <essayBody>, <identifier>
 Then the attribute title in Essay is equals <title>
 Then the attribute essayBody in Essay is equals <essayBody>
 Then the attribute identifier in Essay is equals <identifier>
Examples:
        | title | essayBody | identifier |
        | "This is a String" | "This is a String" | "EssayId" |
        | "This is a String" | "This is a String" | "EssayId" |
        | "This is a String" | "This is a String" | "EssayId" |
 #TODO replace the types in the table with a corresponding value

Scenario: Testing the setter method for object Essay and attribute title
 Given an attribute title with type String and value "This is a String"
 When title is set to "This is a String"
 Then the attribute title in Essay is equals "This is a String"
 #TODO replace "This is a String" with a corresponding value for title
```

```
Scenario: Testing the setter method for object Essay and attribute title
        with null
 Given an attribute title with type String and value "This is a String"
 When title is set to null
 Then an exception illegal argument is thrown in Essay
 #TODO replace "This is a String" with a corresponding value for title
```

4.2.3 Range check für Attribute

Die folgenden Use Cases werden mit diesem Feature abgedeckt:

As a Software Developer

- I want to ensure that an Attribute on or below its defined maximum
- I want to ensure that an Attribute on or above its defined minimum
- I want to ensure that an Attribute can only be set within its defined range

so that I can ensure wheter the attributes have been implemented correctly.

Bei diesem Feature wird das Schlüsselwort “range“ verwendet. Die spezifischen Notationen sind im Tutorial zu finden G.3. Um ein Attribut zu überprüfen, dass sich in einem bestimmten Wertebereich befinden soll, welcher sowohl von einem Minimum als auch einem Maximum begrenzt ist, werden fünf Arten von Testfällen generiert. Dabei handelt es sich um die Testfälle “Boundary Data In“ und “Boundary Data Out“ bei der unteren Wertegrenze, “Boundary Data In“ und “Boundary Data Out“ bei der oberen Wertegrenze und den Fall “normal Data“ welcher repräsentativ für Werte steht, die sich innerhalb des erlaubten Wertebereichs befinden. Die Abbildung 4.2 visualisiert die zu testenden Fälle an einem einfachen Beispiel.

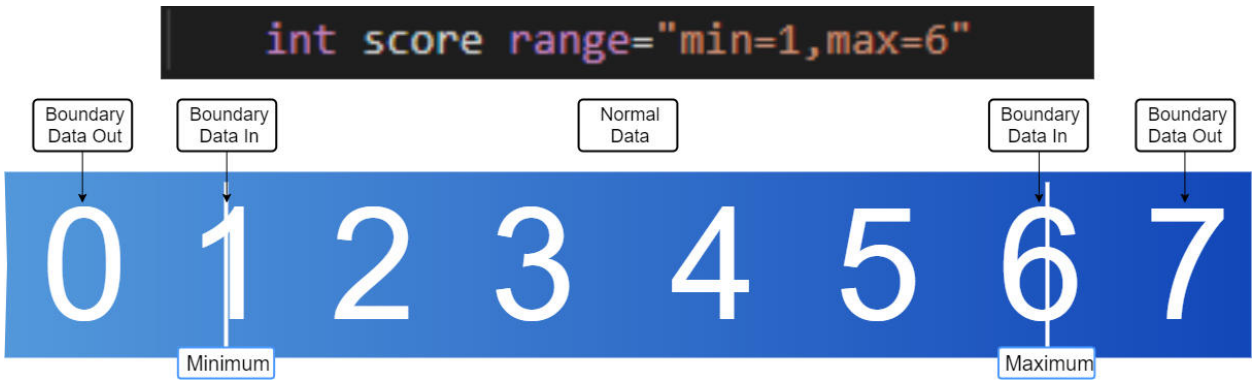

Abbildung 4.2: Visualisierung der Testfälle in einem Range Check

Wird eine Range lediglich mit einem Minimum, aber ohne Maximum oder umgekehrt angegeben, entfallen die Testfälle “Boundary Data In“ und “Boundary Data Out“ für die entsprechende Wertegrenze. Um die Testfälle zu prüfen, werden die generierten Daten mittels Setter Methode gesetzt und nachfolgend geprüft, ob die einzelnen Werte erfolgreich gesetzt wurden. Für die Testdaten, die sich ausserhalb des erlaubten Bereichs befinden, wird getestet ob eine IllegalArgumentException geworfen wird.

4.2.4 Grössentests für Collections

Der folgende Use Case wird mit diesem Feature abgedeckt: “As a Software Developer I want to test if a Collection cannot grow above its definded SizeMaximum so that I can ensure wheter the size contraint has been implemented correctly“. Thematisch würden hier auch die Überprüfung eines Mindestgrössenconstraints und eines Rangeconstraints dazuzugehören, auf die Umsetzung dieser Use Cases wurde jedoch aufgrund einer Kosten-Nutzen-Analyse bewusst verzichtet. Die Size Tests werden mit analogen Daten zum Range check Test umgesetzt. Bei den Tests wird eine Collection mit einer bestimmten Grösse erwartet und je nach Testfall wird ein Element der Collection hinzugefügt oder entfernt. Im nächsten Schritt wird getestet, ob die Grösse dem erwarteten Wert entspricht oder, ob die erwartete Exception geworfen wird.

4.2.5 Assoziationen

Bei den Überprüfungen der Assoziationen liegt der Fokus auf dem Löschen und Verändern von Elementen und wie sich dies auf die unterschiedlichen Assoziationen auswirkt. Jedes Feature deckt jeweils die notwendigen Funktionalitäten für je einen Use Case ab.

4.2.5.1 1:1 Aggregation zwischen Entity und ValueObject

Der folgende Use Case wird mit diesem Feature abgedeckt: “As a lead developer I want to test an 1:1 Aggregation so that I can ensure whether the relationship has been implemented correctly“. Um die Referenzen von einem Entity zu einem ValueObject zu testen, wird getestet, ob beim Löschen des Entitys, das dazugehörige ValueObject auch gelöscht wird. Zudem wird überprüft, ob beim Ändern des ValueObjects, das alte ValueObject gelöscht und die neue Referenz korrekt gesetzt wurde.

4.2.5.2 1:n Aggregation

Der folgende Use Case wird mit diesem Feature abgedeckt: “As a lead developer I want to validate the behaviour of an Aggregation for 1:n relationships so that I can ensure whether the relationship have been implemented correctly“. Es werden Referenzen überprüft, die als

Aggregation definiert sind. Eine Aggregation ist eine Referenz welche indiziert, dass das referenzierte Element zum referenzierenden Element gehört, im Sinne einer "has-a" Beziehung. Eine Aggregation wird mit dem Schlüsselwort *cascade="persists"* definiert [12]. Bei der Aggregation wird überprüft, ob beim Löschen des Basiselements die referenzierten Elemente nicht gelöscht werden. Zusätzlich wird überprüft, ob beim Löschen von einem Element, welches referenziert wurde, dieses aus der Liste der referenzierten Elemente gelöscht wird.

4.2.5.3 1:n Composition

Der folgende Use Case wird mit diesem Feature abgedeckt: "As a lead developer I want to validate the behaviour of a Composition for 1:n relationships so that I can ensure whether the relationship have been implemented correctly". Es werden Referenzen überprüft, die als Komposition definiert sind. Eine Komposition ist ein Spezialfall der Aggregation, bei der das referenzierte Objekt in seiner Existenz vom referenzierenden Objekt abhängig ist. Eine Komposition wird in CML mit dem Schlüsselwort *cascade="persists,remove"* definiert[12]. Bei der Komposition wird überprüft, ob beim Löschen des Basiselements, die von ihm referenzierten Elemente auch gelöscht werden. Zusätzlich wird überprüft, ob beim Löschen von einem Element, welches referenziert wurde, dieses aus der Liste der referenzierten Elemente gelöscht wird.

4.2.6 DDD Patterns

Bei den DDD Patterns liegt das Überprüfen der definierten Eigenschaften gemäss den Patterns im Vordergrund.

4.2.6.1 Entity Identity check

Der folgende Use Case wird mit diesem Feature abgedeckt: "As a domain expert I want to check an Entity for Identity so that I can check whether the DDD pattern has been applied correctly". Eines der Haupteigenschaften, die das Entity Pattern aufweist, ist die Identität[13]. Die Identität wird anhand von vier Tests überprüft:

- Testen, ob eine ID vorhanden ist
- Überprüfen, ob zwei Entities mit unterschiedlichen Idenitites unterschiedlich sind
- Überprüfen, ob zwei Entities mit den gleichen Idenitites gleich sind
- Testen, ob zwei Identities mit der gleichen ID auch die gleichen Attributwerte haben

4.2.6.2 Überprüfung des Aggregate Lifecycles

Der folgende Use Case wird mit diesem Feature abgedeckt: “As a domain expert I want to test the Lifecycle of an Aggregate so that I can check whether the DDD pattern has been applied correctly“. Eine weitere Eigenschaft von einer Entity ist, dass es einen Lifecycle aufweist[13]. Der Lifecycle von einem Entity kann in CML mit dem *AggregateLifecycle* modelliert werden. Es kann somit nur der Lifecycle des *RootAggregates* dargestellt werden. Die verschiedene Zustände werden mit einem Enum definiert. Die Zustände können mit Methoden verändert werden, die im CML mit dem Schlüsselwort *write* versehen wurden. Diese Methoden haben entweder keinen, einen oder mehrere Eingangszustände und einen Endzustand, wobei mehrere Alternativen für diesen Endzustand definiert werden können[14]. Um den Lifecycle zu testen, wird bei den *write* Methoden getestet, ob sie ausgeführt werden können, wenn der richtige Ausgangszustand vorhanden ist. Zudem wird nach der Ausführung der Methode überprüft, ob der Zustand richtig verändert wurde. Diese Tests sind in der Abbildung 4.3 als grüne Pfeile eingezeichnet. Die Abbildung 4.3 stellt eines State Diagramm eines Aggregate Lifecycles dar. Ausserdem wird überprüft, ob bei einem falschen Ausgangszustand die Methode eine IllegalStateException wirft. Diese Tests sind mit den schwarzen Pfeilen in der Abbildung 4.3 illustriert. Falls der Methode kein Ausgangszustand zugewiesen wurde, wird nur getestet, ob der Zustand durch die Ausführung richtig gesetzt wird.

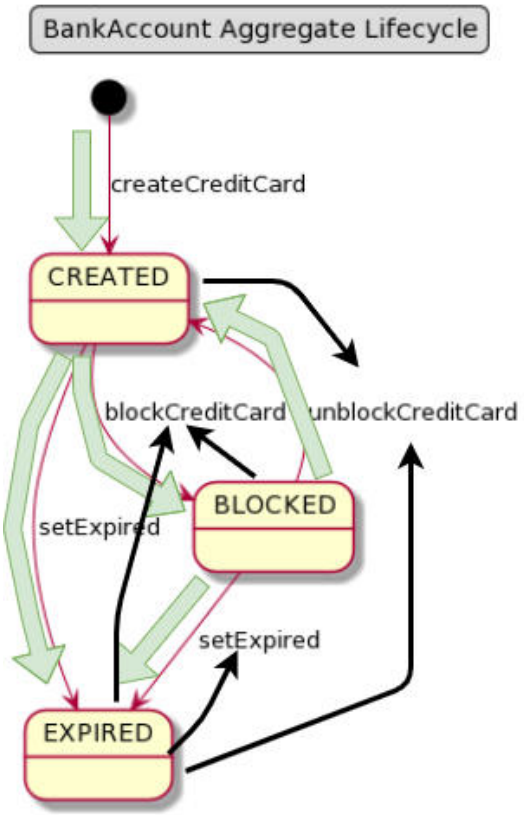

Abbildung 4.3: Aggregate Lifecycle State Diagramm Beispiel

4.2.7 Schwierigkeiten

Die einzelnen Gherkin-Statements müssen innerhalb von einem Projekt einzigartig sein, weil es nicht möglich ist, zwei gleiche Step Definitions in einem Projekt zu definieren[10]. Diese Schwierigkeit wurde in der Definition der Gherkin Texte miteinbezogen. Mit der genauen Definition, welche Elemente beispielsweise eine Exception werfen, konnte diese Vorgabe meistens eingehalten werden. Eine vollständige Behebung des Problems lies sich mit einer Anreicherung durch Referenzen, in den einzelnen Steps der Testszenarien, erreichen.

5 Ergebnisse & Diskussion

5.1 Ergebnisse

Mit dem Prototyp kann gezeigt werden, dass die Testgenerierung in Gherkin Feature Files basierend auf einem CML Modell möglich ist. Je genauer das CML Modell definiert ist, desto mehr Tests können generiert werden. Der Prototyp setzt 9 Features um, die 13 von 24 Use Cases abdecken. In der Abbildung 5.1 ist mit einer Tabelle dargestellt, welche DDD Patterns in den Features verwendet werden.

Use Cases DDD Patterns	Konstruktoren und Settermethoden	Range check int	Collection maximum	1:1 Assoziation	1:n Aggregation	1:n Komposition	Entity Idenity	Aggregate Lifecycle
Entity	x	x		x	x	x	x	
ValueObject	x	x		(x)	(x)	(x)		
Aggregate								x
AggregateRoot Entity	x	x		x	x	x	x	x
BoundedContext								
Domain Events								
Repositories								
Factories								
(Attributes)	x	x	x					
(Methods)	x							x
(References)			x	x	x	x		

Abbildung 5.1: Verwendung DDD Patterns in Feautures, x wird vom Feauture verwendet, (x) wird indirekt verwendet

Im Listing 5.1 ist ein Beispiel von einem CML Inputfile, dass eine Kreditkarten Domäne modelliert. Werden mit dem DDD-2-BDD Tool die Tests für den Aggregate Lifecycle generiert, dann ergibt sich das Feature File, welches im Listing 5.2 zu sehen ist. Das Aggregate Lifecycle Feature ist das komplexeste, da es alle zustandssverändernden Operationen in allen möglichen Kombinationen in einem Test abdeckt. Darin liegt auch der grosse Nutzen dieses Features, mit einem Klick werden für alle Varianten Testfälle generiert, die ansonsten aufwändig auszudenken wären, ein Prozess, der sehr fehleranfällig ist.

Listing 5.1: Kreditkarten Beispiel CML Model

```
BoundedContext ExampleDomain{
    Aggregate BankAccount{
        Entity CreditCard{
            aggregateRoot
            - CreditCardId  id
            BigInteger number required
            int cvCode required
            - @Person creditCardOwner required
            Date expiryDate required
            BigDecimal credit

        }
        ValueObject CreditCardId{
            String id key
        }
        Entity Person{
            String name required
            String surname required
            String address
            - List<@CreditCard> creditCards size="2"
        }
        enum States{
            aggregateLifecycle
            CREATED, BLOCKED, EXPIRED
        }
        Service Services{
            boolean createCreditCard(@Person person, Date expiryDate,
             BigInteger number, int cvCode)
                : write[ -> CREATED];
            boolean blockCreditCard(@CreditCard creditCard)
                                : write [ CREATED -> BLOCKED ];
            boolean unblockCreditCard(@CreditCard creditCard)
                                : write [ BLOCKED -> CREATED];
            void setExpired(@CreditCard creditCard)
                                : write [ CREATED, BLOCKED -> EXPIRED];
        }
    }
}
```

Listing 5.2: Generierter Gherkin Output des Kreditkarten Beispiels

```
Feature: Testing the aggregate lifecycle operations for BankAccount
        # Story: testing the lifecycle operations

Scenario: Test the aggregate state with the operation
            createCreditCard and state "not set"
                When the operation createCreditCard is executed
                Then the aggregate state change to CREATED

Scenario: Test the aggregate state with the operation
            blockCreditCard and state CREATED
                Given an aggregate BankAccount with the state CREATED
                When the operation blockCreditCard is executed
                Then the aggregate state change to BLOCKED

Scenario: Test the aggregate state with the operation
            blockCreditCard and state BLOCKED
                Given an aggregate BankAccount with the state BLOCKED
                When the operation blockCreditCard is executed
                Then a illegal state exception is thrown

Scenario: Test the aggregate state with the operation
            blockCreditCard and state EXPIRED
                Given an aggregate BankAccount with the state EXPIRED
                When the operation blockCreditCard is executed
                Then a illegal state exception is thrown

Scenario: Test the aggregate state with the operation
           unblockCreditCard and state BLOCKED
                Given an aggregate BankAccount with the state BLOCKED
                When the operation unblockCreditCard is executed
                Then the aggregate state change to CREATED

Scenario: Test the aggregate state with the operation
            unblockCreditCard and state CREATED
                Given an aggregate BankAccount with the state CREATED
                When the operation unblockCreditCard is executed
                Then a illegal state exception is thrown

Scenario: Test the aggregate state with the operation
            unblockCreditCard and state EXPIRED
                Given an aggregate BankAccount with the state EXPIRED
                When the operation unblockCreditCard is executed
                Then a illegal state exception is thrown
```

```
Scenario: Test the aggregate state with the operation
          setExpired and state CREATED
              Given an aggregate BankAccount with the state CREATED
              When the operation setExpired is executed
              Then the aggregate state change to EXPIRED

Scenario: Test the aggregate state with the operation
          setExpired and state BLOCKED
              Given an aggregate BankAccount with the state BLOCKED
              When the operation setExpired is executed
              Then the aggregate state change to EXPIRED

Scenario: Test the aggregate state with the operation
          setExpired and state EXPIRED
              Given an aggregate BankAccount with the state EXPIRED
              When the operation setExpired is executed
              Then a illegal state exception is thrown
```

In der Abbildung 5.2 ist das User Interface des Prototyps zu sehen. Es kann ein CML File hochgeladen werden und die Benutzer*innen können auswählen, welche Tests generiert beziehungsweise welche Features angewendet werden sollen. Zusätzlich bietet das Frontend Buttons an, um alle oder keine Tests auszuwählen.

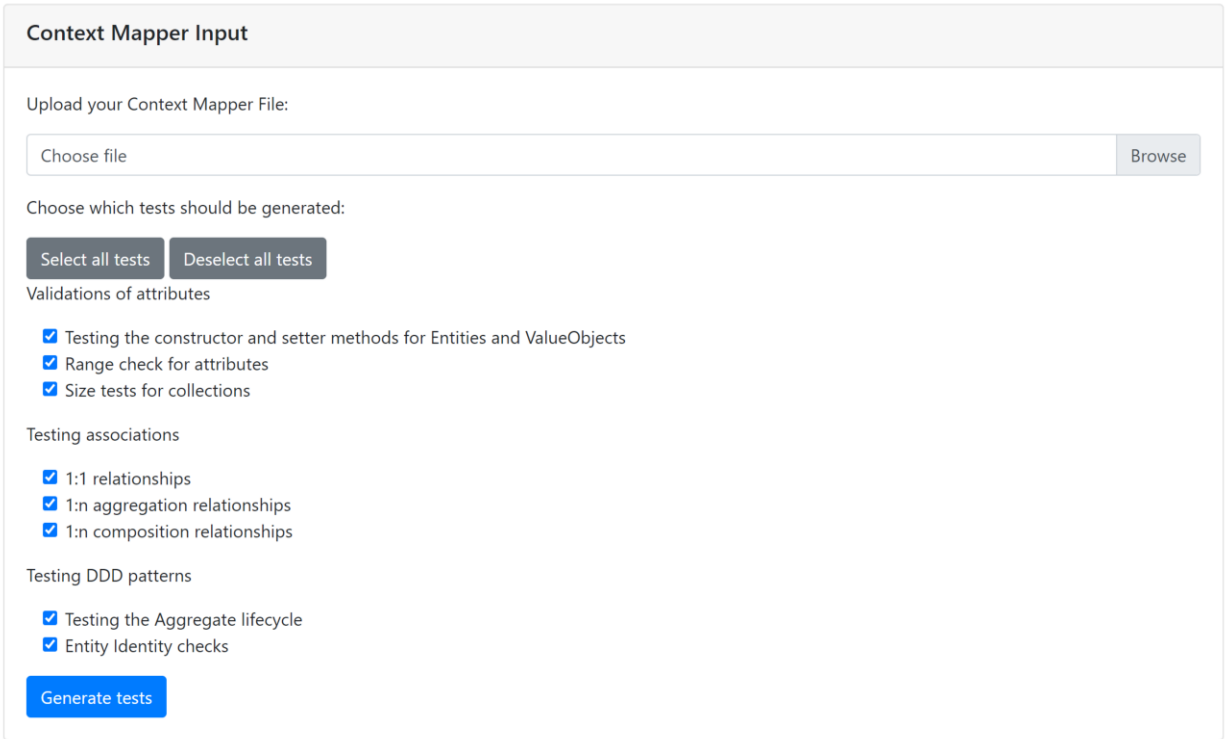

Abbildung 5.2: User Interface vor Testgenerierung

In Abbildung 5.3 ist ein Screenshot des User Interfaces nach der Testgenerierung zu sehen. Im grünen Bereich wird den Benutzer*innen mitgeteilt, ob die Testgenerierung erfolgreich war und wie viele Features Files generiert wurden. Falls während der Testgenerierung eine oder mehrere Warnungen auftreten, wird das Feld orange eingefärbt. Falls ein oder mehrere Fehler auftreten wird es rot dargestellt. In einer Liste mit aufklappbaren Elementen werden die einzelnen Feature Files dargestellt. Diese können mit dem Copy to Clipboard Button in die Zwischenablage gespeichert werden oder mit dem Download Gherkin Feature Button heruntergeladen werden. Ebenfalls steht dem Benutzer ein File zur Verfügung, dass alle Feature Files gesammelt enthält, welches ebenfalls kopiert oder heruntergeladen werden kann. Zusätzlich wird ein herunterladbares Logfile generiert, dass Details zu den einzelnen Generierungen enthält. Neben der Generierung über das grafischen User Interface kann mittels Curl[1] der Testgenerierungsprozess auch über die Command Line stattfinden.

Zur Testung, einer Implementation anhand der in den Gherkin Featue Files enthaltenen Testszenarien, wird ein BDD Tool eingesetzt. Für Softwareprojekte die mit Java entwickelt werden eignen sich hierfür zum Beispiel Cucumber oder JBehave. Um die Benutzer*innen in diesem Schritt zu unterstützten wurden zwei Tutorials erstellt, die diese Schritte genau erläutern. Das Beginner Tutorial beschreibt, wie Cucumber in ein Maven oder Gradle Projekt eingebunden werden kann. Zusätzlich wird anhand eines Beispiels erläutert, wie die Step Definitions der folgenden drei Features: “Testing the constructor and setter methods for Entities and ValueObjects“, “Range check for attributes“ und “Entity Identity checks“ implementiert werden können. Im Advanced Tutorial wird anhand eines komplexeres Beispielprojekt erklärt, wie die Step Definitions mit Cucumber für jedes Feature implementiert werden können. Zusätzlich zu den Tutorials wird den Benutzer*innen ein File zur Verfügung gestellt, welches erläutert, wie die Tests generiert werden und die Limitationen der Testszenarien aufzeigt. Im Anhang G sind die Benutzeranleitungen zu finden.

1 https://curl.se/

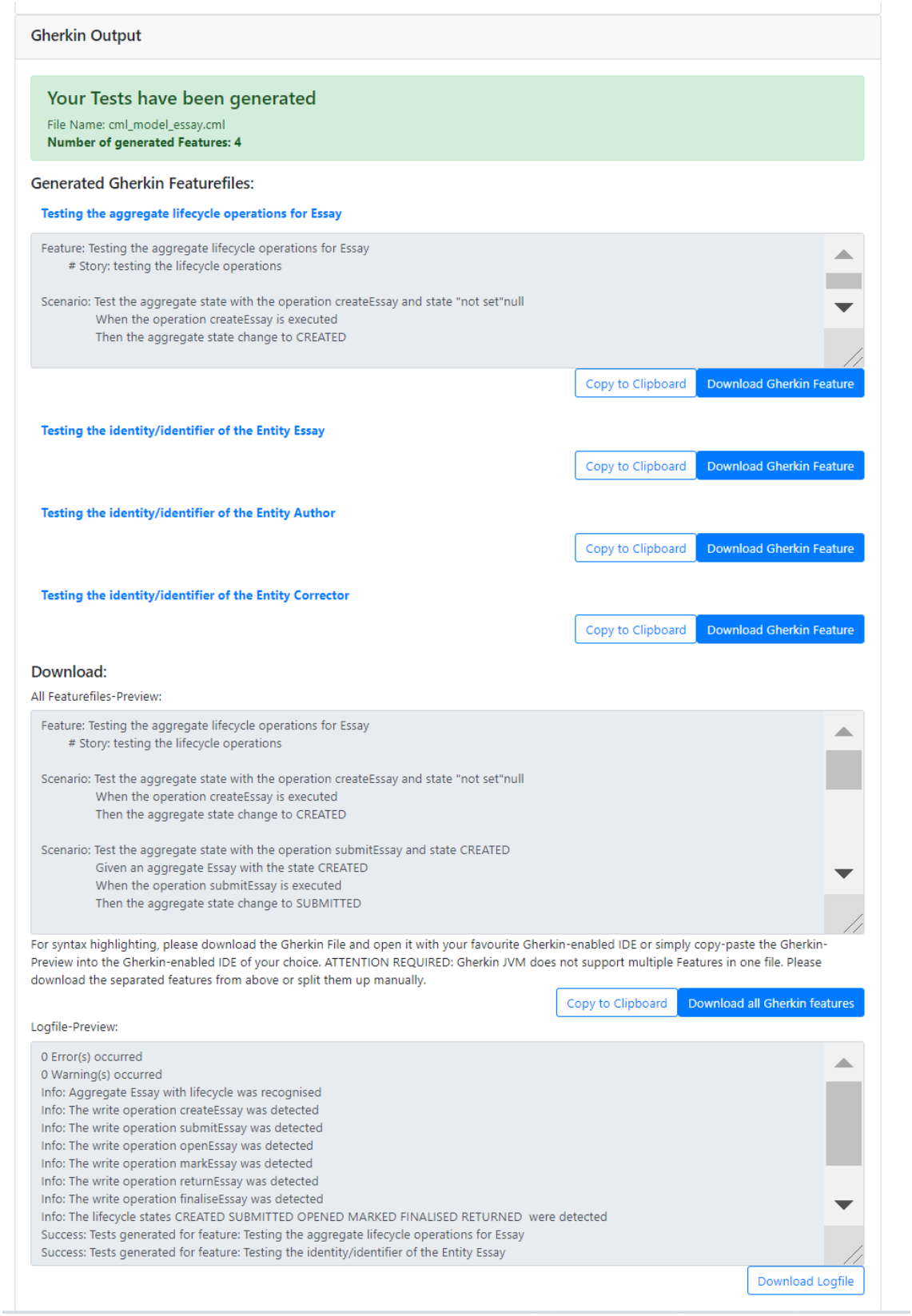

Abbildung 5.3: User Interface nach Testgenerierung

5.2 Diskussion

Der Aufwand der Testimplementierungen kann nicht vollständig abgenommen werden, sondern es findet eine Verschiebung des Aufwands von einer herkömmlichen Testimplementation in die Implementation der Step Definitions für Cucumber statt. Diese Schwäche, des in dieser Bachelorarbeit entwickelten Ansatzes, kristallisiert sich besonders bei der Erstellung der Beispielprojekte heraus, da sich die Implementation der Step Definitions als sehr zeitaufwändig gestaltet. Die Ursache der Verschiebung liegt in der Natur von Cucumber, denn dies mappt die Testszenarien beziehungsweise die einzelnen Steps auf Methoden, basierend auf deren Annotations. Die Steps selbst müssen dabei vom Nutzer ausprogrammiert werden. Dies führt zur Problematik, dass obwohl das Testszenario automatisch generiert wird, weiterhin menschliche Fehler in den Testcode einfliessen können. Cucumber bietet zwar die Möglichkeit den Step Definitions verschiedene primitive Datentypen zu übergeben, mit Data Tables können die Steps auch mehrfach ausgeführt werden, jedoch haben wir bei Projektbeginn von den Testautomatisierungsframeworks deutlich mehr Funktionen erwartet. So erhofften wir uns zeitweise von der Generierung der Code Snippets mehr als nur das Erstellen eines generischen Methodenstubs. Der hohe Implementationsaufwand bei den Step Definitions kommt nicht nur durch die wenigen Funktionalitäten zustande, sondern auch dadurch, dass es keine öffentlichen Projekte gibt, die Cucumber in einem grösseren Umfang verwenden. Die Schwierigkeit bereits publizierte Projekte zu finden, die Cucumber benutzen, zeigte sich bereits im Jahr 2013[15]. Unsere Recherchen haben ergeben, dass dieser Umstand sich kaum verändert hat. In der Cucumber Dokumentation werden meist nur simple Beispiele erklärt. Ausserdem ist in der Dokumentation nicht ersichtlich, wie mit Cucumber über mehrere Coderepositories hinweg getestet werden kann. Zudem sind keine Best Practices vorhanden und auf Stackoverflow, sowie in Blogpost werden jeweils, für dieselbe Problemstellung, wie beispielsweise zum Testen von Exceptions, unterschiedliche Methoden verwendet [16]. In Cucumber noch ein fehlendes Features ist aus unserer Sicht, dass “null“ nicht speziell, sondern nur als String behandelt wird. Zudem haben wir eine native Unterstützung von Exceptions vermisst. In Tests mit dem alternativen BDD-Framework JBehave hat sich gezeigt, dass dieses auf dem gleichen Funktionsprinzip basiert und damit auch dieselben Limitationen in sich birgt. Auch hier sind die generierten Code Snippets lediglich Methodenstubs und der grosse Anteil des Implementationsaufwand bleibt in den Händen des Entwicklers.

Des Weiteren weisen unsere generierten Tests die Schwäche auf, dass sie sehr nahe an der Implementation sind. Dies ist mit ein Grund, weshalb der gewünschte Nutzen von BDD nicht erreicht wird. Die Tests sind je nach Feature eher im Bereich der Unittests einzuordnen, welche nicht zu den Stärken von BDD gehören[9]. Ein weiterer Kritikpunkt an den generierten Testfällen ist, dass die Exceptions nach den Exceptions von Java benannt sind. Dies kann bei der Verwendung mit anderen Sprachen zu Verwirrungen führen.

Im Verlauf der Arbeit hat sich gezeigt, dass es schwierig ist Tests zu generieren, die sich auf das Softwareverhalten beziehen ohne Annahmen über die Implementation zu machen.

Die notwendigen Business-Rules sind im CML Modell nicht ausreichend dokumentiert, um sinnvolle Annahmen bei der Testgenerierung treffen zu können. Ein Beispiel hierfür sind die Operationen, welche ein Entity in zwei unterschiedliche Lifecycle Zustände überführen können. Hier fehlt im CML Modell die Information woraus sich der Entscheid zusammensetzt in welchen Zustand die Entity übergehen soll. Diese Information wäre jedoch für die Generierung eines aussagekräftigen Testfalls wichtig.

Trotz der genannten Schwierigkeiten konnte mit dem Prototyp gezeigt werden, dass es in eingeschränktem Umfang möglich ist, anhand eines CML Modells, Tests zu generieren. Ebenso wird Nutzer*innen des Context Mappers der Einstieg ins BDD erleichtert. Die Benutzer*innen kann sich aus dem eigenem CML Modell Gherkintests generieren lassen und haben so bereits einige Testszenarien, die eine Basis für die Qualitätssicherung des Softwareprojekts bieten. Zusätzlich steht ein Leitfaden zur Verfügung, der Hilfestellungen zur Implementation der Step Definitions bietet.

6 Ausblick

Der Prototypen DDD-2-BDD liesse sich durch Einfügen weiterer Funktion im CML ausbauen. So könnten zum Beispiel bei den Methoden welche eine Aggregate Lifecycle Zustandsänderung auslösen, Constraints definiert werden, um das Verhalten besser zu beschreiben. Dies würde eine bessere Grundlage für die Testgenerierung bieten, wodurch sich auch das dynamisches Verhalten besser abbilden liesse.

Obwohl der Prototyp bereits an mehreren verschiedenen Domänen getestet wurde, waren diese Domänen vergleichsweise klein, daher müsste die Funktion zusätzlich noch anhand mehrerer grösserer Beispieldomänen überprüft werden. Weiter sollte die Implementierung der Step Definitions mit komplexeren Projekten getestet und verbessert werden. Dies würde zeigen welche Arbeitsschritte gut funktionieren und die entwickelte Arbeitsmethode könnte ausgebaut werden. Mit einer Ergänzung der BDD-Tools um weitere Funktionalitäten könnte der Prototyp weiter ausgebaut und verbessert werden

Mit einer zunehmenden Verbreitung der BDD-Tools würden sich Best Practices herauskristallisieren, welche in die entwickelte Arbeitsmethode integriert werden können.

Alternativ zur gemeinsamen Anwendung von DDD und BDD mittels des entwickelten Prototyps, liessen sich die beiden Methoden auch ohne unterstützende Werkzeuge in einem Softwareprojekt anwenden. Hierfür müssten zu Projektbeginn zusammen mit dem Kunden Akzeptanztests in BDD-Form erstellt werden, woraufhin die Software mittels DDD modelliert wird. Als nächster Schritt kann basierend auf dem DDD Modell die Software implementiert werden. Danach wird die Implementation mit den BDD Akzeptanztests überprüft und ihr Verhalten auf Korrektheit validiert. Hierbei kann während des gesamten Projektverlaufs mit der Ubiquitous Language gearbeitet werden, welche in beiden Methoden vorkommt. Es wären weitere Studien notwendig um die Relevanz des DDD-2-BDD Prototyps im Vergleich zu einem Vorgehen ohne Toolunterstützung abzuschätzen.

Literaturverzeichnis

[1] *DomainDrivenDesign*. https://martinfowler.com/bliki/DomainDrivenDesign.html. Version: 03.06.2021

[2] *Behaviour-Driven Development - Cucumber Documentation*. https://cucumber.io/docs/bdd/. Version: 07.06.2021

[3] Stenberg, Jan: Behaviour-Driven Development Combined with Domain-Driven Design. In: *InfoQ* (24.2.2015). https://www.infoq.com/news/2015/02/bdd-ddd/

[4] Xebia Blog: *Combining Domain-Driven Design and Behaviour Driven Development — Xebia Blog*. https://xebia.com/blog/combining-domain-driven-design-and-behaviour-driven-development/. Version: 2018

[5] mobileLIVE: *The Value at the Intersection of TDD, DDD, and BDD - mobileLIVE*. https://perspectives.mobilelive.ca/blog/value-of-tdd-bdd-ddd/. Version: 26.01.2021

[6] Santos, Eloisa Cristina S. ; Beder, Delano M. ; Penteado, Rosangela A. D.: A Study of Test Techniques for Integration with Domain Driven Design. In: *2015 12th International Conference on Information Technology - New Generations*, IEEE, 13.04.2015 - 15.04.2015. – ISBN 978–1–4799–8828–0, S. 373–378

[7] Context Mapper: *Tactic DDD Syntax*. https://contextmapper.org/docs/tactic-ddd/. Version: 31.05.2021

[8] *The Practical Test Pyramid*. https://martinfowler.com/articles/practical-test-pyramid.html. Version: 02.06.2021

[9] Automation Panda: *BDD 101: Unit, Integration, and End-to-End Tests*. https://automationpanda.com/2017/10/14/bdd-101-unit-integration-and-end-to-end-tests/. Version: 2017

[10] Wynne, Matt ; Hellesøy, Aslak ; Tooke, Steve: *The cucumber book: Behaviour-driven development for testers and developers*. Second edition. Dallas, Texas : Pragmatic Bookshelf, 2017 (The pragmatic programmers). http://proquest.tech.safaribooksonline.de/9781680502497. – ISBN 9781680502381

[11] Team, Sculptor: *Advanced Tutorial*. http://sculptorgenerator.org/documentation/advanced-tutorial#changeable. Version: 21.03.2020

[12] Team, Sculptor: *Advanced Tutorial*. http://sculptorgenerator.org/documentation/advanced-tutorial#cascade. Version: 21.03.2020

[13] Evans, Eric: *Domain-driven design: Tackling complexity in the heart of software*. Upper Saddle River, NJ : Addison-Wesley, 2011. – ISBN 0321125215

[14] Context Mapper: *Aggregate*. https://contextmapper.org/docs/aggregate/. Version: 31.05.2021

[15] Automatically Generating Tests from Natural Language Descriptions of Software Behavior. In: *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering*, SciTePress - Science and and Technology Publications, 04.07.2013 - 06.07.2013. – ISBN 978–989–8565–62–4, S. 238–245

[16] samach: *Cucumber JVM: Test if the correct exception is thrown*. https://stackoverflow.com/questions/17272161/cucumber-jvm-test-if-the-correct-exception-is-thrown. Version: 2013

[17] *DomainDrivenDesign*. https://martinfowler.com/bliki/DomainDrivenDesign.html. Version: 08.06.2021

[18] EventStorming: *EventStorming*. https://www.eventstorming.com/. Version: 31.03.2020

[19] Team, Sculptor: *Advanced Tutorial*. http://sculptorgenerator.org/documentation/advanced-tutorial. Version: 21.03.2020

[20] Solis, Carlos ; Wang, Xiaofeng: A Study of the Characteristics of Behaviour Driven Development. In: *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, IEEE, 30.08.2011 - 02.09.2011. – ISBN 978–1–4577–1027–8, S. 383–387

[21] Cucumber.io: *Introducing Example Mapping*. https://cucumber.io/blog/bdd/example-mapping-introduction/. Version: 10.03.2021

[22] John Ferguson Smart: *Feature Mapping - a lightweight requirements discovery practice for agile teams - John Ferguson Smart*. https://johnfergusonsmart.com/feature-mapping-a-lightweight-requirements-discovery-practice-for-agile-teams/. Version: 2019

[23] *Gherkin Reference - Cucumber Documentation*. https://cucumber.io/docs/gherkin/reference/. Version: 11.06.2021

[24] *Composite Steps*. https://jbehave.org/reference/stable/composite-steps.html. Version: 20.01.2021

[25] Gamage, Thilina A.: JBehave Vs Cucumber JVM: Comparison and Experience Sharing. In: *Agile Vision* (26.7.2017). https://medium.com/agile-vision/jbehave-vs-cucumber-jvm-comparison-and-experience-sharing-439dfdf5922d

[26] *JBehave vs Cucumber | LibHunt*. https://java.libhunt.com/compare-jbehave-vs-cucumber-jvm. Version: 11.06.2021

[27] CardBoard: *CardBoard | Digital Whiteboard and User Story Mapping.* https://cardboardit.com/. Version: 22.04.2021

[28] FeatureMap: *Story Mapping Software | Product, Design, Management | FeatureMap.co.* https://www.featuremap.co/en. Version: 11.06.2021

[29] Atlassian Marketplace: *Example Mapping for Jira.* https://marketplace.atlassian.com/apps/1220677/example-mapping-for-jira?hosting=cloud&tab=overview. Version: 11.06.2021

[30] Xebia Blog: *Extending the Bounded Context Canvas with BDD Examples — Xebia Blog.* https://xebia.com/blog/extending-the-bounded-context-canvas-with-bdd-examples/. Version: 2020

[31] North, Dan: BDD & DDD. In: *InfoQ* (3.12.2009). https://www.infoq.com/presentations/bdd-and-ddd/

[32] Landre, Einar ; Wesenberg, Harald ; Olmheim, Jorn: Agile enterprise software development using domain-driven design and test first. In: Gabriel, Richard P. (Hrsg.): *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion.* New York, NY : ACM, 2007. – ISBN 9781595938657, S. 983

[33] Sarmiento, Edgar ; Leite, Julio C. ; Almentero, Eduardo ; Sotomayor Alzamora, Guina: Test Scenario Generation from Natural Language Requirements Descriptions based on Petri-Nets. In: *Electronic Notes in Theoretical Computer Science* 329 (2016), S. 123–148. http://dx.doi.org/10.1016/j.entcs.2016.12.008. – DOI 10.1016/j.entcs.2016.12.008. – ISSN 15710661

[34] SOFTWARE TESTING Fundamentals: *Unit Testing - SOFTWARE TESTING Fundamentals.* https://softwaretestingfundamentals.com/unit-testing/. Version: 2011

[35] AI-driven E2E automation with code-like flexibility for your most resilient tests: *What Is Test Automation? A Simple, Clear Introduction.* https://www.testim.io/blog/what-is-test-automation/. Version: 2019

[36] Philipp Hauer's Blog: *Modern Best Practices for Testing in Java.* https://phauer.com/2019/modern-best-practices-testing-java/. Version: 2019

[37] Newman, Jessie: Unit Testing Best Practices - The Startup - Medium. In: *The Startup* (10.8.2020). https://medium.com/swlh/unit-testing-best-practices-853064ad3551

[38] Agile Alliance |: *What is Test Driven Development (TDD)?* https://www.agilealliance.org/glossary/tdd/#q=~(infinite~false~filters~(postType~(~'page~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'tdd))~searchTerm~'~sort~false~sortDirection~'asc~page~1). Version: 2015

[39] *Commandline | EvoSuite*. https://www.evosuite.org/documentation/commandline/. Version: 08.03.2021

[40] *IntegrationTest*. https://martinfowler.com/bliki/IntegrationTest.html. Version: 08.06.2021

[41] TheCodeBuzz: *Integration Testing Best Practices in Agile*. https://www.thecodebuzz.com/integration-testing-best-practices-agile/. Version: 2019

[42] Gafert, Peter: *Unit test your Java architecture*. https://www.archunit.org/. Version: 30.05.2021

[43] StackShare: *What are the best Java Build Tools ?* https://stackshare.io/java-build-tools. Version: 14.06.2021

[44] *Installation - Cucumber Documentation*. https://cucumber.io/docs/installation/. Version: 14.06.2021

[45] StackShare: *Cucumber vs JBehave | What are the differences?* https://stackshare.io/stackups/cucumber-vs-jbehave. Version: 14.06.2021

[46] *Client/Server Architectures for Business Information Systems*. 1997 http://www.objectarchitects.de/objectarchitects/papers/published/zippedpapers/renzel.pdf

[47] JAXenter: *Java-Trends: Top 10 der Frameworks im Jahr 2020*. https://jaxenter.de/java/java-trends-frameworks-91786. Version: 2020

[48] Agile Alliance |: *What are CRC Cards?* https://www.agilealliance.org/glossary/crc-cards/#q=~(infinite~false~filters~(postType~(~'page~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'crc*20card))~searchTerm~'~sort~false~sortDirection~'asc~page~1). Version: 2015

[49] *rfc2388*. https://datatracker.ietf.org/doc/html/rfc2388. Version: 14.06.2021

[50] GitHub: *cucumber/common*. 14.06.2021

[51] *10 Minute Tutorial - Cucumber Documentation*. https://cucumber.io/docs/guides/10-minute-tutorial/. Version: 14.06.2021

Anhang

A Analysebericht

A.1 Einführung

A.1.1 Zweck

Das vorliegende Dokument beinhaltet den Analysebericht der Bachelorarbeit Von DDD zu BDD.

A.1.2 Gültigkeitsbereich

Die Gültigkeit des Analyseberichts beschränkt sich auf die Laufzeit der Bachelorarbeit im FS2021.

A.2 Übersicht

Als ersten geben wir einen Einblick in unser Vorgehen bei der Recherche. Die folgenden Abschnitte fassen unsere Rechercheergebnisse über den aktuellen Stand der Werkzeuge und Anwendung von Domain Driven Design (DDD) und Behavior Driven Development (BDD) zusammen. Im Zusammenhang mit DDD und BDD stellen wir noch einzelne Workshop Formate vor. Zum Abschluss werden die Testgenerierungskonzepte und die Best Practice im Bereich Unit- und Integrationstesting erläutert.

A.3 Vorgehen

Bei der Analyse ging es um eine Bestandsaufnahme der aktuellen Werkzeuge, Tools und Verwendungen von DDD und BDD. Wir haben verschiedene Tools ausprobiert. Darunter waren insbesondere JBehave und Cucumber, die durch die Aufgabenstellung vorgegeben waren. Weitere Tools haben wir aufgrund von unseren Rechercheergebnissen ebenfalls dokumentiert. Der zweite Teil der Analyse war die Recherche nach den aktuellen Methoden, die Methoden haben wir aus der Aufgabenstellung übernommen und mit einigen Methoden ergänzt, die wir besonders interessant gefunden haben in Bezug zu unserem Projekt. Dazu haben wir

hauptsächlich mit der Suchmaschine Google und Google Scholar gearbeitet. Google Scholar wurde besonders für die Suche nach wissenschaftlichen Quellen verwendet. Die verwendeten Suchbegriffe sind bei den Abschnitten vermerkt. Einige Papers fanden wir auch mithilfe der Snowballing Methode[1].

A.4 Domain Driven Design (DDD)

Key words: domain driven design, DDD, event storming, DDD tools, DDD modelling

Domain-Driven Design ist ein Ansatz zur Softwareentwicklung, bei dem die Entwicklung auf der Programmierung eines Domänenmodells basiert, das ein umfassendes Verständnis der Prozesse und Regeln einer Domäne aufweist. Der Name stammt aus einem Buch von Eric Evans aus dem Jahr 2003, das den Ansatz anhand eines Katalogs von Patterns beschreibt.[17]

A.4.1 Anwendung

DDD wird besonders für die Modellierung von komplexen Domänen verwendet. Mithilfe von DDD kann ein Problem Space in kleinere Teile, sogenannte Bounded Context aufgeteilt werden. Dies bietet zum Beispiel den Vorteil, dass eine weniger starke Koppelung stattfindet und Teams unabhängiger voneinander an verschiedenen Bounded Contexts arbeiten können. Die Ubiquitous Language bietet ebenfalls ein wichtiger Grundstein im DDD. Unter Ubquitous Language versteht man eine Sprache über die Domäne, die von allen Teammitglieder verstanden wird. DDD wird meist mit agilem Vorgehen kombiniert.

A.4.2 Event Storming

Event Storming ist ein flexibles Workshop-Format für die kollaborative Erkundung komplexer Geschäftsdomänen. Es handelt sich jedoch nicht um ein DDD Pattern. Event Storming ist sozusagen eine Ergänzung von DDD mit der gleichen Absicht, das Wissen über die Domäne zu erweitern. Ebenfalls können aus einem Event Storming beispielsweise Kandidaten für Bounded Context herauskristallisieren. Event Storming wird in der Praxis häufig in Kombination mit DDD angwendet. [18]

A.4.3 Werkzeuge und Tools

Im folgenden Abschnitt erläutern wir einige Werkzeuge, die für DDD verwendet werden.

1 https://medium.com/edyvision/search-queries-v-s-snowball-method-1bcfc5c63f7d

A.4.3.1 Freie Modellierung

DDD wird oft frei modelliert, d.h. mit Stift und Papier oder auch auf Whiteboards mit Sticky Notes. In der Abbildung A.1 ist ein eigenes gezeichnets Beispiel einer Context Map zu sehen.

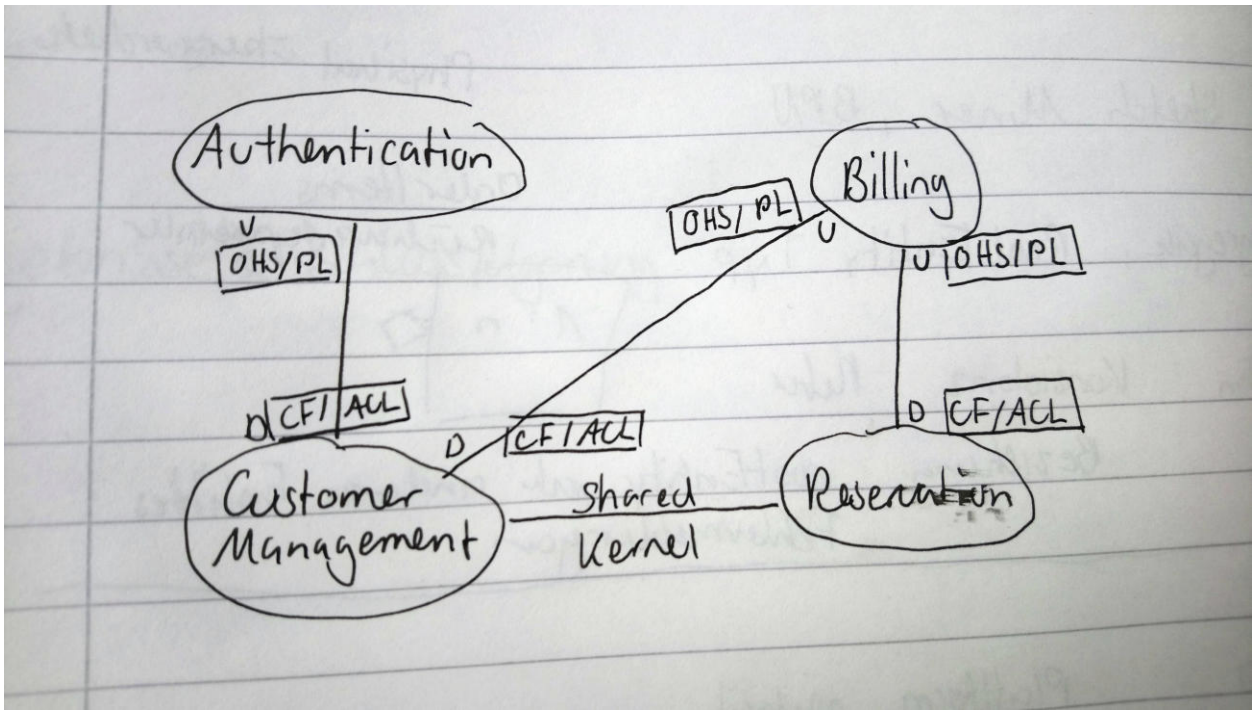

Abbildung A.1: Handgezeichnete Context Map (eigene Darstellung)

A.4.3.2 Context Mapper

Der Context Mapper[2] bietet die Möglichkeit DDD-Modelle in einer maschinenlesbaren Sprache zu modellieren. Dies bietet den Vorteil automatisierte Refactorings durchzuführen und grafische Darstellungen beispielsweise mit Graphviz zu generieren. Der Context Mapper unterstüzt Stragetic und Tactic DDD. Der Syntax für das Tactic DDD basiert auf dem Sculptor Generator [7]. Weiter Informationen zur Syntax sind auf der Webseite des Context Mappers zu finden.

A.4.3.3 Sculptor Generator

Der Scupltor Generator[3] stellt eine Sprache zur Verfügung, die Tactic DDD abbildet. Ebenfalls bietet sich die Möglichkeit aus dem definierten DDD Modell Java Code zu generieren mit

2 https://contextmapper.org/
3 http://sculptorgenerator.org/

dazugehörigen Tests. Der Code sowie die Tests enthalten offene Punkte, die man noch ergänzen muss. Es werden hauptsächlich die Klassen und Methodenheaders generiert. In der Abbildung A.2 ist ein Sculptor DSL Modell zu sehen und in der Abbildung A.3 ist der Code zu sehen der von Sculptor generiert wurde. In der Abbildung A.4 ist der dazugehörige Unittest zu sehen, der ebenfalls vom Sculptor generiert wurde. Beim Testen vom Sculptor Generator ist uns aufgefallen, dass hauptsächlich die Methoden und die Entities mit den Beziehungen vom Sculptor Model in den Java Code transferiert werden. Die Konvertierungskonzepte vom Sculptor Generator sind nur sehr knapp anhand von Beispielen dokumentiert.[19]

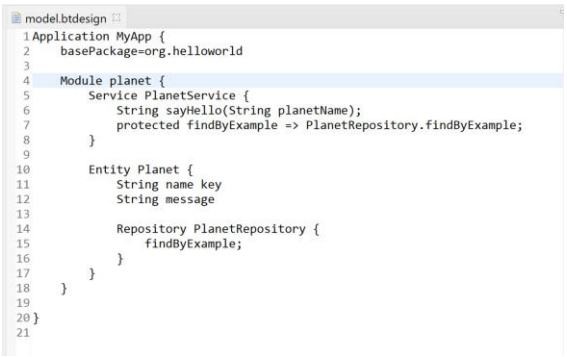

Abbildung A.2: Sculptor DSL File

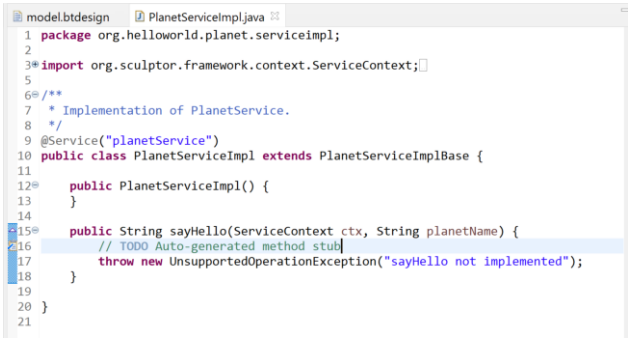

Abbildung A.3: Java Code von Sculptor generiert

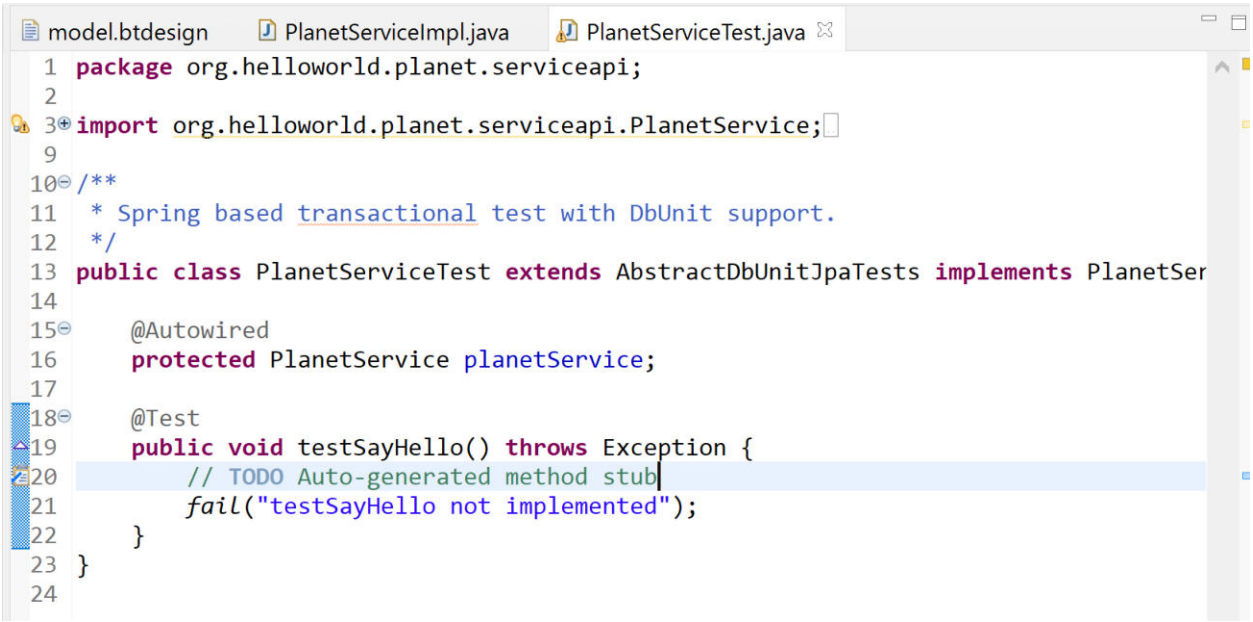

Abbildung A.4: Unittest von Sculptor generiert

A.4.3.4 Vergleich der drei Werkzeuge

In der Abbilung A.5 ist eine Tabelle mit den Vergleichen der Werkzeuge zu sehen. Grün sind alle positiven Merkmale, hell-orange die mittelmässigen Eigenschaften und orange kritische Merkmale. Als positive Merkmale wurden die Eigenschaften bezeichnet, die in unserem Projekt eine Chance bieten und die sich von den anderen Tools stark abheben. Als orange wurden die Merkmale gezeichnet, die weder eine Chance noch ein Risiko darstellen. Die kritischen Merkmale sind Merkmale, die ein Risiko in unserem Projekt darstellen könnten, wenn die Tools verwendet werden.

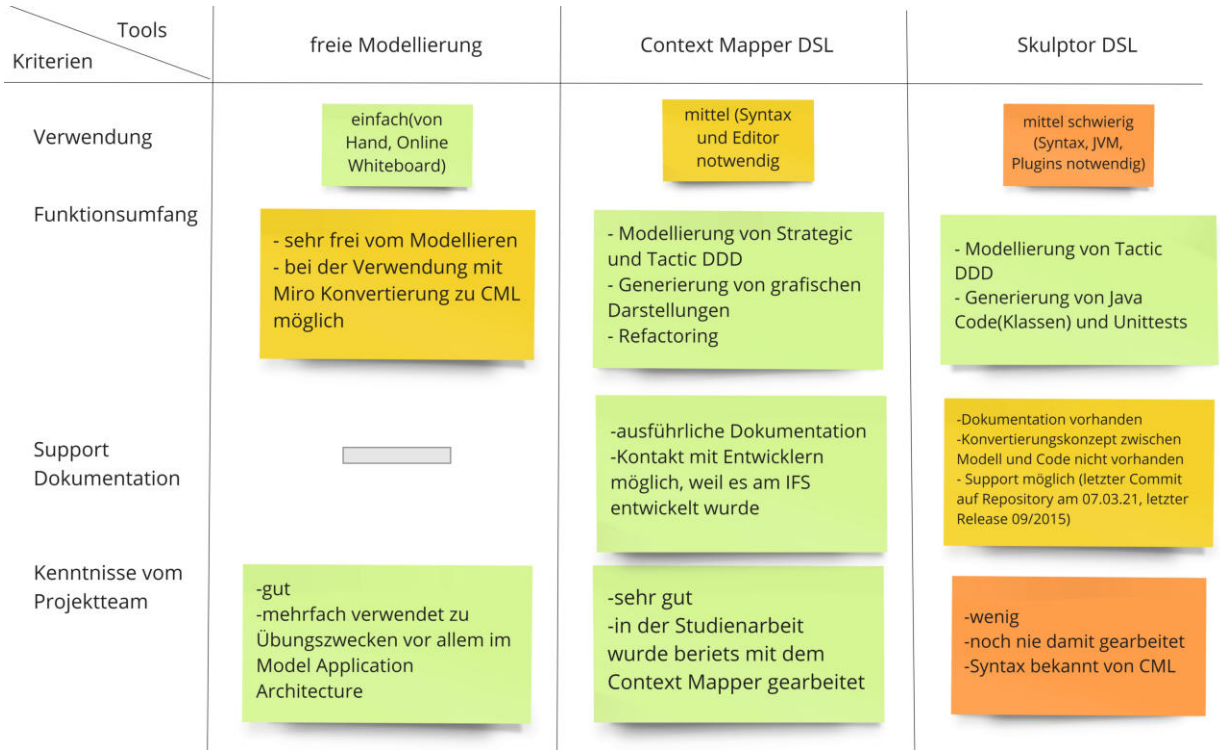

Abbildung A.5: DDD Werkzeug Vergleich

A.5 Behaviour Driven Development (BDD)

Keywords: behaviour driven development, bdd, gherkin, example mapping, feature mapping

BDD ist eine Kombination von Ubiquitous Language, Test Driven Development und automatisierten Akzeptanztests. BDD wird durch sechs Eigenschaften ausgezeichnet[20]:

- Specification by Example
- Ergänzung von User Stories mit Beispiel Szenarien

- Ubiquitous Language
- Iterative Decomposition Prozesse
- Automatisierte Akzeptanztests mit Mapping Regeln
- lesbarerer Behaviour orientierter Spezifikationscode
- Behaviour Driven in verschiedene Phasen

A.5.1 Anwendung

Es gibt bereits mehrere Tools, die BDD unterstützten bei der Implementierung[20]. Zu ihnen zählen wird Cucumber und JBehave. Diese Tools werden in den nächsten Abschnitten noch genauer erläutert. Sie fokussieren sich hauptsächlich auf die Unterstützung von Integrations und End-zu-end Tests[9]. BDD bietet ebenfalls Methoden an, die die Kommunikation erleichtern sollen um schneller Akzeptanzkriterien zu definieren, wie Example Mapping oder Feature Mapping. Die folgenden zwei Abschnitte beschreiben die beiden Praktiken im Detail.

A.5.2 Example Mapping

Das Example Mapping wurde von Matt Wynne erfunden und ist eine Methode um die Akzeptanzkriterien von einer User Story zu definieren und die Problemdomäne besser zu verstehen. Das Example Mapping soll die Meetings für die Spezifikation der Akzeptanzkriterien verkürzen und produktiver machen. [21]

A.5.2.1 Vorgehen

Als erstes wird die User Story, die betrachtet wird, auf eine gelbe Karte geschrieben und oben in der Mitte der Tabelle platziert. Als nächstes werden Akzeptanzkriterien oder Regeln bestimmt, diese werden auf blaue Karte aufgeschrieben. Für jede Regel wir eines oder mehrere Beispiele bestimmt, diese schreibt man auf eine grüne Karte. Wenn während der Diskussion Fragen auftauchen, die von niemandem beantwortet werden können, dann schreibt man diese auf eine rote Karte auf. Sobald der Scope von der User Story für alle klar ist, kann das Meeting beendet werden.[21] Miro bietet ein Example Mapping Template an. In der Abbildung A.6 ist das ausgefüllte Example Mapping Template von Miro[4] zu sehen, dass von uns ergänzt wurde.

4 https://miro.com/templates/example-mapping/

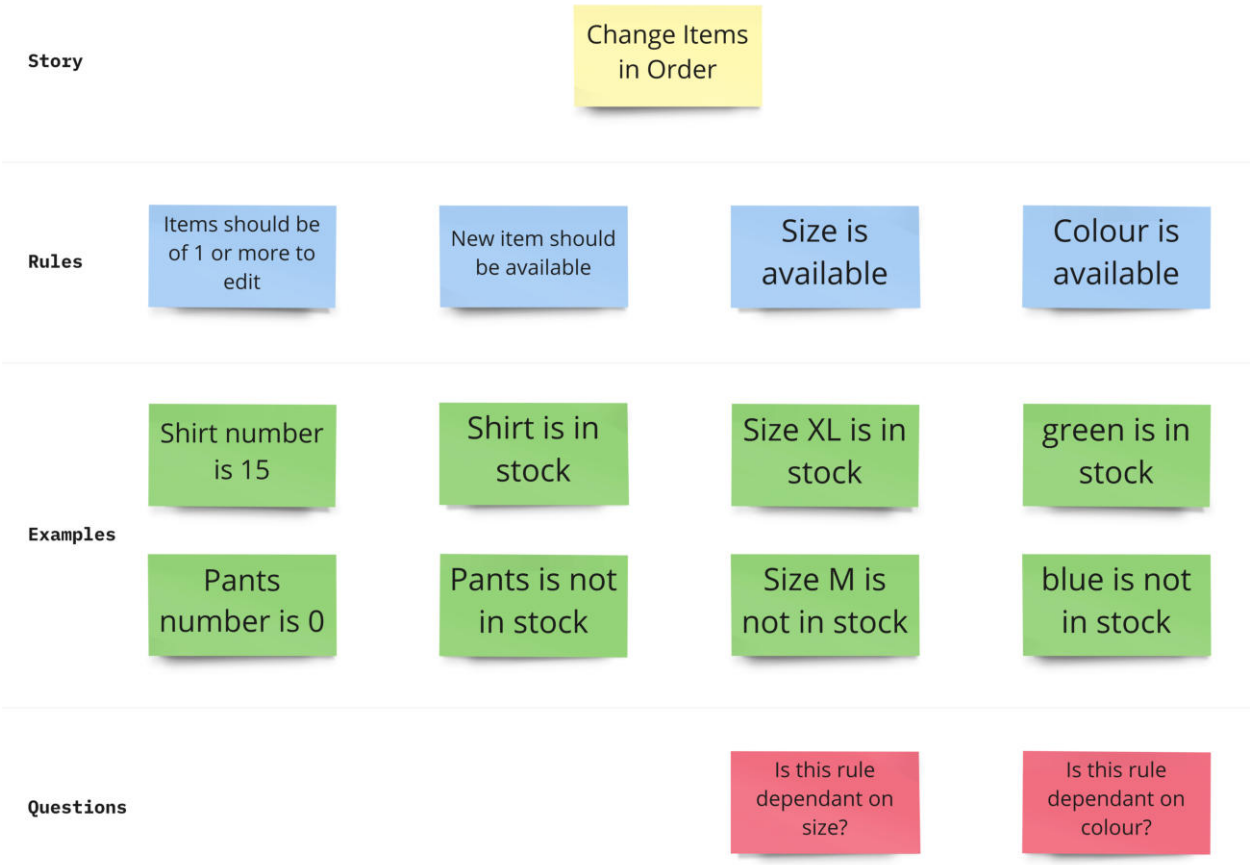

Abbildung A.6: Example Mapping: Miro Template Beispiel

A.5.3 Feature Mapping

Feature Mapping ist eine Methode, die dem Example Mapping sehr ähnlich ist. Es verfolgt, dasselbe Ziel wie das Example Mapping, die Akzeptanzkriterien zu definieren. Im Vergleich zum Example Mapping fokussiert es sich jedoch mehr auf die unterschiedlichen Schritte innerhalb einer User Story und ist nach unserer Meinung etwas näher an den automatisierten Testfällen, als das Example Mapping.[22]

A.5.3.1 Vorgehen

Als erstes wird eine Story oder ein Feature ausgewählt, dass spezifiziert werden soll. Danach wird die Story in mehrere Tasks oder Steps unterteilt. Die Tasks werden auf hellgelbe Karten aufgeschrieben. Danach werden verschiedene Beispiele der einzelnen Steps besprochen. Dann werden Beispiele bestimmt, die einen Flow der einzelnen Steps veranschaulichen. Der Titel des Beispiels wird auf eine grüne Karte geschrieben und die einzelnen Steps auf eine gelbe

Karte. Konsequenzen von einzelnen Steps können mit violetten Karten dargestellt werden. Es kann auch eine Konsequenz Kolonne hinzugefügt werden, falls dies notwendig ist. Dies ist beispielsweise bei einer Berechnung der Note basierend auf den Punkten sinnvoll. Business Rules können zusätzlich mit blauen Karten hervorgehoben werden. Auf den roten Karten werden jeweils die Fragen notiert.[22] In der Abbildung A.7 ist ein Beispiel zu sehen. Das Beispiel stellt die User Story «Teachers can return bad essays to be corrected. In order to allow students to learn from their mistakes. As a teacher marking student esays I want to be able to return an essay to a studen for corrections when the marks are poor», das Beispiel stammt von John Ferguson Smart.

Abbildung A.7: Feature Mapping Beispiel mit Beschriftung

A.5.4 Werkzeuge, Tools

Keywords: BDD, BDD Tools, BDD Framework, Cucumber, JBehave

Für die Beschreibung von Verhaltensspezifikationen der Nutzungsszenarien einer Software wird unter Anwendung der BDD Prinzipien ein Spezielles Format, welches Gherkin-Sprache genannt wird verwendet. Dabei werden User Stories beziehungsweise Szenarien spezifiziert. Dieses Format wird von verschiedenen BDD Tools wie zum Beispiel Cucumber und JBehave unterstützt, wobei die Tools das Format unterschiedlich genau Anwenden. Während JBehave

zusätzlich ein eigenes Format definiert wird trotzdem sichergestellt, dass auch Input im Gherkin-Format verwendet werden kann. Dadurch ist das Gherkin-Format ein defacto Standard in der BDD Community. Das Format ist wie folgt definiert:

Format-Komponente	Beschreibung der Komponente	
Titel	Ein möglichst deskriptiver und expliziter Titel zur eindeutigen Identifizierung der User Story	
Story Beschreibung	Eine kurze Einleitung in die User Story in folgendem Format:	
	As a:	Die Person oder Rolle welche von dem Feature profietiert beziehungsweise es nutzt
	I want:	Das Feature
	so that:	Der Nutzen oder Vorteil den das Feature mit sich bringt.
Akzeptanzkriterien	Eine spezifische Beschreibung aller User Story Szenarien Dabei kommt folgendes Format zum Einsatz:	
	Given:	Vorbedienungen
	When:	Auslösender Event/Trigger
	Then:	Das erwartete Ergebnis

Dabei werden die Given-When-Then Instruktionen als Steps bezeichnet. Diese Steps beschreiben die Schritte mit welchen das Szenario durchschritten werden.

A.5.4.1 Cucumber

Das BDD Framework Cucumber[5] ermöglicht es die mit Gherkin definierten User Stories mittels Annotations direkt auf Code zu mappen, wodurch Tests verständlich dokumentiert und automatisiert ausgeführt werden können. Cucumber Integration sind für eine Vielzahl von Programmiersprachen verfügbar darunter zum Beispiel Java, C++, Javascript, Ruby und Go.

In Cucumber werden die User Stories in Feature Files definiert. Dabei kommt der BDD typischen Given-When-Then Syntax zum Einsatz. Der Syntax basiert dabei auf einigen Schlüsselwörtern welche vom Tool zum Parsen der Feature Files verwendet werden. Die folgenden Primären Schlüsselwörter werden verwendet[23]:

- Feature (High-Level Beschreibung des Software Features (Titel), wird zur Gruppierung der Szenarien verwendet)
- Rule (business Rule, zur Strukturierung der Szenarios dazugehörigen Szenarios)
- Example (alternativ kann der Alias Scenario verwendet werden)

5 https://cucumber.io/

- Given, When, Then, And, But für Steps (Zusätzlich kann auch das Zeichen * als Step Definition verwendet werden, dabei entspricht * dem vorhergehendem Given, When oder Then)
- Background (Kann genutzt werden um Preconditions welche für alle Szenarien in einem Feature gültig sein müssen an einem Ort zu definieren, sie werden dann für jedes Szenarie dieses Features überprüft)
- Scenario Outline (alternativ kann der Alias Scenario Template verwendet werden)
- Examples (alternativ kann der Alias Scenarios verwendet werden)

Welche von den folgenden Sekundären Schlüsselwörter ergänzt werden.

- `""" (Doc Strings)`
- | (Data Tables)
- @ (Tags)
- `# (Comments)`

A.5.4.2 JBehave

JBehave[6] ist ein weiteres BDD Framework das BDD User Stories auf Code mappt und so Automatisiertes Testen mit BDD Spezifikationen ermöglicht. JBehave ist Java spezifisch. JBehave arbeitet primär mit einem eigenen Syntax zur Beschreibung der User Stories und Szenarien kann aber auch Szenarien in Gherkin Format verarbeiten. Der grösste Untschied zwischen Standard Gherkin und JBehaves Adaption besteht im Composite Step, einem Step welcher mehrere andere Steps in sich zusammen fasst[24].

A.5.4.3 Cucumber und JBehave im Vergleich

Die beiden BDD Frameworks Cucumber und JBehave stehen in direkter Konkurrenz zueinander, da beide Tools die gleichen Problemstellungen adressieren. Daher liegt es nahe, die beiden Frameworks miteinander zu vergleichen. Mit einer kurzen Recherche lassen sich einige Blogposts und sonstige Quellen im Internet finden, welche genau einen solchen Vergleich ziehen.[25][26] Durch den direkten Vergleich wird ersichtlich, dass Cucumber und JBehave was ihre Feature angeht sehr ähnlich aufgestellt sind, Cucumber aber mit einer besseren Integration in die IDE, flexiblerer Formatierung und besseren Reports punkten kann. Zusätzlich ist auch die Cucumber Community bei Github und StackOverflow aktiver als jene von JBehave.

6 https://jbehave.org/

A.5.4.4 Tools für Example Mapping und Feature Mapping

Example Mapping und Feature Mapping können sehr gut mit Papier und Stickie Notes durchgeführt werden. Ebenfalls eignen sich Online Whiteboards, wie beipsielsweise Miro[7] dafür. In Miro gibt es fürs Example Mapping bereits ein vorgefertigtes Template. Desweitern gibt es das Tool CARDBOARD[8]. In CARDBOARD kann eine User Story Map im Team generiert werden. Es bietet ebenfalls die Möglichkeit sogenannte Customer Journeys zu erstellen[27], diese Funktion deckt sich teilweise mit der Idee des Feature Mappings. Ein ähnliches Tools ist auch FeatureMap[9], der Fokus liegt hier aber auch bei der Story Map[28]. Zudem gibt es noch das Example Mapping for Jira[10]. Dieses Tool orientiert sich sehr stark an den Vorgaben vom Example Mapping, wie in der Abbildung A.8 zu sehen ist. Es bietet jedoch den Vorteil, zum Vergleich mit dem Arbeiten auf Whiteboards, dass die Examples und Rules direkt ins Arbeitstool Jira übernommen werden können und während der Entwicklung zur Überprüfung dienen.[29]

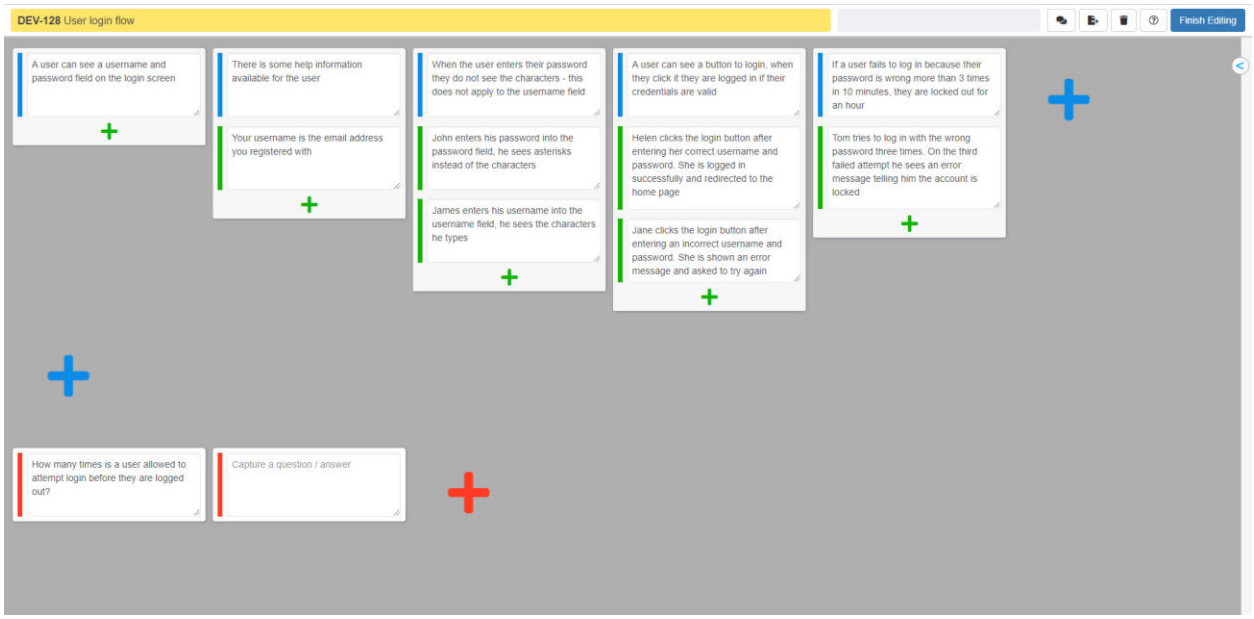

Abbildung A.8: Example Mapping Beispiel mit Jira

Ob eines der genannten Tools bei uns in Frage kommt ist noch unklar. Die vorgestellten Tools sind leider alle, bis auf Miro, kostenpflichtig. Weil das Ziel ist, dass unser Endsoftware Open Source zur Verfügung gestellt wird, sind die deshalb eher weniger geeignet. Ebenfalls ist unklar, wie weit verbreitet die genannten Tools sind. Nebst dem Example Mapping for Jira bieten die anderen Werkzeuge aus unserer Sicht keinen markanten Mehrwert im Vergleich zum

7 https://miro.com/about/
8 https://cardboardit.com/
9 https://www.featuremap.co/en
10 https://marketplace.atlassian.com/apps/1220677/example-mapping-for-jira?hosting=cloud&tab=overview

Modellieren auf Papier oder einem Whiteboard. Im Gegenteil, sie könne sogar einschränken und Einarbeitungszeit kosten.

A.6 DDD in Kombination mit BDD

Keywords: DDD and BDD, DDD combination with BDD, DDD test techniques

DDD und BDD wird gemäss diversen Blogeinträgen und Talks bereits kombiniert benutzt. Beispielsweise werden Bounded Context mithilfe von BDD Test bereits bei der Modellierung überprüft und allenfalls ergänzt. Die beiden Techniken scheinen sich sehr gut zu ergänzen. Die BDD Szenarien bieten eine gute Grundlage zur Überprüfung, ob das Domänenmodell richtig erstellt wurde. Eine nennenswerte Schnittstelle zwischen den beiden Techniken ist auch die Ubiquitous Language. Die Ubiquitous Language bildet bei beiden Techniken die Grundlage für die Kommunikation im Team.[4][30][31][3] Die Ubiquitous Language kann für das Domänenmodell, sowie auch für die dazugehörigen Testszenarien verwendet werden.

Zusammen mit BDD und DDD kommt häufig zusätzlich noch Test Driven Development (TDD) zum Zuge.[5] Eine wissenschaftliche Untersuchung zeigte, dass die Kombination von Agiler Entwicklung, DDD, TDD und JDO eine deutliche Verbesserung der Fähigkeiten, qualitativen hochwertigen Code zu entwickeln, ergibt.[32]

Eine weitere Untersuchung zeigte, dass die Integration von TDD und BDD mit DDD möglich ist[6]. Diese Studie bestätigt, dass die technische Machbarkeit unserer Arbeit in groben Zügen bereits gegeben ist.

A.7 Automatisierte Testgenerierung

Keywords: test scenario generation, test generation

Um von DDD zu BDD zu kommen, müssen BDD Testszenarios generiert werden. Aus diesem Grund suchten wir nach bekannten Testgenerieungskonzepten und Testtechniken. Im folgenden Abschnitt sind die wichtigsten Erkenntnisse aufgeführt.

Das Generieren von Testszenarien anhand von Szenarien, die in der natürlichen Sprache definiert wurden, wurde bereits untersucht. Die Szenarien werden mit einer definierten Syntax und semantischen Regeln spezifiziert. Aus den Szenarien werden dann Petri Nets generiert. Die

Petri Nets werden auf Eindeutigkeit, Vollständigkeit, Konsistenz und Korrektheit überprüft. Danach können von den Petri Nets Test Szenarien generiert werden.[33]

Die Generierung von Testscenarios wurde bereits mit UML-Activity Diagrammen untersucht. Es ist möglich mithilfe von einem UML-Acitivity Diagramm Testszenarien zu generieren.[15]. Dies scheint auf den ersten Blick der Aufagbenstellung nicht besonders ähnlich. Jedoch bietet der Context Mapper, die Möglichkeit den State von einem Aggregate abzubilden und bei den Operationen die Stateänderungen zu dokumentieren. Über die Stateänderungen wäre es möglich, den Flow nach zu vollziehen. Aus dem Flow könnte eine Art Actitvity Diagramm erzeugt werden und aus diesem die Testszenarien. Die drei wichtigsten Merkpunkte von der Testgnerierungs von Activity Diagramms sind:

- Activity Coverage
- Transition Coverage
- Path Coverage

A.8 Best Practice Unit- und Integrationstests

Keywords: best practice unittesting, best practice integrationtesting, unittesting, integrationtesting, test driven development

In diesem Abschnitt zeigen wir die aktuellen Best Practises auf zum Thema Unit- und Integrationstesting.

A.8.1 Unittesting

Unittests sind Whitebox Tests, die von den Entwicklern geschrieben werden. Durch das Testen von kleinen Units, soll sichergestellt werden, dass jede Komponente so funktioniert, wie sie designt wurde. Gute Unittest bieten die Grundlage fürs Refactoring und machen den Code besser wartbar. Unittest minimieren ebenfalls die Zeit zum Debuggen und senken das Geschäftsrisiko, weil Fehler früh erkannt werden.[34] In der Agilen Entwicklung werden Unittest mit der Continous Integration automatisiert ausgeführt[35].

In der folgenden List sind einige Best Practices zusammengetragen, die man beim Schreiben von Unittest beachten soll[36][34][37]:

- Unittest sollten gute Namen haben, die beschreiben, was genau getestet wird.
- Insgesamt sollen alle Use und Edge Cases von jeder public Funktion durch die Unittest abgedeckt werden.

- In einem einzelnen Unittest sollte möglichst wenig Logik enthalten sein. Ein Unittest sollte möglichst nur eine Funktion testen.
- Ein Unittest sollte in drei Teile aufgeteilt werden. Die Testvorbereitung mit dem Erstellen der Daten und der Mock Objekte, danach folgt die Aktion und die zu testende Funktion wird ausgeführt. Im dritten Teil wird die Assertions ausgeführt um den Output oder das Verhalten von der Aktion zu überprüfen. Diese Aufteilung kann auch mit den Keywords Given, When und Then gemacht werden.

A.8.1.1 Test Driven Development(TDD)

TDD ist ein Programmierstyle der Testen, Programmieren und Design eng miteinander verbindet. Beim TDD wird als erstes ein Test geschrieben der eine Komponenete vom System beschreibt. Die Ausführung des Tests schlägt fehl, weil das Feature noch nicht implementiert ist. Danach schreibt man nur so viel Code, wie nötig ist um den Test grün zu machen. Dann kann man den Code noch anpassen, bis er den Qualitätsanforderungen entspricht.[38]

A.8.1.2 Tool EvoSuite

Bei unserer Recherche haben wir ein interessantes Tool gefunden, EvoSuite. Mithilfe von EvoSuite[11] können aus Java Klassen Testfälle mit Assertions generiert werden.[39]

A.8.2 Integrationtesting

Integrationstest überprüfen, ob die unabhängig voneinander entwickelten Units richtig miteinander agieren[40]. Es geht darum das System in einem grösseren Rahmen zu Testen, als bei den Unittests.

In der folgenden Liste sind einige Best Practices zu den Integrationstests zusammanengestellt[41]:

- Integrations Tests sollten sich auf die Interaktion zwischen verschiedene Units/Komponenten fokussieren.
- Integrations Tests sollten während dem Development ausgeführt werden. Je früher das Feedback, desto besser.
- Möglichst reale Ressourcen sollten für die Integrationtests verwendet werden und die Tests sollten unabhängig von der Umgebung sein.

11 https://www.evosuite.org/

A.8.2.1 Tool ArchUnit

ArchUnit[12] ist eine Library mit der Integrationstest ausgeführt werden können, um genauer zu sein Architekturtests. Mithilfe von ArchUnit können Abhängigkeiten zwischen Packages und Klassen, Layers und Slices auf zyklische Abhängigkeiten und mehr überprüft werden.[42]

A.9 Lessons Learned

Durch die Analyse hat sich gezeigt, dass DDD und BDD bereits kombiniert verwendet werden. Es gibt jedoch keine klare Vorgaben oder Tools, die dies unterstützen. Ebenfalls hat sich herausgestellt, das BDD nicht nur von Specification by Example lebt sondern auch vom Dialog. Diese Formen von BDD werden durch Example Mapping und Feature Mapping spezifiziert. Ebenfalls hat sich ergeben, dass die Generierung von Tests mehrfach untersucht wurde. EvoSuite bietet ein Tool, dass anhand des Sourcecodes die Unittest generiert. Zusammenfassend konnte festgestellt werden, dass es keine einheitliche Form gibt wie DDD und BDD kombiniert werden kann und es keine Toolunterstützung in diesem Bereich gibt. Die meisten Anwendungsfälle von BDD konzentrieren sich auf einen bestimmten Anwendungszweck und werden nicht während des ganzen Softwareprozesses angewendet. Es gibt noch keine ganzheitliche Integrierung von BDD im Softwareprozess.

12 https://www.archunit.org/

B Anforderungsspezifikationen

B.1 Einführung

B.1.1 Zweck

Das vorliegende Dokument spezifiziert die Anforderungen von *DDD-2-BDD*.

B.1.2 Gültigkeitsbereich

Das Dokument beschränkt sich auf die Projektdauer der Bachelorarbeit im FS2021.

B.1.3 Übersicht

- Allgemeine Beschreibung
- Personas
- Use Cases
- Nichtfunktionale Anforderungen
- Beschreibung der Schnittstellen zu Umsystemen

B.2 Allgemeine Beschreibung

B.2.1 Produkt Funktion

Das Produkt DDD-2-BDD soll anhand von einem Domain Driven Design (DDD) Model Akzeptanztest generieren. Das DDD Modell muss als Context Mapper Language (CML) File vorhanden sein um eine Generierung zu ermöglichen. Die Entscheide bezüglich der Werkzeugwahl sind in der Software Architektur und Design D dokumentiert. Die Akzeptanztest und die Zielsprache ist Gherkin. Das Produkt soll für Software-Entwickler innerhalb von 5-10 Minuten erlernbar sein, gemäss der Aufgabenstellung. Auch sollte es robust und

benutzerfreundlich auf mangelnde CML Modelle reagieren. Die generierten Tests streben eine hohe Testabdeckung an, um einen Vorteil im Vergleich von manuellen geschriebenen Test zu garantieren. Die Generierung der Tests werden in einem separatem Dokument aufgeführt und ist im Anhang G.3 zu finden.

B.2.2 Benutzer Charakteristiken

DDD-2-BDD richtet sich besonders an Software Entwickler, die bereits Erfahrungen mit DDD und Akzeptanztest haben. Mehr dazu ist im Abschnitt Personas zu finden.

B.3 Personas

Mit unserer Applikation möchten wir drei unterschiedliche Benutzergruppen ansprechen. Die erste Personengruppe hat keinen Bezug zu DDD und arbeiten in der allgemeinen Entwicklung. Die zweite Gruppe betreibt hauptsächlich Objektorientierte Modellierung und die dritte Gruppe sind DDD-Experten. Jede der drei Personengruppe wird in einer Person beschrieben.

B.3.1 Software Entwickler Patrick

Patrick ist Software Entwickler in einer kleinen Firma. Er ist kein Fan von Software Modellierung und würde am liebsten schon am Anfang eines Projektes mit dem Programmieren beginnen. Sein Team arbeitet mit dem Context Mapper und DDD. Patrick kennt sich jedoch nicht gut aus mit DDD und er findet es sehr kompliziert, auch von den Agilen Praktiken ist er nicht so überzeugt und begeistert. Trotzdem nutzt er CML um besonders die Eigenschaften der Datentypen zu definieren und mit den Kunden zu besprechen, wie beispielsweise Range Checks. Er fände es nützlich, wenn er die Testfälle direkt generieren lassen könnte im CML.

B.3.2 Teamleiterin Entwicklung Lea

Lea ist Teamleiterin von einem zehnköpfigem Entwicklungsteam. Das ganze Team hat Erfahrung mit DDD und nutzt aus diesem Grund den Context Mapper aktiv. Lea fokussiert sich jedoch mehr auf die Objektorientierten Domänen. Sie ist der Meinung, dass das Herzstück von der Software durch die Objektorientierte Entwicklung eine höhere Qualität hat. Lea arbeitet seit geraumer Zeit mit den Agilen Praktiken. Sie verlässt sich im Bezug auf die Agilen Praktiken, jedoch auf den Experten in ihrem Team. Ihr ist wichtig, dass die Assoziationen zwischen den Objekten lückenlos dokumentiert sind. Sie fände es ebenfalls hilfreich, wenn sie die definierten Eigenschaften jeweils direkt im Code auch wieder überprüfen kann.

B.3.3 Software Architekt Fabio

Fabio ist ein erfahrener Software Architekt. Er arbeitet seit zehn Jahren mit DDD. Seit einiger Zeit nutzt er auch den Context Mapper. Er orientiert sich sehr streng an den DDD Patterns und möchte jeweils den Code überprüfen, ob er auch den DDD Patterns entspricht. Zusätzlich ist im wichtig, falls eine solche Funktion vorhanden ist, dass er genau weiss, was im Hintergrund passiert. Er findet das die Dokumentation ein wichtiger Teil ist im Software Bereich. Fabio arbeitet seit über zehn Jahren mit den Agilen Praktiken und kennt sich damit bestens aus. Ebenfalls ist er ein Fan von Open Source Projekten.

B.4 Use Cases

Bei der Erstellung der Use Cases haben sich zwei Hauptaktivitäten herauskristallisiert. Zum einen das Generieren von Gherkin Test anhand des CML Models und zum anderen das Integrieren der Gherkin Test im Software Engineering Prozess. Beim Generieren der Gherkin Test handelt es sich um die Use Cases der Applikation und bei integrieren in den Workflow Prozess geht es eher um die ergänzenden Anleitungen und Dokumentationen.

B.4.1 Acitivity 1: create Gherkin Aceptance Tests with CML-Model

Ein Benutzer hat sein DDD Modell in CML abgebildet und möchte nun mit *DDD-2-BDD* Gherkin Testszenarien generieren. *DDD-2-BDD* generiert die Testszenarien und gibt dem Benutzer ein generiertes Gherkin File zurück. Dem Benutzer steht ebenfalls ein Log-File zur Verfügung um die Generierung zu überprüfen. Bei Fehlern wird dem Benutzer eine sinnvolle Rückmeldung gegeben. Die Use Cases wurden drei User Stories zugeordnet. Jede User Story hat einen Actor. Im Listing ist das CML File mit den User Stories zu sehen. In den Kommentare wurde vermerkt, ob die Use Cases minimal, target oder outstanding sind.

Listing B.1: Use Cases im CML Format

```
UserStory testValidationsOfAttributes{
    As a "Software␣Developer"
    I want to "ensure" an "Attribute" with its "Maximum" //minimal
    I want to "ensure" an "Attribute" with its "Minimum" //minimal
    I want to "ensure" an "Attribute" with its "Range" //minimal
    I want to "check" an "Attribute" with its "Length" //target
    I want to "verify" an "Attribute" with its "Pattern" //target
    I want to "assert" an "Attribute" for "immutable" //target
    I want to "assert" an "Attribute" for "changeable" //target
    I want to "assert" an "Attribute" for "required" //target
    I want to "assert" an "Attribute" for "nullable" //target
    I want to "ensure" an "Collection" with its "SizeMaximum" //target
```

```
    I want to "ensure" an "Collection" with its "SizeMinimum" //target
    I want to "ensure" an "Collection" with its "SizeRange" //target
    so that "I can check whether the
    attributes have been mapped correctly"
}
UserStory testAssociations{
    As a "Lead Developer"
    I want to "test" an "Association" for "1:1 Relationship" //minimal
    I want to "confirm" an "Aggregation" for "1:n Relationship" //target
    I want to "confirm" a "Komposition" for "1:n Relationship" //target
    I want to "examine" a "Duplication" for "references" //target
    I want to "test" an "Asscoiation" for "n:m Relationship" //outstanding
    I want to "test" an "Association" for "bidirectional
    n:m Relationship" //outstanding
    I want to "examine" an "Idempotence" for "operations" //outstanding
    I want to "test" an "Inheritance" //outstanding
    so that "I can ensure whether the associations
    have been applied correctly"
}
UserStory testDDDPatterns{
    As a "Domain Expert"
    I want to "check" a "Value_Object" for "Identity" //minimal
    I want to "check" an "Entity" for "Identity" //minimal
    I want to "control" an "Lifecycle" for a "Aggregate" //target
    I want to "control" an "Service" for "Statelessness" //outstanding
    so that "I can check whether the
    DDD patterns have been applied correctly"
}
```

B.4.1.1 User Story 1: ensure Validations of Attributes

Der Benutzer möchte die Attribute auf definierte Eigenschaften überprüfen. Vorausgesetzt ist, dass der Benutzer die Eigenschaften im CML angegeben hat. Dem Nutzer werden für die Attribute entsprechenden Test Szenarien zurückgeben. In der Tabelle sind die möglichen Eigenschaften aufgelistet sortiert nach den Landingzones.

Landing Zone	Attribut Eigenschaften
Minimal Goal	Minimum, Maximum, Range
Target	Length, Pattern, required, nullable, changeable, immutable, Collection: Minimum, Maxium, Range
Outstanding	–

B.4.1.2 User Story 2: test Associations

Der Benutzer möchte kontrollieren, ob die Assoziationen zwischen den verschiedenen Objekten richtig implementiert wurden. Es werden von der Applikation Testszenarien generiert, die die Beziehungen zwischen den Objekten testen.

B.4.1.3 UC02.1: test 1:1 Aggregations

Der User möchte testen, ob die 1:1 Aggregation richtig implementiert worden sind. Der DDD-2-BDD Prototyp generiert die hierfür notwendigen Tests.

B.4.1.4 UC02.2: validate Aggregation behaviour for 1:n relationships

Der Benutzer hat eine 1:n Beziehung modelliert mit einer Aggregation. Er möchte mithilfe der Testszenarien sicherstellen, dass sie korrekt implementiert wurde. Es werden ihm Testszenarien zurück gegeben, die testen, ob bei dem Löschen des einen Objekts, die referenzierten Elemente vorhanden bleiben.

B.4.1.5 UC02.3: confirm Komposition behaviour for 1:n relationships

Der User hat mit einer Komposition eine 1:n Beziehung modelliert. Um die Komposition zu überprüfen, möchte er Testszenarien generieren lassen. Ihm werden Testszenarien zurückgegeben, die überprüfen, ob beim Löschen des eines Objekt die referenzierten Elemente ebenfalls gelöscht werden.

B.4.1.6 UC02.4: test for duplications in reference collections

Der Benutzer möchte überprüfen, ob die Inhalte von einer Liste nicht identisch sein können. Beispielsweise, wenn er bei einem Essay Objekt eine List von Autoren hat, dann will er sicherstellen, dass ein Autor nicht zweimal in der Liste erfasst sein kann. Die Applikation gibt ihm die notwendigen Testszenarien zurück.

B.4.1.7 UC02.5: test Association for n:m relationship

Der User hat eine n:m Beziehung modelliert und möchte überprüfen, ob die Beziehung korrekt implementiert wurde. Die Applikation gibt ihm Tests zurück, die die Implementierung testen.

B.4.1.8 UC02.6: test Association for bidirectional n:m relationship

Der User hat eine bidirektionale n:m Beziehung modelliert und möchte überprüfen, ob die Beziehung korrekt implementiert wurde. Die Applikation gibt ihm Tests zurück, die die Implementierung testen.

B.4.1.9 UC02.7: examine idempotence for operations

Der Anwender hat seine Operationen/Methoden als Idempotenz bestimmt und möchte nun überprüfen, ob dies auch der Programm Logik entspricht. Die Testszenarien überprüfen, ob das mehrfache ausführen dieser Operation/Methode immer zum selben Ergebnis führt.

B.4.1.10 UC02.8: test inheritance

Der User hat eine Vererbung in seiner Applikation modelliert. Er möchte nun Testszenarien generieren, die ihm sinnvolle Test ausgeben um die Logik zu überprüfen.

B.4.1.11 User Story 3: check DDD patterns

Der Benutzer möchte überpüfen, ob er die DDD Patterns korrekt angewendet hat.

B.4.1.12 UC03.1: check Identity of Entities

Der Benutzer möchte, dass seine modellierten Entities auf Uniqueness überprüfen. Falls eine *create* Methode bei der eine Id mitgegeben wird, dann wird ein Testszenario generiert bei dem unter anderem überprüft wird, ob das Erstellen von Objekten mit gleichen Id einen Fehler ausgibt. Falls kein *create* Methode vorhanden ist oder sie keine Id verwendet, wird überprüft, ob das Erstellen von Entities mit gleichen Attributen unterschiedliche Objekte erstellt.

B.4.1.13 UC03.2: check Identity of Value Objects

Der Benutzer will überprüfen, ob die Value Objects keine Identität haben. Um sicherzustellen, dass die Value Objects keine Identität haben wird die equals Methode verwendet. Dem Benutzer werden Testszenarien zurückgegeben, in denen überprüft wird, ob zwei Elemente mit gleichen Attributen auch gleich (equals()) sind.

B.4.1.14 UC03.3: check Lifecycle of Aggregate

Der Benutzer will kontrollieren, ob der Lifecycle vom Aggregate richtig umgesetzt wurde. Vorausgesetzt ist, das das Aggregate verschiedene States hat und die Stateänderungen bei den Operationen modelliert sind. Es wird anhand von den verschiedene States ein Statediagramm erstellt und Testszenarien generiert, die jeden möglichen Pfad im Statediagramm abdecken.

B.4.1.15 UC03.4: control if Service is stateless

Der Benutzer möchte überprüfen, ob das implementierte Service Pattern, stateless verwendet wird. Das Programm generiert Testszenarien, die überprüfen, ob auf dem Service kein State vorhanden ist.

B.4.2 Activity 2: integrate Gherkin Tests in the Software Engineering Work Flow

Der Benutzer hat mit seinem CML Modell die Gherkin Testfälle generiert. Nun möchte er mehr darüber wissen, wann es am sinnvollsten ist diese Testszenarien zu verwenden und wie er sie in einem Projekt integrieren kann. In der folgenden Liste sind, die Fragen aufgeführt, die innerhalb von der Applikationsdokumentation beantwortet werden sollen.

- Zu welchem Zeitpunkt, kann ich das DDD-2-BDD Tool im Projekt integrieren?
- Wie werden meine Testszenarien generiert?
- Sind die Testszenarien vollständig? Wie hoch ist die Testabdeckung?
- Wie kann ich die Testszenarien in meinem Projekt integrieren und automatisieren?
- Welcher Vorteil bringt mir die Verwendung von DDD-2-BDD?
- Wie kann man das Tool ohne DDD verwenden?

B.5 Weitere Anforderungen

Qualitätsmerkmale

Die Zusammenstellung der Anforderungen der Qualitätsmerkmale basiert auf dem Produktqualitätsmodel aus ISO/IEC 25010:2011.

B.5.1 Funktionale Stabilität

- Funktionale Vollständigkeit: Die Use Cases von diesem Dokument werden vollständig umgesetzt. Bei den Use Cases bei denen Landing Zones angegeben sind, streben wir die Landingzone Target an. Falls genügend Zeit vorhanden ist, soll die Landingzone Oustanding umgesetzt werden. Ebenfalls soll durch die Generierung von den Testszenarien ein Vorteil im Vergleich zum manuellen Testen vorhanden sein und dem Nutzer einen Mehrwert bieten.
- Funktionale Korrektheit: Die Software soll für korrekte CML Modelle syntaktisch korrektes Gherkin generieren. Ob die Generierung korrekt ist, wird anhand von Beispielen überprüft. Die Testbeispiele sind in einem separaten File aufgeführt.

B.5.2 Performance

Wir streben eine möglichst kurze Verarbeitungszeit an. Die Generierung der Testszenarien soll nicht mehr als drei Minuten dauern, damit der Benutzer nicht zu lange auf die Generierung warten muss, bei einem CML Input von 700 Zeilen. Die Zeilen sind auf 700 fixiert weil LakesideMutual Beispiel ungefähr dieser Grösse entspricht und dies der erwartete Input ist.

B.5.3 Benutzerfreundlichkeit

- Erlernbarkeit: Die Software soll ohne Vorkenntnisse in CML und BDD innerhalb von maximal 10 Minuten in ihren Grundfunktionen erlernbar sein, sodass bereits mindestens die ersten Testszenarien generiert werden können.
- Nutzerfehlerschutz: Es können nur Files verarbeitet werden, die korrektes CML enthalten. Dem Nutzer wird innerhalb von 20 Sekunden eine Meldung gegeben, wenn das File nicht korrekt ist.
- Bedienbarkeit: Die Software soll übersichtlich gestaltet und logisch aufgebaut sein, um eine intuitive und einfache Bedienung zu ermöglichen. Ein neuer Benutzer soll sich innerhalb von 2 Minuten zurechtfinden können.

B.5.4 Sicherheit

Es soll sichergestellt werden, das keine vertraulichen Daten über unverschlüsselte Kanäle transportiert werden.

B.5.5 Portabilität

Die Software soll auf den gängigen Betriebssystemen (Ubuntu, Windows, Mac) einsetzbar sein, allenfalls unter Verwendung von weiteren Softwarekomponenten.

B.5.6 Modularität und Erweiterbarkeit

Die Software soll modular aufgebaut werden und in einzelne logische Komponente aufgeteilt sein, sodass die Software einfach und mit minimalem Aufwand auf neue CML-Notationen und Testgenerierungs-Konzepte erweitert werden kann. Zur Erreichung dieses Qualitätsmerkmals werden verschiedener möglichen Änderungsszenarien definiert und eine maximale Bearbeitungszeit beziehungsweise Umsetzungszeit in Entwicklungsstunden angegeben.

Szenario	**maximale Bearbeitungszeit [h]**
Es soll ein neues Testgeneriungs Konzept implementiert werden mit gleichbleibenden Funktionen vom Context Mapper.	20
Es soll die Unterstützung für ein neues CML Feature implementiert werden.	10-20
Es soll ein neues Testgeneriungs Konzept implementiert werden mit neuen Funktionen vom Context Mapper.	30-40

Die Codebasis soll entsprechend dieser Änderungsszenarien strukturiert sein, sodass die genannten Zielzeiten realistisch umsetzbar sind. Zur Sicherstellung dieses Qualitätsmerkmals wird der Ansatz des **Codereviews während des Projektverlaufs** herangezogen. So werden Gegenseite Code-Reviews bei jedem Merge Request durchgeführt.

B.5.7 NFR Spezifikationen

Die Quality Attribute Scenario Templates ergänzen und spezifizieren einige der genannten Qualitätsmerkmale.

Scenario für DDD-2-BDD, NFR 1		
Scenario(s)		Erstnutzung des Services
Business Goals(s)		Kleine Einstiegshürde, frühe Erfolgserlebnisse die zu Nutzerzufriedenheit und Kundengewinnung führen
Relevant Quality Attributes(s)		Erlernbarkeit, Bedienbarkeit
Scenario Components	Stimulus	Nutzer testet den Prototyp mit dem Ziel ein Testszenarien zu generieren von einem CML Modell.
	Stimulus Source	Neuer Enduser
	Environment	Zur Laufzeit, normale Betriebsbedingungen, keine erhöhte Last.
	Artifact	Alle Komponenten und Schnittstellen auf den verschiedenen logischen Layern, die für die Testgenerierung notwendig sind
	Response	Der User lernt die Applikation kennen und kann für sein CML File Testszenarien generieren.
	Response Measure	Der User kann sich innerhalb von 10 Minuten einen Überblick von der Applikation verschaffen und die ersten Testszenarien generieren lassen.
Questions		-
Issues		abhängig von der Performance, Durchlaufzeit

Scenario für DDD-2-BDD, NFR 2		
Scenario(s)		Testgenerierungs Konzept
Business Goals(s)		Nutzen der Applikation steigert sich, Vorteil zu manuell geschriebenen Akzeptanztests
Relevant Quality Attributes(s)		Funktionale Vollständigkeit
Scenario Components	Stimulus	Nutzer möchte Testszenarien generieren
	Stimulus Source	Testszenarien Generierung
	Environment	Zur Laufzeit, normale Betriebsbedingungen, keine erhöhte Last.
	Artifact	Testkonzepte(Testbeispiele), Implementation der Generierung
	Response	Der User erhält Testszenarien, die schneller generiert wurden und eine höhere Testabdeckung und Aussagekraft haben, im Vergleich zu seinen manuell geschriebenen Test.
	Response Measure	Für alle Use Cases werden die Sunny Day, Rainy Day und Stormy Day Szenarien generiert
Questions		Wie erreichen wir eine hohe Testabdeckung und Aussagekraft?
Issues		abhängig von Benutzerinput

B.6 Schnittstelle: Context Mapper

Als externe Schnittstelle verwenden wir die Context Mapper Library[1]. Im CML werden wir hauptsächlich die Taktischen DDD Elemente verwenden. Zum Einsatz werden sicher Aggregates, Entities und ValueObjects kommen. Bei den Aggregate ist der AggregateLifecycle und die State Tranisitions[2] sicherlich relevant. Ebenfalls kommt die Taktische DDD Syntax vom Sculptor Generator zum Einsatz, die von CML unterstützt wird. In der Sculptor Syntax sind unter anderem Optionen möglich wie Valdiation, Cascadation und Inheritance. Mit der Sculptor Syntax können Objekt Attribute genauer spezifiziert werden mit den Keywords: nullable, required, immutable, changeable. [3]

B.7 Fazit

Dieses Dokument zeigt auf, in welche drei Benutzergruppen die einzelnen Use Cases aufgeteilt werden können. Ebenfalls wird durch die erstellten Fragen, den Leitfaden für die Tutorials und Benutzeranleitungen vorgegeben. Die nicht funktionalen Anforderungen ergeben wichtige Vorgaben für die Architekturentscheidungen. Die Beschreibung der externen Schnittstellen geben bereits in der frühen Projektphase wichtige Hinweise auf mögliche Hindernisse und Probleme bei der Entwicklung. Zudem bietet dieses Dokument die Grundlage für die Erstellung der System-, Performance- und Usabilitytests.

1 https://www.javadoc.io/doc/org.contextmapper/context-mapper-dsl/latest/index.html
2 https://contextmapper.org/docs/aggregate/
3 http://sculptorgenerator.org/documentation/advanced-tutorial

C Domainanalyse

C.1 Einführung

Ziel des DDD-2-BDD Projekts ist es, die beiden Arbeitsmethoden DDD (Domain Driven Design) und BDD (Behaviour Driven Design) zu kombinieren. Dafür wird der Prototyp DDD-2-BDD entwickelt. Dieses Dokument widmet sich der Domainanalyse des DDD-2-BDD Prototyps und somit der Analyse der Domänen DDD, BDD und der Kombination dieser Zwei. Das hierbei erarbeitete bessere Verständnis der Domänen soll nachfolgend genutzt werden, um ein Transferkonzept zu erstellen, welches Information aus der DDD Domäne in die BDD-Domäne überführt.

C.1.1 Zweck und Ziel

Ziel der Domainanalyse ist sowohl das Erkennen und Verstehen der elementaren Geschäftsregeln der Domäne als auch Anforderungen und Möglichkeiten für Transferkonzepte zu identifizieren. Dabei wird ein Domänenmodell erstellt, welches dabei hilft, ein gemeinsames Verständnis aller Projektbeteiligten über die zugrundeliegenden Geschäftsregeln des Softwareprojekts zu schaffen.

C.1.2 Gültigkeitsbereich

Die Gültigkeit dieser Domainanalyse beschränkt sich auf die Laufzeit der Bachelorarbeit während des Frühlingssemesters 2021.

C.1.3 Limitationen

Im Bereich BDD beschränkt sich dieses Dokument auf die BDD-Testingframeworks. Example/Feature Mapping werden bewusst nicht betrachtet, da es sich bereits abgezeichnet hat, dass die Umsetzung beziehungsweise Integration dieser Elemente in den Prototyp den Umfang der Bachelorarbeit sprengen würde.

C.1.4 Beschreibung der Domänen und Einschränkung auf Tools

DDD ist eine Philosophie für die Softwareentwicklung, welches das Denken in Domänen während den verschiedenen Softwareentwicklungsschritten fördert. Das Denken in Domänen hat zur folge, dass die Strukturen und die sprachlichen Elemente des Codes sich nach der abzubildenden Geschäftsdomäne richten. Dabei initiiert DDD einen kreativen und kollaborativen Prozess zwischen technischen und domänenspezifischen Experten, welcher iterativ das konzeptionelle Modell erschafft, um so die Geschäftsdomäne und ihre Problemstellungen optimal abzubilden. Bei BDD handelt es sich ebenfalls um eine Arbeitsmethode, welche die Zusammenarbeit zwischen Entwicklern und Domänenexperten fördert. Hierbei steht jedoch nicht Domäne, sondern das erwartete Verhalten der Software im Fokus. Für beide Domänen gibt es bereits Tools, welche die Verwendung der entsprechenden Praktiken unterstützen. Während diese Tools auf auf der DDD Seite eher optionaler Natur sind, wird bei BDD davon ausgegangen, dass ein entsprechendes BDD-Framework verwendet wird. Für dieses Projekt gehen wir davon aus, dass auf der Seite DDD der Context Mapper [1] zum Einsatz kommt und auf der BDD Seite das BDD-Framework Cucumber eingesetzt wird. Die Entscheidungen bezüglich der Werkzeugwahl sind in der Software Architetkur zu finden.

C.1.5 Überschneidung der Domänen und Nutzung in einem Projekt

Das DDD-2-BDD Projekt soll eine Einweg-Brücke zwischen den beiden Domänen, ausgehend von der DDD-Domäne und endend in der BDD-Domäne, darstellen. Das folgende Diagramm bietet eine Übersicht über die Domänen, ihre Elemente und die Überschneidungen.

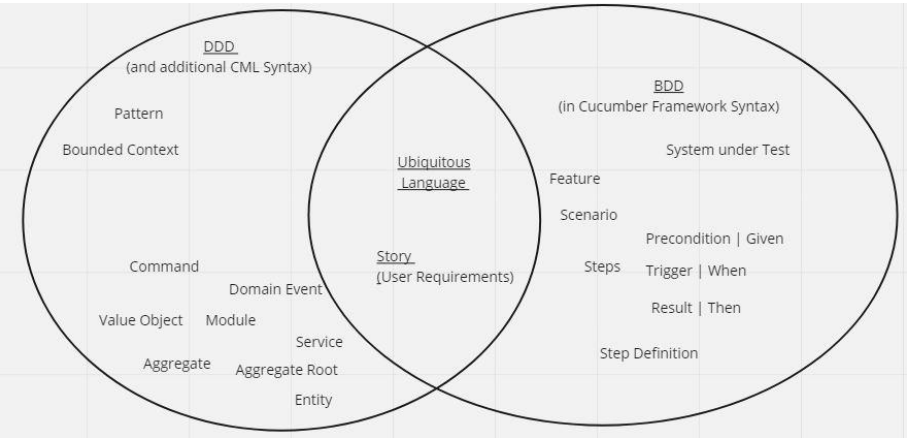

Abbildung C.1: Venn Diagramm der Domänen DDD und BDD

Durch die Grafik C.2 wird offensichtlich, dass die Userstory und die Ubiquitous Language in beiden Domänen vorkommt, wodurch sie für den Transfer eine besondere Wichtigkeit erhalten.

1 https://contextmapper.org/

Hierbei ist zu beachten, dass das Konzept User Story beziehungsweise User Requirement nicht Teil von DDD ist, jedoch Teil des CML Syntax. Bei der Ubiquitous Language handelt es sich um ein Konzept, welches seinen Ursprung in der DDD Bewegung hat und von der BDD Community adaptiert wurde. In beiden Communitys handelt es sich dabei um die Idee, dass in einem Softwareprojekt von allen beteiligten die gleichen Begriffe für bestimmte Elemente verwendet werden sollen, um Ungenauigkeiten in der Kommunikation zu vermeiden.
In einem weiteren Schritt wird nun das obige Diagramm um eine weitere Domäne ergänzt. Diese neue Domäne steht substitutiv für die zu entwickelnde Software in einem arbiträren Softwareprojekt. Die Idee dahinter ist, dass DDD für Architektur und Design der Software benutzt wird, während die Software zeitgleich das SUT(System under Test) der BDD-Domäne darstellt. Für dieses Schaubild wird mit Begriffen aus Objekt Orientierten Programmiersprachen (OOP) gearbeitet. Das dargestellte Konzept ist jedoch nicht OOP spezifisch und kann auch zusammen mit funktionaler Programmierung angewendet werden.

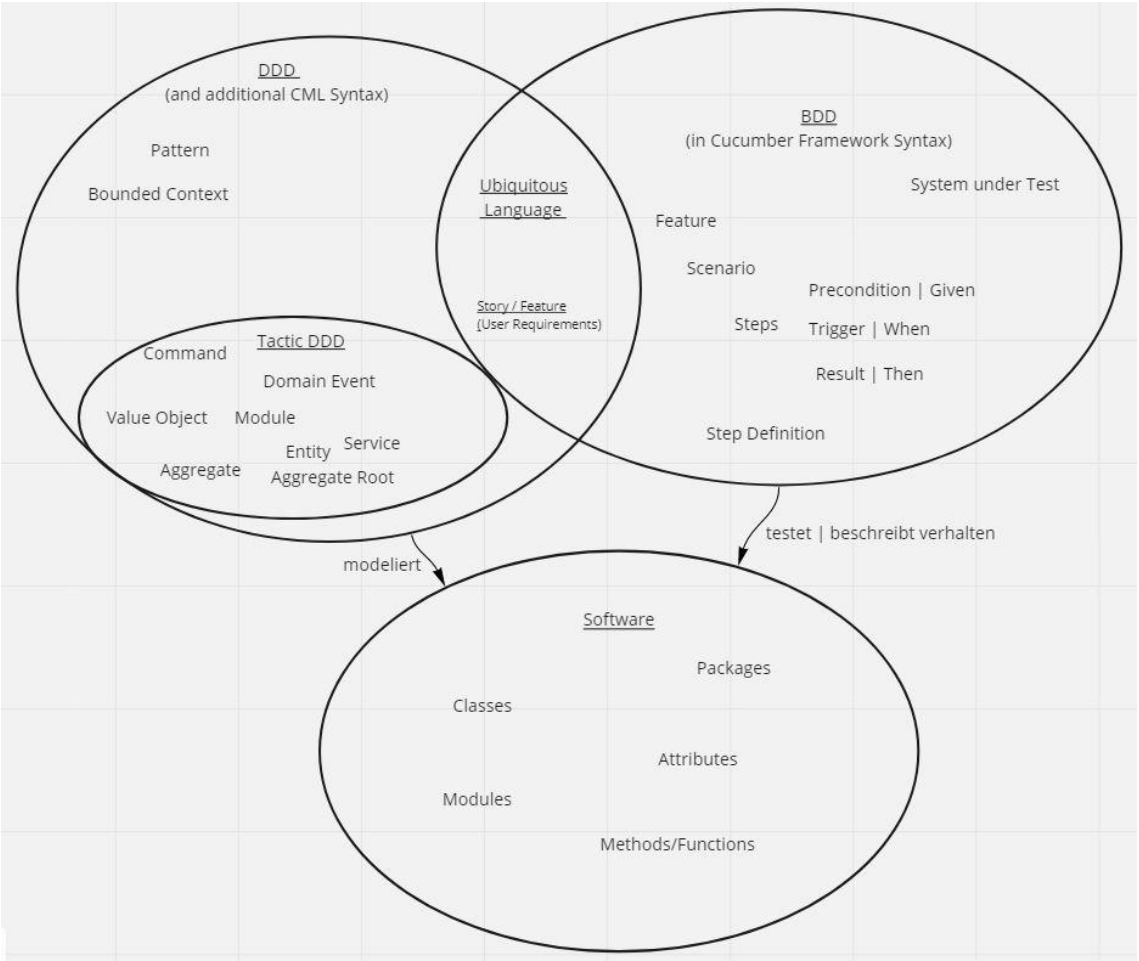

Abbildung C.2: Venn Diagramm der Domänen DDD, BDD und arbiträrer Software

Für die Modellierung der Software kommt Taktisches DDD zum Einsatz. Das geschaffene

Modell besteht hauptsächlich aus Modules, Aggregates, Entities, ValueObjects und Services. Eine Beschreibung dieser Konzepte erfolgt in der Erklärung des Domänendiagramms. Das mit taktischem DDD modellierte Modell beschreibt die Softwarestruktur, nun gilt es diese Softwarestruktur zu testen und damit sicherzustellen, dass die Software auch das vom Kunden gewünschte Verhalten hat. Hierfür kommt BDD zum Einsatz. Die BDD-Szenarien werden in einem herkömmlichen Softwareprojekt manuell in Zusammenarbeit mit Domänenexperten/-Kunden erstellt. Die manuelle Erstellung der BDD-Testszenarien ist zeitaufwendig. Das Ziel des Prototyps ist es daher, diesen zeitaufwändigen Prozess in Teilen zu automatisieren, indem anhand des taktischen DDD Models einige Testszenarien generiert werden. Die generierten Szenarien können dann genutzt werden, um das Softwareverhalten mit dem Kunden zu besprechen und gegebenenfalls weiter zu spezifizieren. Hierbei ist anzumerken, dass es sich bei den generierten Szenarien, wirklich nur um Szenarien handelt. Die Step Definitions, welche benötigt werden um diese Testszenarien auszuführen, werden nicht generiert.

Das folgende Entscheidungsdiagramm zeigt auf, wie der DDD-2-BDD Prototyp beispielhaft in einem Softwareprojekt genutzt werden kann.

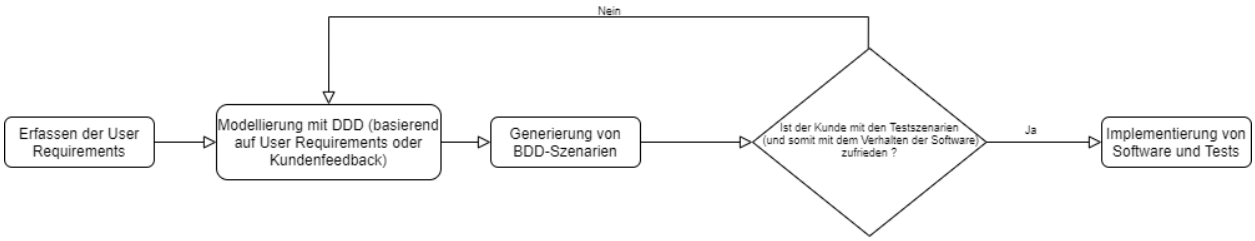

Abbildung C.3: Entscheidungsdiagramm zur Verwendung des DDD-2-BDD Tools

Abbildung C.4 beinhaltet ein Beispiel ,an dem die Dreiecksbeziehung zwischen DDD Modell, BDD-Testszenarien und Implementation erkannt werden kann. Die Java Klasse Author bildet das gleichnamige Entity aus dem CML Modell ab. Das in der Abbildung oben rechts gezeigte Featurefile mit den zwei Testszenarien überprüft, ob das im CML spezifizierte Verhalten von der Implementation korrekt umgesetzt wird. Zu beachten ist, dass hier exemplarisch Java Code verwendet wurde, die Konzepte sind jedoch genauso auf andere Programmiersprachen anwendbar.

Context Mapper Modell (DDD Modell | Input)

```
Entity Author {
    String name length="150" required
    String class
    -List<@Essay> publicizedEssays cascade="persists" inverse
    -List<@Essay> pendingEssayList cascade="persists,remove"

    def @EssayId createEssay(@Essay essay) : write [ -> CREATED];
    def boolean addCoAuthor(@Author coauthor): write[CREATED -> CREATED];
    def boolean updateEssay(@Essay essay): write[CREATED -> CREATED];
    def boolean submitEssay(@Essay essay) : write [ CREATED, RETURNED-> SUBMITTED];
}
```

Gherkin Feature (BDD Testszenarien | Output)

```
Feature: Testing the 1:n composition for the entity Author
    # Story: Testing cascade: persists, remove

  Scenario: When deleting an Author, the corresponding elements of the collection pendingEssayList are deleted
    Given an entity Author with the references pendingEssayList
    When the entity Author is deleted
    Then the references pendingEssayList are deleted

  Scenario: When deleting an Essay, the corresponding Essay in the collection pendingEssayList is removed.
    Given an entity Author with the references pendingEssayList
    When an element Essay in the collection pendingEssayList is deleted
    Then the element Essay is removed from the reference collection pendingEssayList
```

Implementation (Beispielhaft in Java | SUT)

```
public class Author {
    private String name; //required
    private List<Essay> pendingEssayList = new LinkedList<>(); // cascade="persists,remove"
    private List<Essay> publicizedEssays = new LinkedList<>(); //cascade="persists" inverse
    private List<Essay> allEssays = new LinkedList<>();
```

Abbildung C.4: Beispiel für Dreiecksbeziehung zwischen DDD Modell, BDD-Testszenarien und Implementation

C.2 Domainmodel

C.2.1 Domain Diagramm

Abbildung C.5 entspricht dem erarbeiteten Domänenmodell. Besondere Relevanz haben hierbei das taktische DDD und die Story/Feature Elemente, weil diese eine Transferierung des DDD (beziehungsweise genauer CML) Inputs in BDD Output in Form von Gherkin Szenarien ermöglichen. Strategisches DDD kommt nicht zum Einsatz. Die Verwendung von Strategischen DDD Patterns und somit der Information, wie die Domäne in einzelne Bounded Contexte aufgeteilt werden und wie sie miteinander Interagieren, wäre zwar für die Erfassung des inter-Bounded-Context Verhaltens elementar, würde aber den Umfang der Bachelorarbeit sprengen. Dies gilt ebenso für die Elemente für Application and Process Layer[2] Modellierung. Nachfolgend werden die Konzepte des taktischen DDD, die User Requirements und die

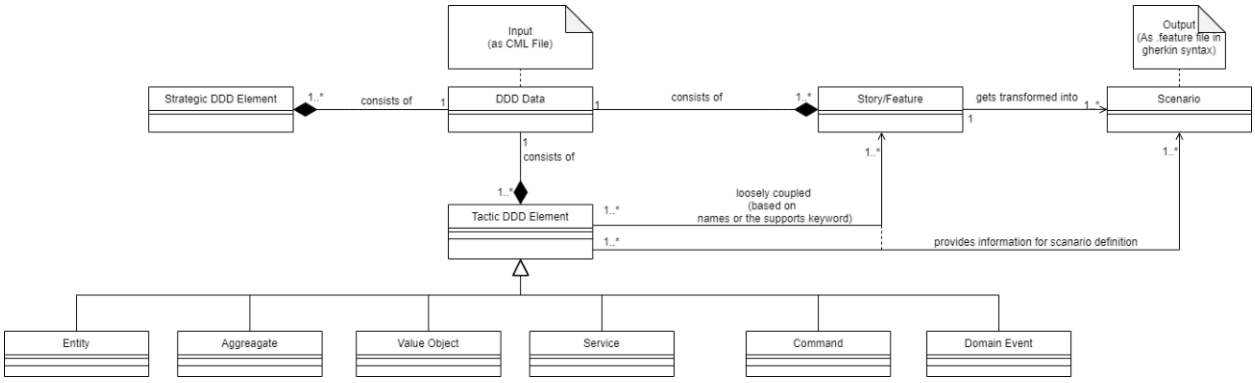

Abbildung C.5: Domainmodell des DDD-2-BDD Projekts

2 https://contextmapper.org/docs/application-and-process-layer/

Gherkin Szenarios genauer spezifiziert.

C.2.2 User Requirement

Der Context Mapper unterstützt das Erfassen von User Requirements sowohl mit User Stories als auch Use Cases, unser Prototyp beschränkt sich jedoch auf die Verarbeitung von User Stories. Dieser Entscheid basiert darauf, dass BDD (aufgrund des Agilen Hintergrunds) eher mit User Stories arbeitet als mit UseCases.

C.2.2.1 User Story

User Storys in CML werden mit folgendem Syntax[3] definiert:

```
UserStory US1_Example {
  As an "Insurance Employee"
    I want to "create" a "Customer" with its "firstname", "lastname" // attributes are optional ('with its' part)
    I want to "update" an "Address" for a "Customer" // reference is optional ('for a' part)
    I want to "offer" a "Contract" for a "Customer" // reference is optional ('for a' part)
  so that "I am able to manage the customers data and offer them insurance contracts."
}
```

Abbildung C.6: Syntax für UserStory in CML

In BDD können Features über einen Kommentar um eine User Story erweitert werden, dies ist jedoch optional und wird in der Praxis unterschiedlich gehandhabt.

C.2.3 Taktisches DDD

Nachfolgend werden die Elemente des Taktischen DDD kurz spezifiziert. Für eine ausführliche Spezifikation verweisen wir auf die Sculptor DSL [4] Die Elemente des Taktischen DDD werden für die Modellierung der Domains und SubDomains[5] verwendet. Das folgende Beispiel aus der Contextmapper Documentation beschreibt die Syntax von Domains, Subdomains und ihre Verbindung zu den User Stories. Auf eine Erläuterung der Konzepte Domain Event und Command wird verzichtet, da diese für den in dieser Bachelorarbeit entwickelten Prototyp keine besondere Bedeutung haben. Besonderes Augenmerk ist hierbei auf das Schlüsselwort **supports** zu richten, welches eine Assoziation zwischen dem Subdomain und den beiden UserStories erstellt.

3 https://contextmapper.org/docs/user-requirements/
4 http://sculptorgenerator.org/documentation/advanced-tutorial
5 https://contextmapper.org/docs/subdomain/

```
/* Syntax example for Subdomain that supports specific user requirements: */
Domain Insurance {
  domainVisionStatement = "Insurance domain vision statement ..."

  Subdomain CustomerManagementDomain supports CreateCustomers, CreateCustomerAddresses {
    type = CORE_DOMAIN
    domainVisionStatement = "Subdomain managing everything customer-related."

    Entity Customer {
      String firstname
      String familyname
    }
  }

}

UserStory CreateCustomers {
  As an "Insurance Employee"
    I want to "create" a "Customer" with its "firstname", "lastname"
  so that "I am able to manage the customers data and offer them insurance contracts."
}

UserStory CreateCustomerAddresses {
  As an "Insurance Employee"
    I want to "create" an "Address" for a "Customer"
    I want to "update" an "Address" for a "Customer"
  so that "I am able to manage the customers addresses."
}
```

Abbildung C.7: Syntax Domänen und Subdomänen

C.2.3.1 Module

Module können genutzt werden, um zusammenhängende Konzepte und Aufgaben zu organisieren und um die Komplexität zu reduzieren.

C.2.3.2 Entity

Objekte mit einer Identität. Der Zustand des Objekts kann sich während seines Lebenszyklus verändern.

C.2.3.3 Aggregate

Ein Aggregate ist eine Gruppe von zusammengehörenden Objekten, welche als eine Einheit zu betrachten ist im Bezug auf Speicherung und Datenänderungen. Jedes Aggregate besitzt ein root-entity. Das root-entity ist das einzige von ausserhalb des Aggregates sichtbare Objekt.

C.2.3.4 ValueObject

Value Objects sind Objekte ohne eigene Identität. Sie sind typischerweise immutable.

C.2.3.5 Service

Nach Sculptor agiert ein Service als Service Layer um ein Domänenmodell. Es stellt ein definiertes Interface mit den von ausserhalb der Domäne zur Verfügung stehenden Operationen den Clients zur Verfügung.

C.2.4 Gherkin Szenario

In BDD beschreiben Szenarien das Verhalten der Software, sodass das Softwareverhalten auch mit nicht technisch versierten Personen besprochen werden kann. Zusätzlich lassen sich solche Szenarien auch als Basis für die Softwaretests verwenden. Daraus folgt, dass die Spezifikation der Softwarefunktionalität und dessen Verifizierung auf demselben Input basieren. Szenarien werden dabei in sogenannten Features gebündelt. Dabei beschreibt ein einzelnes Szenario einen expliziten Ablauf, den die Software durchlaufen soll, während mehrere Szenarien zusammen das Feature im Ganzen beschreiben. Alle Features zusammengefasst, spezifizieren die Softwarefunktionalität als Ganzes. Ein Szenario besteht grundsätzlich aus drei Elementen. Einer Liste von Vorbedingungen, jeweils annotiert durch das Schlüsselwort **Given**, einer Liste von eintreffenden Ereignissen jeweils annotiert durch das Schlüsselwort **When** und einer Liste von Ergebnissen annotiert durch das Schlüsselwort **Then**. Zusätzlich können die Schlüsselwörter Given, When und Then, wenn sie mehrfach in Folge auftreten, auch durch die Schlüsselwörter **And, But** und ***** ersetzt werden. Dies ist jedoch nur Syntactic Sugar um die Lesbarkeit zu verbessern. Die folgenden zwei Beispielszenarios[6] zeigen dies nochmals auf: Um ein Gherkin Szenario als ausführbaren Tests zu benutzen, muss das Verhalten der

```
Example: Multiple Givens
  Given one thing
  And another thing
  And yet another thing
  When I open my eyes
  Then I should see something
  But I shouldn't see something else
```

Abbildung C.8: Szenario Beispiel mit And/But

```
Scenario: All done
  Given I am out shopping
  * I have eggs
  * I have milk
  * I have butter
  When I check my list
  Then I don't need anything
```

Abbildung C.9: Szenario Beispiel mit *

einzelnen Steps (also alle Givens, Whens und Thens) definiert werden. Dies geschieht in einem Step Definition File. In einem Java Projekt wird zur Definition eines Step Verhaltens eine Methode entwickelt, welche mit einer @Annotion versehen wird. Dadurch ist es dem BDD-Framework möglich den Step auf die Methode zu mappen.

6 https://cucumber.io/docs/gherkin/reference/

C.3 Fazit

Durch die Analyse der drei Domänen konnte ein besseres Verständnis der Problemstellung und Geschäftsregeln erarbeitet werden. Weiter konnten die Elemente des Taktischen DDDs als Informationsträger, der für die Testgenerierung relevanten Informationen, lokalisiert werden. Sie dienen als Grundlage des Transfers. Das erarbeite Domänenmodell stellt die Grundlage für die weiteren Softwareengineeringentscheide dar.

D Softwarearchitektur und Design

D.1 Einführung

D.1.1 Zweck

Das vorliegende Dokument beschreibt die Softwarearchitektur und das Design des DDD-2-BDD Tools.

D.1.2 Gültigkeitsbereich

Das Dokument beschränkt sich auf das Projekt DDD-2-BDD und die Dauer der Bachelorarbeit im FS2021.

D.1.3 Übersicht

- Wahl der Werkzeuge
- Architektonische Ziele und Einschränkungen
- Logische Architektur
- Externe Schnittstellen
- Deploymentdiagramm

D.2 Wahl der Werkzeuge

Im ersten Abschnitt geben wir an, wieso wir CML als Domain Driven Design (DDD) Inputfile verwenden und warum als Output ein Behaviour Driven Developtment (BDD) Gherkin File zur Verfügung gestellt wird. Ebenfalls wird erläutert welche Technologie in den Tutorials verwendet wird.

D.2.1 DDD Input

Die Aufgabenstellung gibt vor, dass die Appliaktion einen beliebigen DDD Input verarbeiten muss. Wir haben uns dazu entschieden Context Mapper (CML)- Files zu verwenden und haben somit die Variante von freier DDD Modellierung und Sculptor Files verworfen. Diese drei Varianten haben sich anhand der Analyse als mögliche Inputvarianten ergeben. Das CML-File bietet uns den Vorteil, dass das Strategische und Taktische DDD unterstützt wird im Vergleich zum Scultpor File, dass nur Taktisches DDD unterstützt. Die freie Modellierung ist den beiden anderen Methoden klar im Nachteil, weil das Auslesen der Daten viel aufwändiger ist. Die Context Mapper Files bieten uns ein sehr breites Spektrum an Funktionen, schränken aber die Nutzergruppe markant ein, weil die Applikation nur von Benutzern des Context Mappers verwendet werden kann.

D.2.2 BDD Output

Durch die Aufgabenstellung war gegeben, dass die Applikation Tests generieren muss und der Output im Zusammenhang mit BDD stehen muss. Wir haben diverse BDD Tool getestet und deren Einsatz recherchiert. Unter anderem haben wir Cucumber und JBehave getestet, sowie Tools, die Funktionen fürs Example Mapping anbieten, wie CARDBOARDIT. Wir haben uns dazu entschieden die Testfiles in Form von Gherkin zu generieren, weil dies den Vorteil bringt, dass sie Technologie unabhängig sind. Wir haben uns gegen das Generieren von JBehave und Cucumber Tests entschieden, weil diese nur von den jeweiligen Frameworks genutzt werden können und es besteht die Möglichkeit die Gherkin Files ebenfalls zu verwenden mit den genannten Tools.

D.2.3 Automatisierungstool für Tutorials

Die Aufgabenstellung gibt im Backend vor, dass Java verwendet werden soll. Aus diesem Grund haben wir uns dazu entschieden, bei den Tutorials die Beispiele ebenfalls mit Java zu programmieren. Die Beispiele werden einmal mit dem Build Tool Gradle[1] und einmal mit dem Build Tool Maven[2] zur Verfügung gestellt. Die Build Tools wurden aufgrund ihrer Verbreitung gewählt[43]. Dadurch das Java verwendet wird, bietet sich für die Gherkin Automatisierung JBehave und Cucumber an. Wir entschieden uns Cucumber zu verwenden, weil Cucumber im Vergleich zur JBehave von mehreren Sprachen verwendet werden kann[44] und weil es weiter verbreitet ist als JBehave[45].

1 https://gradle.org/
2 https://maven.apache.org/

D.3 Architektonische Ziele, Entscheide und Einschränkungen

Dieses Kapitel beschreibt die Architektonischen Ziele und die Entscheidungen, die wir anhand der Anforderungen getroffen haben.

D.3.1 Applikationsform

Für die Applikationsform stehen drei Varianten zur Auswahl, ein Command Interface Line (CLI), eine Webapplikation oder die Integrierung im Context Mapper. Die Tabelle zeigt die Vor- und Nachteile von den drei Applikationsformen auf.

Variante	**Vorteile**	**Nachteile**
Command Line Interface	- Resource Usage: verteilt - Implementationsaufwand gering - User Experience(UX): gut für Power User - Integration in CI/CD	- UX: wenige Funktion, Anleitung notwendig, kein Gestaltungsspielraum - Installation benötigt - Einarbeitungszeit eher hoch
Integration im Context Mapper	- UX: angenehm für User, wenn er Context Mapper bereits nutzt - kein GUI erforderlich	- Installation benötigt - erfordert Kenntnisse vom Context Mapper, kann User die Context Mapper nicht kennen abschrecken - Einarbeitung in Context Mapper Applikation notwendig - starke Abhängigkeit vom Context Mapper
Webapplikation	- UX: viele Gestaltungsmöglichkeiten - keine Installation notwendig - Einarbeitungszeit gering - mithilfe von Curl einfach eine CLI zu erstellen	- Resource usage: zentralisiert (kann mit mehreren Instanzen und Loadbalancer verteilt werden, sodass kein Bottle-Neck entsteht.) - PaaS Cloud Deployment möglich (Bsp. Heroku) - Implementationsaufwand hoch, GUI Design notwendig, zusätzliche Technologie im Frontend notwendig

Wir haben uns entschieden die Applikation als Webapplikation umzusetzen, weil eine Webapplikation eine kürzere Einarbeitungszeit hat als die beiden anderen Applikationsformen und weil sie keine Installation vom Benutzer benötigt. Dies führt zur Erfüllung der nicht funktionalen Anforderung der Erlernbarkeit B.5.3. Ebenfalls hat es keine direkte Abhängigkeit zum Context Mapper und gibt uns Spielraum in der Gestaltung des User Interfaces. Wir nehmen dafür in Kauf, dass wir einen höheren Implementationsaufwand haben und mehr Zeit für das GUI Design einplanen müssen.

D.3.2 Distribution Pattern

Wir verwenden das Distribution Presentation Pattern. Mit diesem Pattern befindet sich nur die Presentation auf dem Tier 1 Client und der Dialog Control befindet sich auf dem Tier 2, dem Server.[46] Durch diese Aufteilung erhalten wir einen modularen Applikationsaufbau.

D.3.3 Technologiewahl

Durch die Aufgabenstellung ist die Sprache Java im Backend bereits vorgegeben. Wir haben uns dazu entschieden das Framework Spring Boot[3] zu verwenden, weil es das beliebtestes Framework mit Java ist[47] und weil wir in der Studienarbeit bereits erste Erfahrungen damit sammeln konnten. Im Frontend haben wir uns für Thymeleaf entschieden. Uns war wichtig, dass wir ein schlankes Frontend verwenden können, da wenige Funktionen im Frontend zur Verfügung gestellt werden. Ebenfalls haben wir bei Thymeleaf in der Kombination mit Spring Boot schon einige Erfahrungen gesammelt. Um ein Command Line Interface ebenfalls den Nutzern zur Verfügung zu stellen, werden wir voraussichtlich Curl[4] verwenden.

D.3.4 Datenbank

Gemäss den funktionalen und nicht funktionalen Anforderungen ist keine Datenbank notwendig. Aus diesem Grund haben wir uns dazu entschieden, darauf zu verzichten. Durch das Weglassen der Datenbank sinkt die Komplexität der Applikation und bei der Benutzung muss sich der User nicht registrieren. Dies führt zur Erfüllung der Bedien- und Erlernbarkeit B.5.3. Jedoch müssen wir auch auf einige Funktionalitäten verzichten. Der User hat beispielsweise keine Möglichkeit seine Files zu speichern.

3 https://spring.io/projects/spring-boot
4 https://curl.se/

D.4 Logische Architektur

Dieser Abschnitt beschreibt die logische Architektur von der DDD-2-BDD Applikation. Im ersten Abschnitt geben wir eine kurze Übersicht mithilfe eines Schichtendiagramms. Der zweite Abschnitt stellt das Component Diagramm vor und erläutert die wichtigsten Componenten. Im letzten Abschnitt wird detailliert auf die Architektur und das Design eingegangen mithilfe eines Package Diagramms.

D.4.1 Schichtendiagramm

In der Abbilung D.1 ist das Schichtendiagramm zu erkennen. Orange dargestellt ist der Presentation Layer, der die schlanke User Interface Logik enthält. Im Business Layer befindet sich der Kern der Applikation, der zuständig für die Generierung der Testfälle ist. Der Data Access Layer enthält die Klasse CMLReader. Der CMLReader ist dafür zuständig, dass die Daten aus dem CML File zu lesen, dass wir vom Benutzer entgegennehmen. Weil auf den Einsatz von einer Datenbank verzichtet wird, wird mit Files gearbeitet.

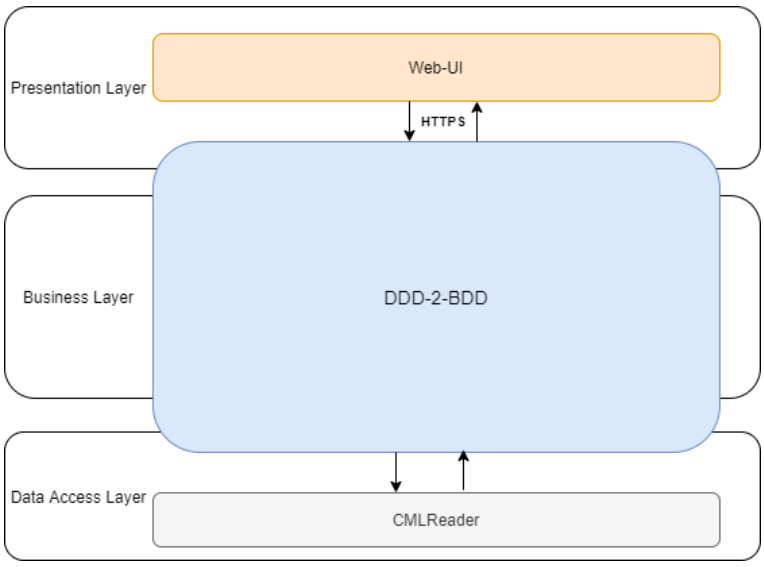

Abbildung D.1: Schichtendiagramm DDD-2-BDD

D.4.2 Component Diagramm

Die Abbilung D.2 zeigt das das Component Diagramm. Die Farben sind jeweils aus dem Schichtendiagramm übernommen. Der TutorialViewController ist grün eingefärbt, um ihn von der Business Logik zu differenzieren. Die TutorialViewController enthält kaum Logik und ist lediglich für die Darstellung der Tutorials zuständig. Die weiteren Komponenten werden im nächsten Abschnitt genauer beschrieben.

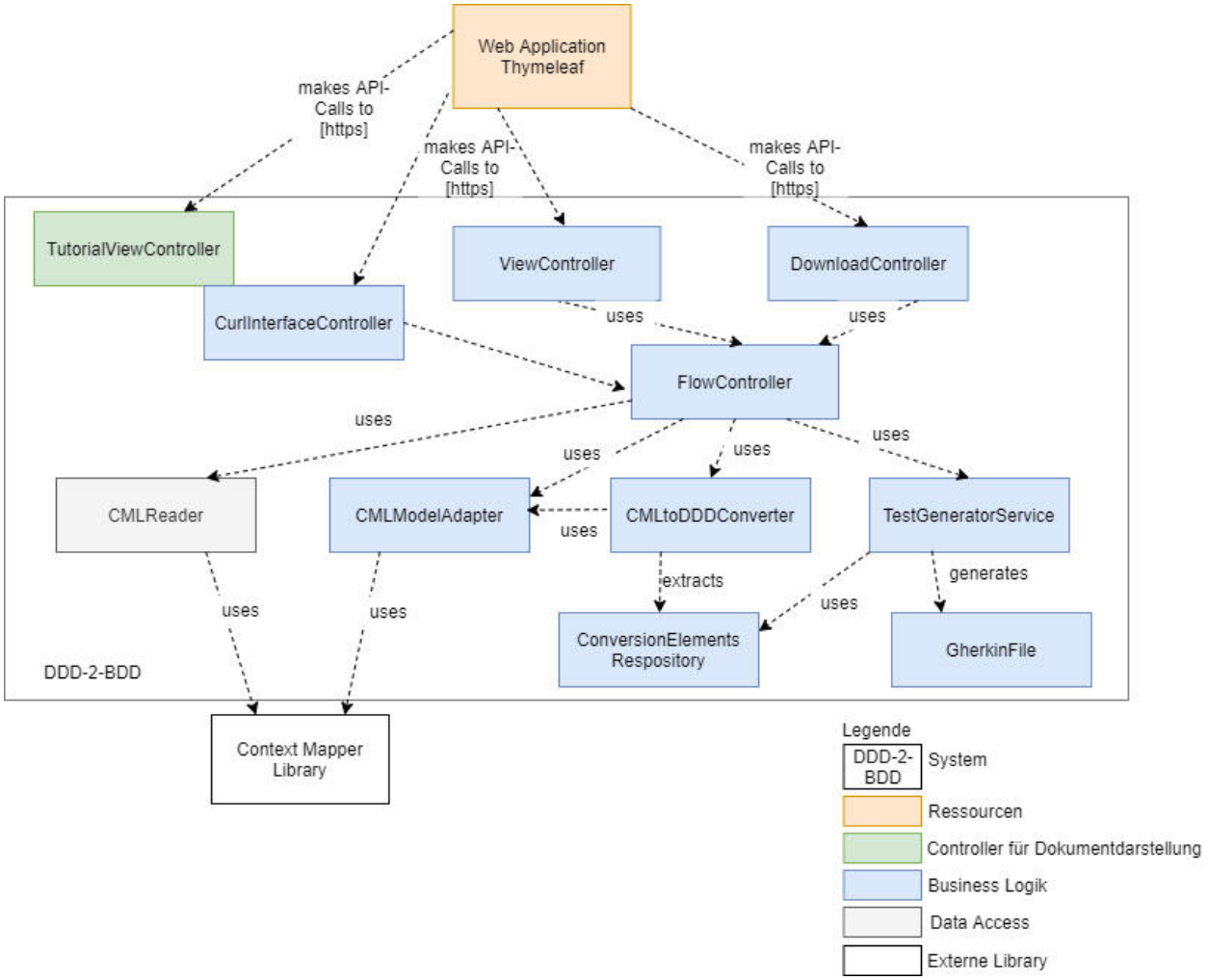

Abbildung D.2: Component Diagramm DDD-2-BDD

In den folgenden zwei Tabellen ist der CMLtoDDDConverter D.2 und der TestGeneratorService D.3 mithilfe von CRC-Cards genauer beschrieben. CRC Cards beschreiben die Komponente, ihre Responsibilites und die Collaborators[48].

Component: CMLtoDDDConverter	
Responsibilities: • aus dem CMLModelAdapter die einzelnen Elemente zu IConversion Elementen konvertieren • anhand des TestingModels entscheiden, welche Elemente konvertiert werden müssen • konvertierte Elemente dem ConversionElementRespository hinzufügen	**Collaborators:** • FlowController greift auf CMLtoDDDConverter zu • CMLtoDDDConverter nutzt CMLModelAdpater • CMLtoDDDConverter nutzt TestingModel • CMLtoDDDConverter nutzt ConversionElementRepository
Candidate implementations technologies:	Spring Service Component

Tabelle D.2: CRC-Card: CMLtoDDDConverter

Component: TestGeneratorService	
Responsibilities: • generiert die Gherkin feature files • fügt die generierten Gherkin feature files dem GherkinFile hinzu	**Collaborators:** • FlowController greift auf TestGeneratorService zu • TestGeneratorService greift auf die einzelnen IConversion Elementen der ConversionElementsRepository zu • TestGeneratorService nutzt das GherkinFile
Candidate implementations technologies:	Spring Service Component

Tabelle D.3: CRC-Card: TestGeneratorService

D.4.3 Package Diagramm

In der Abbilung D.3 ist das Package Diagramm zu sehen, welches das Component Diagramm konkretisiert. Die Farben sind wieder aus den vorherigen Grafiken übernommen worden. In den folgenden Abschnitten erläutern wir die einzelnen Packages genauer.

Abbildung D.3: Package Diagramm DDD-2-BDD

D.4.3.1 Ressources

Im ressources Package sind die statischen Ressourcen, wie Thymleaf und CSS Files abgelegt und auch Bilder.

D.4.3.2 Presentation

Wie bereits erwähnt, wird ein schlanker Presentation Layer verwendet. Der ViewController, DownloadController und CurlInterfaceController greifen über den FlowController auf die Businesslogik zu. Der ViewController ist zuständig für das Darstellen des generierten Gherkinoutputs, als auch für das Input Formular. Das CML File wird in Form von einem mulitpart/form-data (7 BIT Encoding) ins Backend übertragen[49]. Die maximale Filegrösse beträgt 5MB und wurde über die Spring Properties konfiguriert. Im Frontend findet die erste Inputvalidierung statt mit der Prüfung, ob das File mit .cml endet. Der DownloadController ist zuständig für die richtigen Aufrufe um jeweils das Logfile, sowie das Gherkinfile herunterzuladen. Der CurlInterfaceController ist zuständig für das entgegennehmen der Curlaufrufe. Der TutorialViewController stellt sicher, dass die passenden Files dargestellt werden. Im Package model sind die Modelle, die im Presentation Layer verwendet werden enthalten. Unter anderem das TestingModel, welches die Daten enthält, welche Tests vom Benutzer ausgewählt wurden.

D.4.3.3 Business Logik

Die Business Logik beinhaltet vier Packages sowie den FlowController und die Klasse UtilityConstants. In der UtilityConstants haben wir die konstanten Strings gespeichert, die in der Testgenerierung und im Gherkinoutput verwendet werden. Der FlowController ist die Schnittstelle zum Data Access Layer und Presentation Layer. Ebenfalls kontrolliert er den Zugriff auf die weiteren Packages in der Business Logik. Das Sequenzen Diagramm des FlowControllers ist in der Abbildung D.4 dargestellt. Der FlowController konvertiert das Multipartfile aus dem Frontend in ein ByteArrayInputStream und gibt diesen dem CMLReader weiter. Im Package cml_model_adapter ist die Klasse CMLModelAdapter und die dazugehörige Factory Klasse CMLModelFactory. Die Klasse CMLModelFactory verwendet das Packages wrapper für das Erstellen des CMLModelAdapters. Der CMLModelAdapter ist ein Adapter auf die Context Mapper Model, sozusagen eine Aufbereitung des Context Mapper Models, damit es die Konvertierung für uns vereinfacht. Im Package converter_elements sind die Elemente, die für die Generierung gebraucht werden. Die Klasse ConversionsElements enthält lediglich eine Liste von Conversion Elementen. Die Conversion Elemente implementieren, das Interface IConversionElement. Unter anderem bieten die Conversion Elemente die Methode generateTest() an, die ein Feature File zurückgeben. Die Conversion Elemente verwenden bei Bedarf einen zusätzlichen Reader, die im Package Reader enthalten sind. Bei der Testgenerierung wird auf das Package test_generators zugegriffen. Jedes IConversionElement hat eine zugehörige Klasse im Package test_generators. Die Conversion Elemente sind alle Elemente,

die mithilfe des CMLtoDDDConverter aus dem CMLModelAdapter gefiltert wurden und für die Generierung der Gherkin Test gebraucht wird.

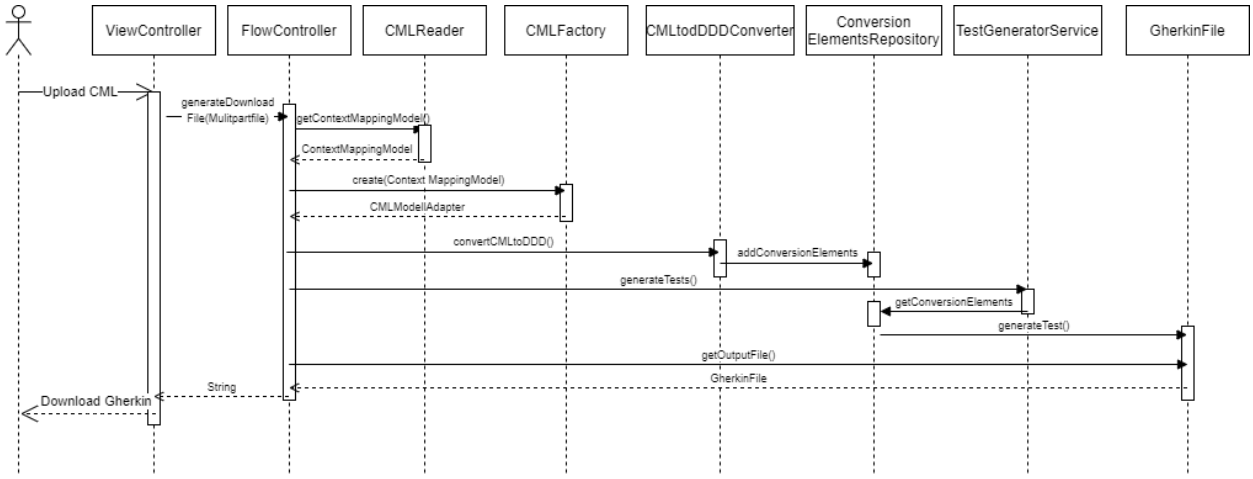

Abbildung D.4: Sequenz Diagramm FlowController

D.4.4 Data Access

Der Data Acces Layer beinhaltet den CMLReader. Der CMLReader liest das Inputfile als ByteArrayInputStream ein und generiert das entsprechende Context Mapper Model. Im CMLReader wird ebenfalls überprüft, ob das CML File korrekt ist, falls nicht wird dem Benutzer ein Fehler zurück gemeldet. Der CMLReader arbeitet mit der Context Mapping DSL. Unvollständige oder fehlerhafte CML Modelle können von der Library nicht verarbeitet werden, aus diesem Grund wird bei einem fehlerhaften Modell eine Meldung an den Nutzer zurückgegeben.

D.4.5 Security

Aufgrund des Entscheids die Applikation als Webapplikation zu gestalten (und diese nach Bedarf auch zu Deployen) hat die Applikation einen erhöhten Bedarf an Sicherheitsmassnahmen als zum Beispiel eine lokale Desktopapplikation. Der folgende Absatz soll nun eine kleine Analyse der Applikation mit dem Fokus auf die Sicherheitsrisiken und Sicherheitsamtsnahmen bieten. Eine vertiefte Sicherheitsanalyse würde jedoch den Scope des Projekts übersteigen, gerade weil das Endprodukt unserer Arbeit ein Prototyp ist und nicht etwa eine kommerzielle Anwendung. Die Sicherheit der Applikation ist natürlich wichtig, hat aber im Zuge unserer Bachelorarbeit nicht missionskritischen Charakter. Zu den Risiken sind sicherlich die folgenden drei Hauptszenarien zu zählen:

- Szenario 1: Ein Angreifer versucht die Daten eines einzelnen Nutzers zu stehlen (CML Input)

- Szenario 2: Ein Angreifer versucht den Service selbst zu komprimieren und so die Kontrolle über die Applikation/die ausführende Maschine zu erlangen, Endziel: Remote Code Execution
- Szenario 3: Ein Angreifer versucht den Service mittels einer Denial of Service Attacke lahmzulegen.

Die File-Upload Funktion ist zurzeit der einzige geplante Userinput welcher es einem Nutzer oder potenziell einem Angreifer erlaubt Daten an den Service zu schicken die über ein simples GET hinweggehen. Um eine böswillige Nutzung dieser Uploadfunktion zu verhindern müssen die folgenden Teile des Files bedacht werden:

- Metadaten des Files: Der Pfad beziehungsweise der Name der Datei könnte genutzt werden um mit Kontrollcharakteren die Datei in eine nicht erwartete Stelle des Filesystems zu kopieren und so eine Konfigurationsdatei zu überschreiben. Dies soll durch auf einen Ordner limitierte Schreibrechte verhindert werden.
- Grösse des Files: Ein Angreifer könnte versuchen mittels arbiträr grossen Dateien (oder auch vielen sehr kleinen) den Service zu Überlasten, sodass legitime Anfragen nicht mehr bearbeitet werden. Zu grosse Dateien können durch eine Limitierung der erlaubten Dateigrösse verhindert werden, das Verschicken von vielen kleinen Dateien per Script könnte mittels Capatchas gelöst werden. Auf diese Massnahme möchten wir aber aus Zeitgründen verzichten.
- Der Inhalt des Files: Ein Angreifer könnte versuchen mittels versteckten Instruktionen in der hochgeladenen Datei die Applikation zu manipulieren. Unsere Applikation liest die Datei mittels CML Library ein. CML hat einen strikten Syntax welcher auf Plaintext basiert und beinhaltet nach bisherigem Stand des Wissens keine Instruktionen, die zu Ausführung von arbiträrem Code führen könnten.

Weiter sind natürlich die folgenden Standardsicherheitsmassnahmen zu treffen:

- Verwendung von HTTPS bei Verwendung ausserhalb von localhost
- Verwendung von HTTP Strict Transport Security (HSTS) bei Verwendung ausserhalb von localhost
- Unterbinden von Cross-Site Requests

Abschliessend ist nochmal darauf hinzuweisen, dass aufgrund der Projektstruktur die Sicherheit der Applikation ausnahmsweise nicht missionskritisch ist und grundsätzlich die Sicherheitsrisiken durch das Starten einer lokalen und von aussen abgeschotteten Instanz ausgeschlossen werden können.

D.5 Externe Abhängigkeiten

Durch die Verwendung von der Context Mapper Library und dem Output, der in Gherkin vorhanden sein soll, ergeben sich zwei externe Abhängigkeiten.

D.5.1 Context Mapper Library

Die Context Mapper Library vereinfacht uns das Einlesen des CML-Files. Die Context Mapper Models weisen eine hohe Verschachtlung auf, insbesondere wenn auf die taktischen DDD Elemente zugegriffen werden muss. Beispielsweise beim Aufruf von einem Attribut, muss zuerst auf den BoundedContext, dann auf das Aggregate und danach auf das DomainObject zugegriffen werden und erst danach kann auf das Attribut zugegriffen werden. Aus diesem Grund haben wir uns dazu entschieden, einen CMLModelAdapter zu implementieren. In der Klasse CMLModelAdapter stellen wir die entsprechenden Methoden zur Verfügung um direkt auf die Elemente zuzugreifen zu können, die wir benötigen. Der CMLModelAdapter erhöht auch die Portabilität der Software, das bei einer zukünftigen Konkurrenzprodukt des Context Mappers nur der Adapter angepasst werden müsste.

D.5.2 Gherkin Syntax

Die Gherkin Syntax hat eine überschaubare Komplexität, sowie auch ein überschaubares Feature Set[50]. Aus diesem Grund und weil wir keine Library gefunden haben, die Gherkin in Java unterstützt, haben wir uns entschieden, diese Komponente selbst zu implementieren. Somit haben wir uns von der Abhängigkeit gelöst. Jedoch müssen unsere Output Files mit Cucumber verwendbar sein, aus diesem Grund muss unser Output den Regeln des Gherkin Syntax entsprechen. Die Validierung des Outputs wird nach der Implementation von einem neuen Feature mit der IDE manuell getestet. Um den Aufwand der Gherkin Implementation möglichst klein zu halten, haben wir das Feature Set möglichst simpel gehalten und nur bei Bedarf ergänzt. Unser Tool unterstützt die folgenden Syntaxelemente: Feature, Story, Scenario, Given, When, Then, sowie Data Tables mit Scenario Outline.

D.6 Deploymentdiagramm

Durch die Verwendung des Springboot Frameworks liegt ein Deployment als ausführbare FAT-JAR nahe. Wir haben uns entschieden, dieses JAR-File in ein Docker Image zu verpacken und als Docker Container auszuliefern. Daraus ergibt sich das folgende Deployment Diagramm:

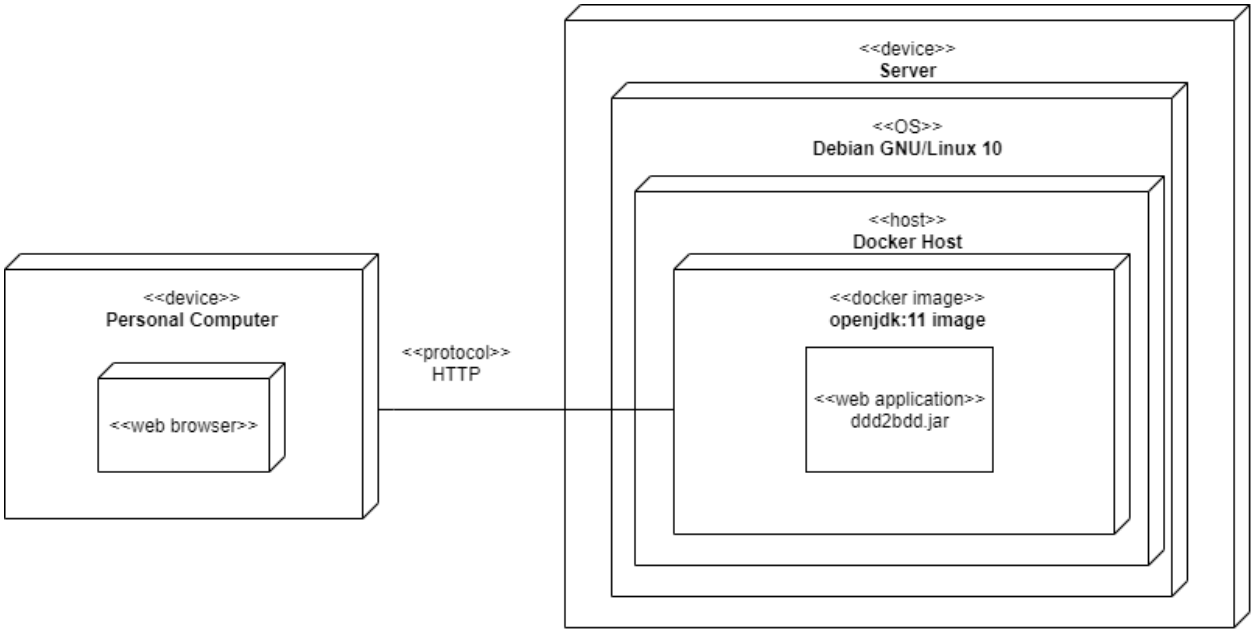

Abbildung D.5: Deployment Diagramm

Ebenfalls bietet sich die Möglichkeit das FAT-JAR File lokal ohne Docker zu deployen oder mit einer Platform as a service (PaaS) Cloud.

D.7 Fazit

Die Entscheidungen bezüglich des CML Inputs und des Gherkin Outputs sind in diesem Dokument festgehalten, sowie die Entscheidung, dass bei den Tutorials mit Cucumber gearbeitet wird. Zusätzlich dazu sind die architektonischen Ziele, Entscheide und Einschränkungen festgelegt, sowie die Logische Architektur. Es wird eine Webapplikation umgesetzt mit Thymeleaf und Spring Boot. Es wird das Distribution Pattern angewendet, genauere Details zur Logischen Architektur sind im Package Diagramm zu finden. Die externen Abhängigkeiten zum Context Mapper und Gherkin werden kurz erläutert und wie mit ihnen umgegangen wird. Ebenfalls wird kurz auf die Security Massnahmen eingegangen. Das Deployment mit Docker wird anhand des Deployment-Diagramms erläutert.

E Qualitätssicherung

E.1 Einführung

E.1.1 Gültigkeitsbereich

Das Dokument beschränkt sich auf die Projektdauer der Bachelorarbeit im FS2021.

E.1.2 Referenzen

Die Referenzen sind als Fussnoten angegeben.

E.1.3 Übersicht

Das Dokument beschreibt die Qualitätsmassnahmen, die Sicherung der Geschichte und die Codestatistiken. Am Ende des Dokuments ist beschrieben, wie Continous Integration(CI) und Continous Delivery(CD) aufgebaut sind.

E.2 Qualitätsmassnahmen

E.2.1 Coding Guidelines

Um die Code Style Guidelines durchzusetzten wird das Tool sonarlint[1] eingesetzt.

1 https://www.sonarlint.org/

E.2.2 Definitons of Done

Die folgende Definition of Done wurde für die Arbeitspakete festgelegt:

- Keine Fehler oder Warnungen
- Pipeline erfolgreich durchlaufen
  - Build ohne Fehler
  - Unit Tests: alle Tests sind erfolgreich, sinnvolle Testcoverage
- Nichtfunktionale Anforderungen erfüllt
- Code Review Prozess durchlaufen
- In *develop* Branch gemerged
- Dazugehöriges Issue ist abgeschlossen
- Falls nötig: Dokumentation angepasst oder neues Arbeitspaket für Dokumentationsänderungen angelegt

E.2.3 Bug Monitoring

Gefundene Bugs werden im GitLab als Issue erfasst und mit dem Tag *incident* markiert.

E.2.4 Stunden-Erfassung auf den Arbeitspaketen

Die Stundenerfassung mit Toggl[2] ist mit GitLab verknüpft und die Zeit wird pro Arbeitspaket im Toggl erfasst. Somit kann im Nachhinein nach vollzogen werden, wieviel Zeit für die einzelnen Arbeitspakete verwendet wurde.

E.2.5 Code-Reviews

Nach dem Vier-Augen-Prinzip wurden Code-Reviews bei jedem Pull-Request durch das andere Teammitglied durchgeführt. Im Code-Review wurde besonders auf die im Studium erlernten Clean Code Techniken geachtet.

2 https://toggl.com/

E.2.6 Dokumentation-Reviews

Die Dokumente werden gegenseitig gegengelesen. Die einzelnen Software Dokumente werden ebenfalls dem Betreuer für die inhaltliche und sprachliche Korrektur zum Review abgegeben. Die Verbesserungsvorschläge werden jeweils in einer neuen Dokumentversion umgesetzt. Für einzelne Dokumente wird ein sprachliches Review von externen Personen durchgeführt.

E.2.7 Unit Tests (Microtesting und Integrations Tests)

Das Java Backend wird mit JUnit[3] getestet. Der Test Fokus ist in der Business Logik. Insbesondere die Klassen, die für die Testszenariengenerierung zuständig sind sollen eine hohe Testabdeckung aufweisen. Um sicherzustellen, dass die bereits entwickelten Funktionalitäten auch nach Änderungen an der Codebasis weiterhin funktionieren, sollen die Tests automatisiert und bei jeder Änderung überprüft werden. Um dies zu realisieren, verwenden wir die CI/CD Funktionalitäten von GitLab [4]

Im Thymeleaf Frontend haben wir aufgrund von Erfahrungen aus der Studienarbeit davon abgesehen ein Testingframework wie zum Beispiel Thymeleaf Testing[5] einzusetzen. Stattdessen haben wir uns entschieden das UI als Teil des Peer Review Prozesses bei jedem Merge Request manuell auszutesten.

E.2.8 Systemtests

Die Systemtest werden manuell durchgeführt auf der Beta Version. Die Testspezifikationen- und Protokolle sind im File Testprotokoll F zu finden.

E.2.9 Perfomance- und Usabilitytest

Die Performance- und Usabilitytest werden ebenfalls manuell durchgeführt auf der Beta Version. Sie dienen zur Überprüfung der Nichtfunktionalen Anforderungen. Die Testspezfikationen- und Protokolle sind im File Testprotokoll F zu finden.

3 https://junit.org/junit5/
4 https://about.gitlab.com/
5 https://github.com/thymeleaf/thymeleaf-testing

E.3 Sicherung der Geschichte

Die Dokumente werden jeweils mit einem Datum und der Version versehen. Bei kleinen Änderungen wird die Version um 0.1 erhöht, bei grossen Änderung um eins. Alle Dokumente sind im GitLab Repository abgelegt und die genauen Änderungen können in den GitLab-Commits eingesehen werden.

Das Code-Repository ist ebenfalls im GitLab. Die Beta Version wird mit einem Tag gekennzeichnet für die Sicherung. Das Code-Repository wird als zip File gespeichert für die Abgabe.

E.4 Codestatistik

E.4.1 Verwendete Programmiersprachen

Die Abbildung E.1 zeigt die Prozentanteil der verwendeten Sprachen. Zu beachten ist jedoch, dass bei den Html Files sehr viele Zeilen vorhanden sind, diese kommen durch die Tutorials zu Stande. Die Tutorials enthalten sehr viele Codebeispiele, was zu einer hohen Zeilenanzahl führt und die Statistik somit etwas manipuliert.

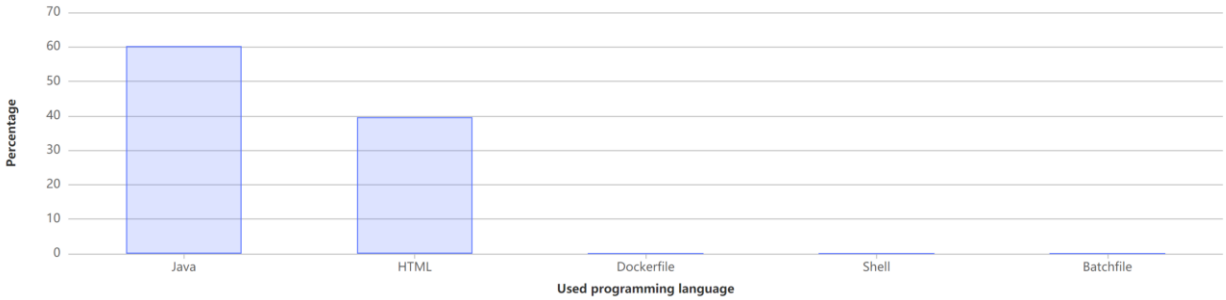

Abbildung E.1: Verwendete Sprachen in Prozent

E.4.2 Code Statistik von SonarQube

Die Abbildung E.2 zeigt die Auswertung, die von SonarQube[6] erstellt wurde. Die Anzahl der Smells ist mit 52 aus unserer Sicht relativ hoch. Jedoch wird es von SonarQuebe mit A bewertet, was in Ordnung bedeutet und ist somit kein gravierendes Problem darstellt. Mit weiteren Refactorings könnte man die Anzahl der Smells noch reduzieren. Ebenfalls ist die Anzahl der Unittests sehr tief mit 64 Tests.(Die Coverage konnte nicht richtig gemessen werden von

6 https://docs.sonarqube.org/latest/

SonarQuebe, die 0.0% ist eine Fehlinformation.) Wir haben uns darauf konzentriert, möglichst die Business Logik gut zu testen. Ebenfalls haben wir uns kein Ziel für die Testabdeckung gesetzt, da unsere Priorität beim Testen bei System- und Usabilitytest lag. Zusätzlich ist noch zu erwähnen, dass es sich bei unserem Tool um einen Prototypen handelt. Die Hauptaufgabe des Prototyps war zu zeigen, dass die Testgenerierung möglich ist.

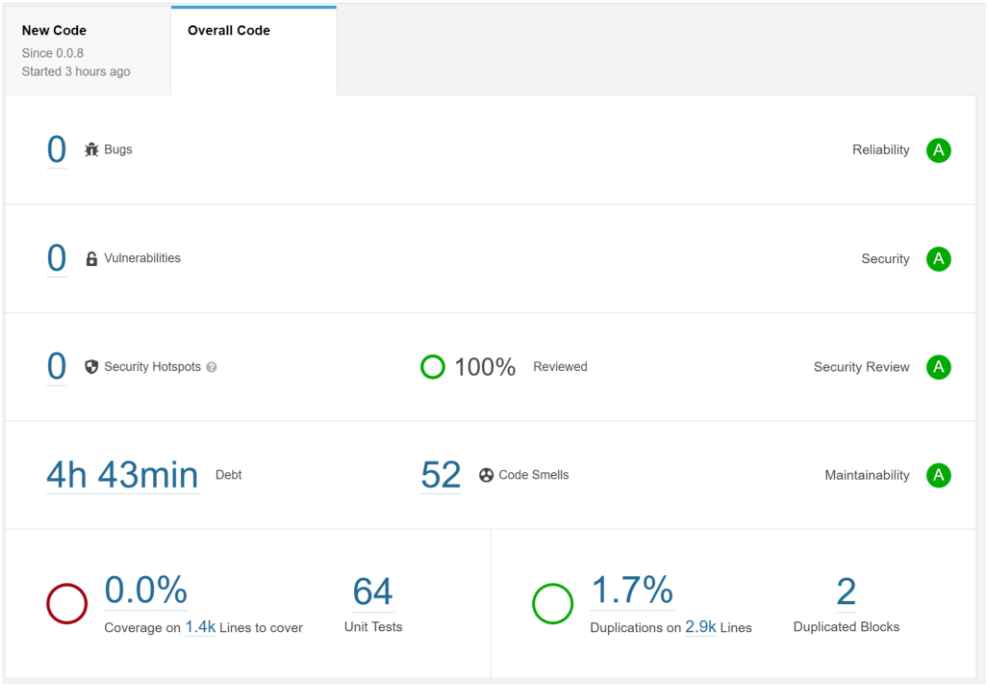

Abbildung E.2: Auswertung SonarQube

E.4.3 Lines of Code

In der Abbildung E.3 sind die Anzahl Zeilen pro Programmiersprache aufgelistet. Die Grafik wurde mit dem Plugin Statistic[7] erstellt. Hier ist wieder dasselbe Probleme mit den Anzahl Html Zeilen, wie bereits bei den verwendeten Sprachen zu erwähnt wurde.

7 https://plugins.jetbrains.com/plugin/4509-statistic

Extension	Count	Size SUM	Size MIN	Size MAX	Size AVG	Lines	Lines MIN	Lines M...	Lines AVG	Lines C...
bat *(BAT files)*	2x	2kB	0kB	2kB	1kB	93	4	89	46	72
css *(CSS files)*	1x	0kB	0kB	0kB	0kB	9	9	9	9	8
gradle *(GRADLE files)*	1x	0kB	0kB	0kB	0kB	1	1	1	1	1
html *(HTML files)*	13x	158kB	0kB	72kB	12kB	3315	8	1635	255	3051
java *(Java classes)*	62x	239kB	0kB	17kB	3kB	5367	7	335	86	4317
md *(MD files)*	1x	2kB	2kB	2kB	2kB	84	84	84	84	48
properties *(Java properties*	2x	0kB	0kB	0kB	0kB	13	5	8	6	12
sh *(SH files)*	1x	0kB	0kB	0kB	0kB	11	11	11	11	6
Total:	**83x**	**403kl**	**3kB**	**95kB**	**20kB**	**8893**	**129**	**2172**	**498**	

Abbildung E.3: Statistic Auswertung Lines of Code

E.5 CI/CD

Um garantieren zu können, dass jede Änderung an der Codebasis getestet wird und compiliert muss für jeden Git Commit ein entsprechender Test durchgeführt werden. Dieser Prozess wird mittels CI/CD automatisiert. Die Verwendung der CI/CD Mechanismen ist analog zu ihrer Verwendung in der dieser Bachelorarbeit vorangegangenen Studienarbeit miro2cml[8]. Aufgrund der analogen Verwendung wie in miro2cml, werden die CI/CD Mechanismen analog zum miro2cml Projekt beschrieben. CI steht für Continuous Integration, also das kontinuierliche integrieren der Änderungen in die Codebasis. Dadurch das für jeden Commit diese Integration gemacht wird, ist das jeweilige Delta klein, was die Übersichtlichkeit verbessert und die Integrationskomplexität verringert. CD steht Continuous Delivery oder Continuous Deployment, in unserem Fall für Continuous Deployment. Dies bedeutet, dass die integrierten Codeänderungen zusätzlich noch automatisch deployt werden. Dadurch werden Änderungen schneller veröffentlicht, was bedeutet, dass sowohl Bugs schneller entdeckt werden und die Komplexität diesen zu finden und zu beheben geringer ist. Weiter kann der Nutzer auch schneller von Verbesserungen und neuen Funktionen profitieren. Wenn ein Commit welcher auf einem Featurebranch entwickelt wurde erfolgreich compiliert und alle automatisierten Tests erfolgreich waren, können die Änderungen in einen der Hauptbranches gemergt werden. Auf den Hauptbranches ist im Gegensatz zu den Featurebranches Continious Deployment aktiviert. Dieser Prozess wird oft auch CI/CD Pipeline genannt, weil der Sourcecode eine Pipline von Instruktionen durchläuft. Um zu beschreiben wie unsere CI/CD Pipeline funktioniert sind die folgenden Komponenten und Konzepte relevant.

E.5.0.1 GitLab

Gitlab[9] ist ein Sourcecode Versionierung und Verwaltung-Tool. Es ist im Grunde ein Git-Server mit vielen zusätzlichen Funktionalitäten. Wir haben uns dazu entschieden, die GitLab

8 https://eprints.ost.ch/id/eprint/935/
9 https://about.gitlab.com/

Instanz der OST zu verwenden. [10]

E.5.0.2 Docker

Docker[11] bietet die Möglichkeit Images zu definieren, diese Images dienen als Applikations-"Schablonen". Aus diesen Docker Images können Docker Container gebuilded werden (Ein Container ist also eine Instanzierung eines Images), welche dann auf jedem Gerät lauffähig ist, auf welchem Docker installiert ist. Oder anderst ausgedrückt, Docker isoliert eine Applikation von ihrer Umgebung und löst so das "it works on my machine" Probblem.

E.5.0.3 Gitlab CI/CD

Die GitLab CI/CD ist eines der zusätzlichen Features, die GitLab von einem generischen Git-Server abhebt. GitLab CI/CD bietet die Möglichkeit, Instruktionsfolgen zu definieren, die sowohl manuell, als auch automatisch ausgeführt werden können. Bei der automatischen Auslösung sind sowohl zeitliche, als auch trigger-on-commit Verfahren möglich. Die Instruktionsfolgen sind in Stages unterteilt. Jede Stage definiert eine Instruktionsfolge und eine Umgebung, in der die Instruktionen ausgeführt werden sollen. Zur Definition der Umgebung können Docker Images verwendet werden. Die eigentliche Ausführung der Instruktionen wird auf den GitLab Runnern durchgeführt. Resultate solcher Stages können als Artefakte deklariert werden. Solche Artifakte können sowohl über das GitLab Web-UI heruntergeladen als auch von anderen Stages wiederverwendet werden. Weiter können einzelne Stages so definiert werden, dass diese nur für Commits auf bestimmten Refs, also Branches/Tags möglich sind.

E.5.0.4 Gitlab Runner

Die Runner können sowohl auf der gleichen Host-Maschine, wie die GitLab Instanz als auch auf einer komplett anderen Host-Maschine laufen. Die CI/CD Instruktionen rufen einen Runner für die eigentliche code-execution auf. Dabei werden die Instruktionen und die Umgebungsdefinition einer Stage an einen Runner übergeben, dieser lädt zuerst die Umgebung und führt dann die Instruktionen aus.

E.5.0.5 GitLab Container Registry

Die GitLab Container Registry, sollte eigentlich eher GitLab Image Registry heissen, den in dieser Regestry können Docker Images hinterlegt, also registriert werden. Ist ein Image erst einmal in der Registry hinterlegt, kann das Image über die Standard Docker Befehle gepullt

10 https://gitlab.ost.ch//
11 https://www.docker.com/

werden. Um den Datenschutz zu gewährleisten ist dafür zwar eine Authentifizierung gegenüber der Registry nötig, diese ist durch die in die CI/CD eingebauten Authorisierungs-Token jedoch einfach umzusetzen. Aber auch von außerhalb der CI/CD Umgebung kann auf diese Registry, wenn auch mit etwas mehr Konfigurationsaufwand zugegriffen werden.

E.5.1 Unsere CI/CD Pipeline

Unsere Pipeline basiert auf vier Stages: test, build, prepare-deploy und deploy. Diese vier Stages werden nun kurz erläutert.

E.5.1.1 Test

Die Test Stage, kümmert sich darum die Integrität der Codebasis zu überprüfen, dabei werden die vordefinierten Unittests ausgeführt. Das ganze geschieht über das Laden einer Gradle Umgebung und das Ausführen des Gradle Tasks **test**. Die dabei entstehende Test Zusammenfassung haben wir als Artifakt deklariert. GitLab erkennt die auf dem jUnitTest-Framework basierenden Testresultate und parst die Zusammenfassung, sodass diese bequem im Web-UI betrachtet werden kann.

E.5.1.2 Build

Die Build Stage, kümmert sich, wie der Name schon sagt, um das Builden der Applikation. Dabei wird wieder zuerst mithilfe von Docker eine Gradle Umgebung geladen. Sobald die Umgebung geladen ist, wird der Gradle Tasks **bootJar** ausgeführt. Dadurch builded Gradle eine FAT-JAR. Diese FAT-JAR wird als Artifakt gespeichert.

E.5.1.3 Prepare Deploy

Die Prepare Deploy Stage, bereitet das Ausliefern der Applikation vor. Dafür wird als erstes, die in der Build Stage gebuildete, FAT-JAR bezogen und eine Verbindung zur GitLab Container Registry aufgebaut. Als nächstes wird das Docker Image gebuilded und in der Registry hinterlegt. Diese Stage wird nur für die drei Hauptbranches **develop, test und master** ausgeführt. Zusätzlich haben wir uns einen Tag **test-deploy** eingeführt. Ist ein Commit mit dem test-deploy Tag getaggt, ist dieser Commit ebenso für das automatische Deployment qualifiziert. Dies ermöglichte es uns, während der Entwicklung mit minimalen Zusatzaufwand unseren Code auch ausserhalb der localhost Umgebung manuell testen zu können.

E.5.1.4 Deploy

Und zuletzt wird in der Deploy Stage die Applikation in Form eines Docker Images aus der GitLab Container Registry bezogen. Dabei wird diese Stage nur für vier Git-Refs, für die auch schon **prepare-deploy** ausgeführt wurde, durchgeführt.

E.6 Fazit

Die getroffenen Qualitätsmassnahmen zeigten sich als wirkungsvoll, zumindest ergab dies die positiv ausgefallene Auswertung mit dem Tool SonarQube die am Ende des Projekts stattfand. Die Anzahl Code Zeilen bietet uns einen persönlichen Vergleich zu anderen Projekten. Die Auswertung mit SonarQube gibt uns ebenfalls Hinweise, wie gut die Implementation ist. Dank der CI/CD konnten wir während dem Projekt viel Zeit sparen, da wir die Unittests nicht manuell ausführen mussten. Zusätzlich konnte dank der CI/CD der Prototyp , stets in seiner aktuellsten Version, sehr unkompliziert per Weblink dem Betreuer zum Testen zur Verfügung gestellt werden.

F Testprotokoll

F.1 Einführung

F.1.1 Version

Version 1.1, 12. Juni 2021

F.1.2 Zweck

Spezifiziert die System-, Performance- und Usabilitytests von DDD-2-BDD.

F.1.3 Gültigkeitsbereich

Das Dokument beschränkt sich auf die Dauer der Bachelorarbeit im FS2021.

F.2 Grundlage für die Erstellung der Tests

Für die Grundlage der Erstellung der Tests dient das Dokument Anforderungsspezifikationen.

F.3 Systemtest

Die Systemtests sind manuell durchgeführt worden auf dem *develop* Branch (lokal).

Nr	Beschreibung	erwartetes Verhalten	Pers	Dat.	Result
T01	User möchte anhand seines CML Files Tests generieren	File kann hochgeladen werden und Tests werden generiert	Saskia	12.06	erfolgreich
T02	User kann auswählen, welche Tests er generieren möchte	Dem Inputformular entsprechend findet die Testgenerierung für die gewählten Eigenschaften statt	Saskia	12.06	erfolgreich
T03	User kann bei Fehler nachschauen, wo das Problem liegt	Dem Benutzer steht ein Logfile zu Verfügung	Saskia	12.06	erfolgreich
T04	User kann Tests generieren für das Maximum, welches im CML angegeben ist	Es werden Tests generiert, die diesen Fall abdecken	Saskia	12.06	erfolgreich, aber nur für Integers umgesetzt
T05	User kann Tests generieren für das Minimum, welches im CML angegeben ist	Es werden Tests generiert, die diesen Fall abdecken	Saskia	12.06	erfolgreich, aber nur für Integers umgesetzt
T06	User kann Tests generieren für den Range, welcher im CML angegeben ist	Es werden Tests generiert, die diesen Fall abdecken	Saskia	12.06	erfolgreich, aber nur für Integers umgesetzt
T07	User kann die Länge von einem Attribut testen	Es werden Tests generiert, die diesen Fall abdecken	Saskia	12.06	nicht umgesetzt
T08	User kann das Pattern von einem Attribut überprüfen	Es werden Tests generiert, die diesen Fall abdecken	Saskia	12.06	nicht umgesetzt
T09	User kann Tests generieren, die die Attribute auf immutable testen	Es werden Tests generiert, die diesen Fall abdecken mit den Konstruktoren und Settermethoden	Saskia	12.06	erfolgreich, Tests müssen noch ergänzt werden mit Daten

Nr	Beschreibung	erwartetes Verhalten	Pers	Dat.	Result
T10	User kann Tests generieren, die die Attribute auf changeable testen	Es werden Tests generiert, die diesen Fall abdecken mit den Konstruktoren und Settermethoden	Saskia	12.06	erfolgreich, Tests müssen noch ergänzt werden mit Daten
T11	User kann Tests generieren, die die Attribute auf required testen	Es werden Tests generiert, die diesen Fall abdecken mit den Konstruktoren	Saskia	12.06	erfolgreich, Tests müssen noch ergänzt werden mit Daten
T12	User kann Tests generieren, die die Attribute auf nullable testen	Es werden Tests generiert, die diesen Fall abdecken mit den Settermethoden	Saskia	12.06	erfolgreich
T13	User kann Tests generieren, die Collections auf ihre maximale Grösse überprüfen	Es werden Tests generiert, die diesen Fall abdecken	Saskia	12.06	erfolgreich
T14	User kann Tests generieren, die Collections auf ihre minimale Grösse überprüfen	Es werden Tests generiert, die diesen Fall abdecken	Saskia	12.06	nicht umgesetzt
T15	User kann Tests generieren, die Collections auf ihre minimale und maximale Grösse überprüfen	Es werden Tests generiert, die die Size überprüfen der Collections	Saskia	12.06	nicht umgesetzt
T16	User möchte Tests generieren, die die 1:1 Beziehungen zwischen Entity und ValueObject überprüfen	Es werden Tests generiert, die das Löschen und Verändern der Referenz überprüfen	Saskia	12.06	erfolgreich
T17	User möchte Tests generieren, die die 1:n Beziehungen überprüfen	Es werden Tests generiert, die das Löschen und Verändern der Referenz überprüfen	Saskia	12.06	nicht umgesetzt, nur mit Aggregation un Komposition

Nr	Beschreibung	erwartetes Verhalten	Pers	Dat.	Result
T18	User möchte Tests generieren, die die 1:n Aggregation Beziehungen überprüfen	Es werden Tests generiert, die das Löschen und Verändern der Referenz überprüfen	Saskia	12.06	erfolgreich
T19	User möchte Tests generieren, die die 1:n Komposition Beziehungen überprüfen	Es werden Tests generiert, die das Löschen und Verändern der Referenz überprüfen	Saskia	12.06	erfolgreich
T20	User möchte Tests generieren, die die n:m Beziehungen überprüfen	Es werden Tests generiert, die das Löschen und Verändern der Referenz überprüfen	Saskia	12.06	nicht umgesetzt
T21	User möchte Tests generieren, die die bidirektionalen n:m Beziehungen überprüfen	Es werden Tests generiert, die das Löschen und Verändern der Referenz überprüfen	Saskia	12.06	nicht umgesetzt
T22	User möchte Tests generieren, die Referenzen auf Idempotenz überprüfen	Es werden Tests generiert, die die Referenz testen	Saskia	12.06	nicht umgesetzt
T23	User möchte Tests generieren, die Vererbung überprüfen	Es werden Tests generiert, die die Vererbung testen	Saskia	12.06	nicht umgesetzt
T24	User möchte Tests generieren, die kontrollieren, ob die ValueObjects kein Identität haben	Es werden passende Testfälle generiert	Saskia	12.06	nicht umgesetzt
T25	User möchte Tests generieren, die kontrollieren, ob die Entities eine Identität haben	Es werden passende Testfälle generiert	Saskia	12.06	erfolgreich

Nr	Beschreibung	erwartetes Verhalten	Pers	Dat.	Result
T26	User möchte Tests generieren, die kontrollieren, ob die States im AggregateLifecycle eingehalten werden	Es werden passende Testfälle generiert	Saskia	12.06	erfolgreich, keine optimale Testabdeckung bei mehreren Output States
T27	User möchte Tests generieren, die kontrollieren, ob die Services stateless sind	Es werden passende Testfälle generiert	Saskia	12.06	nicht umgesetzt

F.4 Usability- und Performancetests

Usabilitytests wurden in erster Linie mit dem Betreuer Olaf Zimmermann durchgeführt. Einmal pro Woche wurde der aktuelle Projektstand als Demo vorgeführt und zusätzlich als Link zum Testen abgegeben. Das Feedback wurde jeweils als Arbeitspaket aufgenommen und umgesetzt. Die Tabelle zeigt auf, welche nicht funktionalen Anforderungen erfüllt wurden. Die Performance- und Usabilitytests wurden manuell durchgeführt auf dem *develop* Branch (lokal). Für die Zeitmessung wurden im Browser die Entwicklungswerkzeuge verwendet.

Nr	Beschreibung	erwartetes Verhalten	Pers	Dat.	Result
NFR01	Erlernbarkeit, Bedienbarkeit	Der User kann sich innerhalb von 10 Minuten einen Überblick von der Applikation verschaffen und die ersten Testszenarien generieren lassen.	Saskia	12.06	erfolgreich
NFR02	Funktionale Vollständigkeit	Für alle Use Cases werden die Sunny Day, Rainy Day und Stormy Day Szenarien generiert.	Saskia	12.06	nicht für alle Use Cases erfüllt, siehe Testgnerationdokument für Testabdeckung

Nr	Beschreibung	erwartetes Verhalten	Pers	Dat.	Result
NFR03	Performance	Die Testgenerierung dauert nicht länger als drei Minuten für ein CML File mit 700 Zeilen.	Saskia	12.06	getestet mit Lakeside-Mutual Beispiel, Responszeit <10s
NFR04	Portabilität	Der Initialisierungsaufwand für die gängigen Betriebssysteme unterscheidet sich um nicht mehr als 2 Minuten	Saskia	12.06	erfolgreich, weil Browseranwendung bei allen System gleich lange haben(getestet mit Windows und Ubuntu)
NFR05	Nutzerfehlerschutz	Dem Nutzer wird innerhalb von 20 Sekunden eine Meldung gegeben, wenn das File nicht korrekt ist.	Saskia	12.06	erfolgreich
NFR06	Bedienbarkeit	Ein neuer Benutzer soll sich innerhalb von 2 Minuten zurechtfinden können.	Saskia	12.06	erfolgreich

NFR01 und NFR06 wurde innerhalb des Usabilitytests des Tutorials getestet.

F.4.1 Usabilitytests für die Tutorials

Um die Tutorials zu überprüfen, wurde ein Usabilitytest durchgeführt mit einer Mitstudentin, die den Context Mapper bereits genutzt hat, aber noch nie mit BDD in Kontakt kam. Es wurde das Beginner Tutorial durchgearbeitet. Für das Beginner Tutorial wurden 90 Minuten gebraucht. Der Informationsgehalt zeigte, sich jedoch als gut und ebenfalls als verständlich. Positiv fand die Testperson auch, dass es viele Links auf detailliertere Beschreibungen gibt. In der folgenden Liste sind noch einige spezifische Erkenntnisse aus dem Usabilitytest dokumentiert:

- Select all, deselect all Button wurde im Frontend vermisst
- Nicht ganz klar, was getestet wird bei den Konstruktoren und Settermethoden
- Probleme beim Tutorialdokument als PDF mit den Codesnippets, die über eine Seite hinausgehen

Nach dem Usabilitytest wurden die gewünschten Buttons im Frontend eingefügt. Um die Tests zusätzlich für den Benutzer zu beschreiben, wurde ein zusätzliches Dokument Testgeneration erstellt. Die Tutorials sind in der Webapplikation verfügbar, dass es keine Probleme gibt, bei Codesnippets, die im PDF geteilt wurden.

In den Anforderungsspezifikationen wurden Fragen definiert, die innerhalb der Applikationsdokumentation beantwortet werden sollen. In der folgenden Tabelle ist aufgeführt, in welchen Dokumenten, die Fragen beantwortet werden.

Frage	**Antwort**
Zu welchem Zeitpunkt, kann ich das DDD-2-BDD Tool im Projekt integrieren?	nicht explizit erläutert, gegeben durch den Einstieg im Beginner Tutorial
Wie werden meine Testszenarien generiert?	im Dokument Testgeneration erklärt
Sind die Testszenarien vollständig? Wie hoch ist die Testabdeckung?	im Dokument Testgeneration erklärt
Wie kann ich die Testszenarien in meinem Projekt integrieren und automatisieren?	spezifische Erklärungen für die generierten Testfälle im Advanced Tutorial
Welcher Vorteil bringt mir die Verwendung von DDD-2-BDD?	nicht explizit erklärt, sollte sich jedoch durch die Testfälle ergeben, die man nicht von Hand schreiben muss
Wie kann man das Tool ohne DDD verwenden?	diese Frage wird in den Dokumenten nicht beantwortet, das Advanced Tutorial kann jedoch Cucumber Nutzern helfen Step Definitions zu definieren, die nicht von DDD-2-BDD generiert wurden

F.5 Fazit

Insegesamt waren 16 von 27 Systemtests erfolgreich. Die Tests, die nicht erfolgreich waren, wurden nicht umgesezt. Es wurden 13 von 24 Use Cases umgesetzt. Die nicht funktionalen Anfoderungen wurden alle erfüllt, bis auf die Unschönheit bei der Testabdeckung. Bei den nicht funktionalen Anforderungen zeigte sich jedoch auch, dass das Tool innerhalb von wenigen Minuten verstanden wird und Tests generiert werden können, jedoch dauert die Verarbeitung der Gherkin Files mit dem Beginner Tutorial bis zu 90 Minuten. Vier von den sechs Fragen wurden innerhalb der Dokumentation beantwortet.

G User Guide

G.1 Beginner Tutorial: DDD-2-BDD

Table of Contents

G.1.1 Introduction

In this tutorial we will give you an introduction how you generate Gherkin feature files with your Context Mapper file (CML) and the tool DDD-2-BDD. We also explain you how you implement the Gherkin feature files into your Gradle or Maven project and how you make the Gherkintests runnable with the framework Cucumber[1].

First setup your project. After that generate the Gherkin feature files with the tool DDD-2-BDD. DDD-2-BDD generates Gherkin feature files. The Context Mapper[2] DSL allows you to model your Domain Driven Design (DDD) domain. In this tutorial an example domain is used, that represents a part of a student system. The corresponding Context Mapper model

1 https://cucumber.io/
2 https://contextmapper.org/

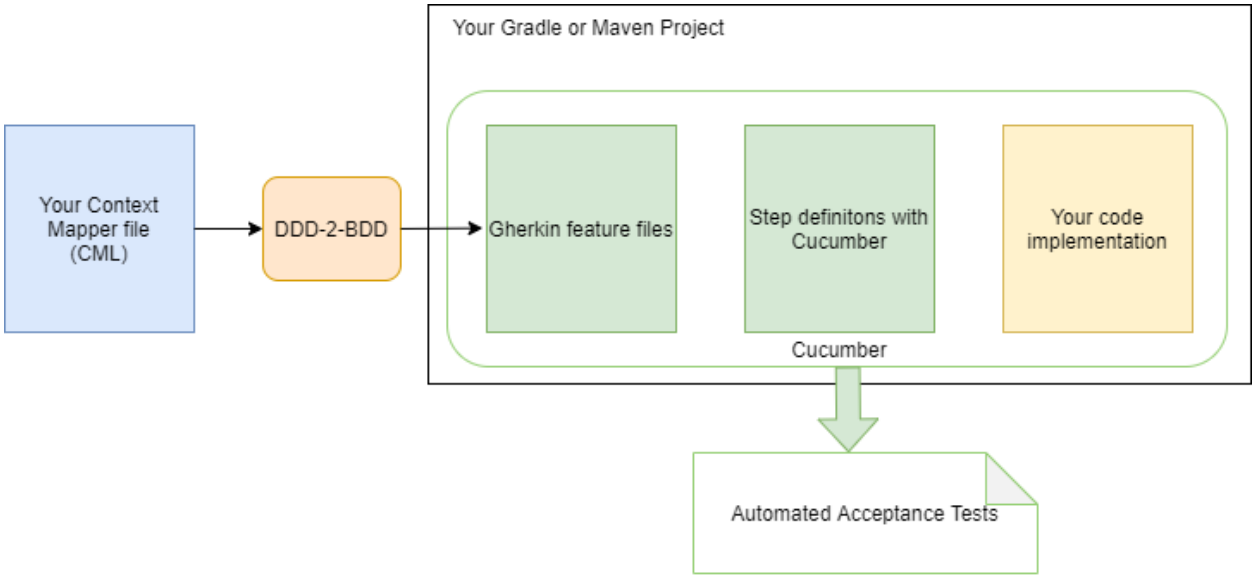

Figure G.1: Overview Tutorial

is shown in the figure below. The model has an Entity *Student* and a ValueObject *StudentId*. The object *Student* has an id, a name, a surname, an address, a subject, a martikelNumber and a semester.

```
BoundedContext ExampleDomain{
    Aggregate Student{
        Entity Student{
            -@StudentId id !changeable
            String name required
            String surname required
            String address
            String subject
            BigInteger martikelNumber !changeable
            int semester range="1,14"
        }
        ValueObject StudentId{
            String id key
        }
    }
}
```

The Entity attributes that have no further keywords have a getter and a setter method. The Entity attributes marked as `required`, have also a getter and a setter method. They are also contained in the constructor. The attributes in an Entity which are marked as `!changeable` are immutable and implemented as final. They are also in the constructor

and have a getter method. The attributes in a ValueObjects are by default `!changeable`. The syntax `-Object or -@Object` implements references, like the *StudentId* in the Entity *Student*. We have the id and the martikelNumber defined as not changeable, because they should not change. The name and the surname are required to register a student and are therefore marked as `required`. A student could maximum study 14 semester and can be registered in the system when he is in his first semester, so we define a range between one and 14 with the key word `range="1,14"`. For further informations about the Strategic syntax visit the Context Mapper website[3] and about the Tactic syntax the Sculptor documentation[4], which is implemented in the Context Mapper.

The DDD-2-BDD tool generates four tests. Two test to check if the constructors and setter methods of the *Student* and *StudentId* has been implemented correctly. One test to confirm if the reference between the *Student* and *StudentId* has been implemented right. And one test to look if the range check has been implemented correctly. The tests are generated as Gherkin feature files. Gherkin feature files are structured executable specification. Further informations about the Gherkin feature files[5]. Mostly they are structured like this:

```
Scenario: TestingModel the attribute semester range minimum with boundary data in
   Given an attribute semester with type int and value 7
   When semester is set to 1
   Then the attribute semester is equals 1
```

The Gherkin features files can be implemented for example in a Maven or Gradle project as *.feature* files. With the framework Cucumber you can define for each step in the Gherkin feature file a corresponding method, the step defintion[6]. The step definitions combine your code with the Gherkin feature files and make the Gherkin tests runnable.

G.1.2 Requirements

- Java 13
- Gradle or Maven(version 3.3.1 or higher)
- Junit
- IDE: Intellij or Eclipse
- Cucumber Plugin installed in your IDE: [7]Plugin for Intellij (including Gherkin), [8]Plugin for Eclipse

3 https://contextmapper.org/
4 http://sculptorgenerator.org/documentation/advanced-tutorial
5 https://cucumber.io/docs/gherkin/reference/
6 https://cucumber.io/docs/cucumber/step-definitions/
7 https://plugins.jetbrains.com/plugin/7212-cucumber-for-java
8 https://cucumber.github.io/cucumber-eclipse/

G.1.3 Step 1: Setup the example project

Hint: This section is copied from the Cucumber Documentation[9].

G.1.3.1 Create an empty Cucumber project

Decide whether you'd prefer to use Gradle or Maven.

With Maven

For Maven, we'll start by creating a new project directory with the `cucumber-archetype` Maven plugin. Open a terminal, go to the directory where you want to create your project, and run the following command:

```
mvn archetype:generate                          \
   "-DarchetypeGroupId=io.cucumber"              \
   "-DarchetypeArtifactId=cucumber-archetype" \
   "-DarchetypeVersion=6.10.4"                  \
   "-DgroupId=exampledomain"                     \
   "-DartifactId=exampledomain"                  \
   "-Dpackage=exampledomain"                     \
   "-Dversion=1.0.0-SNAPSHOT"                    \
   "-DinteractiveMode=false"
```

You should get something like the following result:

```
[INFO] Project created from Archetype in dir: <directory where you created
the project>/cucumber
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Change into the directory that was just created by running the following command:

```
cd exampledomain
```

Open the project in IntelliJ IDEA:

- **File -> Open... -> (Select the pom.xml)**
- Select **Open as Project**

9 https://cucumber.io/docs/guides/10-minute-tutorial/

With Gradle

One way to create this sample Cucumber project using Gradle is to convert the above generated Maven archetype into a Gradle project.

Run the following command from the `exampledomain` directory:

```
gradle init
```

Add following dependency configuration to your build.gradle file:

```
configurations {
    cucumberRuntime {
        extendsFrom testImplementation
    }
}
```

Add the following Task to your `build.gradle` file:

```
task cucumber() {
    dependsOn assemble, testClasses
    doLast {
        javaexec {
            main = "io.cucumber.core.cli.Main"
            classpath = configurations.cucumberRuntime + sourceSets.main.output
             + sourceSets.test.output
            args = ['--plugin', 'pretty', '--glue', 'exampledomain',
              'src/test/resources']
        }
    }
}
```

Note that you also need to add the necessary dependencies/configurations to `build.gradle` depending on which version of Gradle you are using. See the [10]Build Tools section. If you follow this guide be sure to set your `--glue` path to `exampledomain` for this tutorial.

If you have not already, open the project in IntelliJ IDEA:

- **File -> Open... -> (Select build.gradle)**
- Select **Open as Project**

You now have a small project with Cucumber installed.

10 https://cucumber.io/docs/tools/java/#gradle

G.1.3.2 Verify Cucumber installation

To make sure everything works together correctly, let's run Cucumber.

Maven:

```
mvn test
```

Gradle:

```
gradle cucumber
```

You should see something like the following:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running exampledomain.RunCucumberTest
No features found at [classpath:exampledomain]

0 Scenarios
0 Steps
0m0.004s

Tests run: 0, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.541 sec

Results :

Tests run: 0, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Cucumber's output is telling us that it didn't find anything to run.[51]

G.1.4 Step 2: Create the Gherkin feature files

After you have created the project you can start by generating the BDD Output. Start with copying the CML model (shown below and in the introduction) to your device and save it with the ending *.cml* (For Example: exampleDomian.cml). Then upload the cml file to the DDD-2-BDD web application, choose the test *Testing the constructor and setter methods for Entities and ValueObjects* and generate the first Gherkin feature files.

```
BoundedContext ExampleDomain{
    Aggregate Student{
        Entity Student{
            -@StudentId id
            String name required
            String surname required
            String address
            String subject
            BigInteger martikelNumber !changeable
            int semester range="1,14"
        }
        ValueObject StudentId{
            String id key
        }
    }
}
```

The DDD-2-BDD tool creates two different feature files: *Check access for the object Student* and *Check access for the object StudentId*.

G.1.5 Step 3: Import Gherkin feature file

After you have generated the Gherkin feature files you want to implement the files in to your project.

Add a new File to the folder *src/test/resource/exampledomain* with the name *student_acces.feature*. Copy and paste the Feature file from the DDD-2-BDD application *Check access for the object Student* in the *student_access.feature* file.

You can now run the test. When you use Maven you run all tests with the command: `mvn test`. When you use Gradle run the tests with the command: `gradle cucumber`.

Use the command `mvn test Dcucumber.options="src/test/resources/exampledomain /student_access.feature"` to run only the *student_access* file in Maven. (In Intellij it is

also possible with a right click on the file and choose *Run 'Feature: student_access'*. In Eclipse you can go to the project source, right click and choose *Run As Maven test*.)

If warnings appear that the steps are undefined, you have done everything right.

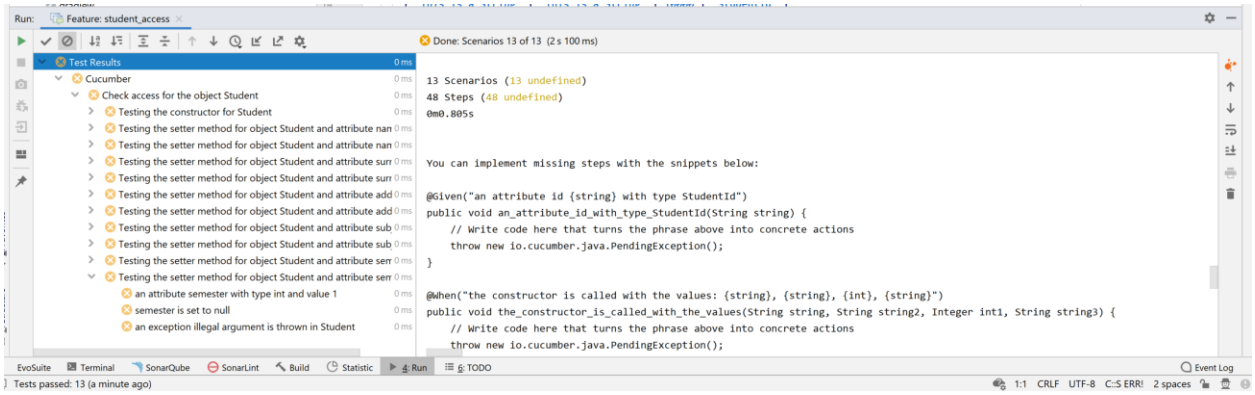

Figure G.2: Undefined steps

Before going to the next step, replace all TODO's in the Gherkin feature file *student_access.feature* with corresponding values.

G.1.6 Step 4: Create the pending steps

Start by renaming the *StepDefinitons.java* file to *StudentSteps.java* in the directory *src/test/java/exampledomain*. We will create for each domain object a step definition file. Visit the Cucumber Website[11] for further informations about grouping the step definitions.

When you run the tests (`mvn test` or `gradle cucumber`), Cucumber suggest you some code snippets to implement the steps definitions. An example snippet is shown in the code below. Scroll down to the following line in the output: `You can implement missing steps with the snippets below` and copy the snippets below to the class *StudentSteps.java*.

11 https://cucumber.io/docs/gherkin/step-organization/

```
You can implement missing steps with the snippets below:

@Given("an attribute {string} with type String")
public void an_attribute_with_type_String(String string) {
// Write code here that turns the phrase above into concrete actions
throw new io.cucumber.java.PendingException();
}
```

When you run the tests (`mvn test` or `gradle cucumber`), the steps are now marked as pending or skipped (when they follow an pending step).

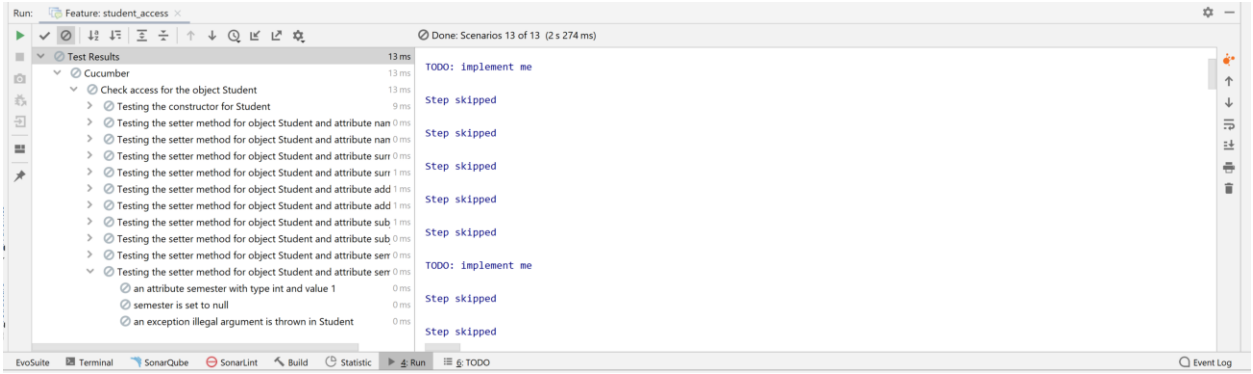

Figure G.3: Pending steps

G.1.7 Step 5: Implement the Java classes

Now implement the corresponding Java classes into your project.

Create the directory *main* with a directory *java*. Add the package *exampledomain* to the *src/main/java* directory and add the implementation for the *StudentEntity* and the *StudentIDValueObject*.

Below is an example of the implementation, that is used in the next steps. If you want you can copy the code or write your own implementation.

```java
//src/main/java/exampledomain/StudentEntity.java
public class StudentEntity {
    private final StudentIDValueObject id; //!changeable
    private String name; //required
    private String surname; //required
    private String address;
    private String subject;
    private final BigInteger martikelNumber; //!changeable
    private int semester; //range="1,14"
```

```
public StudentEntity(StudentIDValueObject id, String name,
     String surname, BigInteger martikelNumber) {
    this.id = id;
    this.name = name;
    this.surname = surname;
    this.martikelNumber = martikelNumber;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getSurname() {
    return surname;
}

public void setSurname(String surname) {
    this.surname = surname;
}

public String getAddress() {
    return address;
}

public void setAddress(String address) {
    this.address = address;
}

public String getSubject() {
    return subject;
}

public void setSubject(String subject) {
    this.subject = subject;
}
```

```
    public int getSemester() {
        return semester;
    }

    public void setSemester(int semester) {
        this.semester = semester;
    }

    public StudentIDValueObject getId() {
        return id;
    }

    public BigInteger getMartikelNumber() {
        return martikelNumber;
    }
}

//src/main/java/exampledomain/StudentIDValueObject.java
public class StudentIDValueObject {
    private final String id;

    public StudentIDValueObject(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }
}
```

G.1.8 Step 6: Implement the steps definitions

Now implement the step definitions. The step definitions are the connection between the Gherkin feature file and your Java code. Further informations about step definitions[12]. There are several different ways to implement your step definitions. This tutorial uses a method that has proven successful in our tests.

Hint: Cucumber only supports the following types: `{int}`, `{float}`, `{word}`, `{string}`, `{}`. In the JVM the following types are also supported: `biginteger`, `bigdecimal`, `byte`, `short`, `long` and `double`. Further informations about parameter types[13]

12 https://cucumber.io/docs/cucumber/step-definitions/
13 https://cucumber.io/docs/cucumber/cucumber-expressions/#parameter-types

Start by defining the variables student, studentId and a list of exceptions as instance variables of the class *StudentSteps*.

```
//src/test/java/exampledomain/StudentSteps.java
private StudentEntity student;
private StudentIDValueObject studentId;
private List<Exception> exceptionsList = new LinkedList<>();
```

Then define the first step. The first Given step uses a object with the type, who is not supported by Cucumber. The defined variable studentId is used to define this object in the step definiton context. The input string is used to make the constructor call of the studentId element.

```
//src/test/java/exampledomain/StudentSteps.java
@Given("an attribute id {string} with type StudentId")
public void an_attribute_id_with_type_StudentId(String string) {
    studentId = new StudentIDValueObject(string);
}
```

Then implement the When statement. First replace the int with the type biginteger in the When statement and the Integer with the BigInteger in the parameter list. We recommend you to rename the parameter list, to make the code understandable. After that, call the constructor from Student. The first element of the constructor is the id, it is replaced with previously defined studentId. The other parameters are used directly because they are supported by Cucumber.

Warning: make sure, that you set the parameters in the correct ordering when creating the new Student object. The Gherkin feature file and your code probably have not the same order. Take a look at the *student_access.feature* file to check how the order of the inputs parameters is in the When statement.

```
//src/test/java/exampledomain/StudentSteps.java
@When("the constructor Student is called with the values: {string},
{string}, {biginteger}, {string}")
public void the_constructor_student_is_called_with_the_values(String  name,
     String surname, BigInteger martikelNumber, String studentIdPlaceholder) {
    student = new StudentEntity(studentId, name, surname, martikelNumber);
}
```

Then implement the Then statements. Use the *assertEquals()* methods from Junit to check if the values are correctly set.

The type biginteger is not directly supported from the Cucumber, it is only supported in the JVM. So you have to change the type of the martikelNumber to to biginteger in the Then statement `@Then("the attribute martikelNumber is equals {biginteger}")` and in

the parameter list `public void the_attribute_martikelNumber_is_equals(BigInteger int1)`.

The StudentId will be checked against the predefined *studentId*.

```
//src/test/java/exampledomain/StudentSteps.java
@Then("the attribute name is equals {string}")
public void the_attribute_name_is_equals(String string) {
     assertEquals(string, student.getName());
}

@Then("the attribute surname is equals {string}")
public void the_attribute_surname_is_equals(String string) {
    assertEquals(string, student.getSurname());
}

@Then("the attribute martikelNumber is equals {biginteger}")
public void the_attribute_martikelNumber_is_equals(BigInteger int1) {
    assertEquals(int1, student.getMartikelNumber());
}

@Then("the attribute id is equals {string}")
public void the_attribute_id_is_equals(String string) {
    assertEquals(studentId, student.getId());
}
```

Now you can run the tests again (`mvn test` or `gradle cucumber`) and the first three scenario will pass. Congratulations.

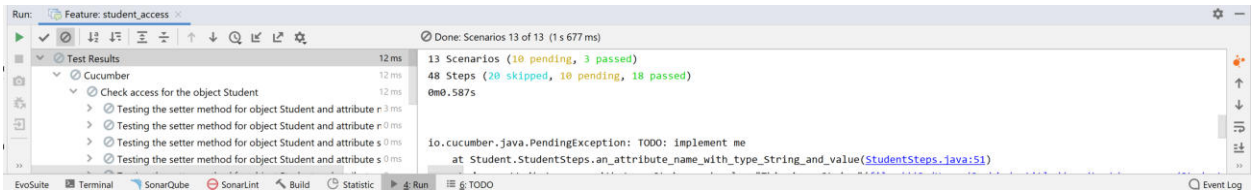

Figure G.4: Three passed scenarios

Continue with further steps. The Given statement creates a student object, with the name string and some random/default values for the other attributes. In the first When statement the name is set to the input string. In the second When statement the name is set to null, this method is surrounded by a try and catch clause. When the name is set to null and an exception is thrown, then it is added to the the exceptionList. The try and catch clause is used as a workaround to check if the errors were correctly thrown. The Then statement is able to check if the right exception was thrown by testing the elements in the exceptionList against the correct error type.

```
//src/test/java/exampledomain/StudentSteps.java
@Given("an attribute name with type String and value {string}")
public void an_attribute_name_with_type_String_and_value(String string) {
        student = new StudentEntity(new StudentIDValueObject("123"), string,
                "surname", new BigInteger("2233"));
}

@When("name is set to {string}")
public void name_is_set_to(String string) {
    student.setName(string);
}

@When("name is set to null")
public void name_is_set_to_null() {
    try{
        student.setName(null);
    }catch (Exception e){
        exceptionsList.add(e);
    }
}

@Then("an exception illegal argument is thrown in Student")
public void an_exception_illegal_argument_is_thrown_in_student() {
    assertEquals(exceptionsList.get(0).getClass(), IllegalArgumentException.class);
}
```

One test will fail. Take a look at the implementation from *setName()*. Do you find the error?

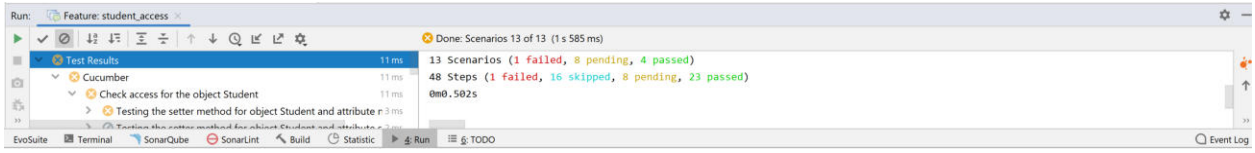

Figure G.5: Failing scenario

The following code can be added to the *setName()* method to make the test pass.

```
//src/main/java/exampledomain/Student.java
public void setName(String name) {
    if (name == null) {
        throw new IllegalArgumentException();
    }
    this.name = name;
}
```

The scenario will pass.

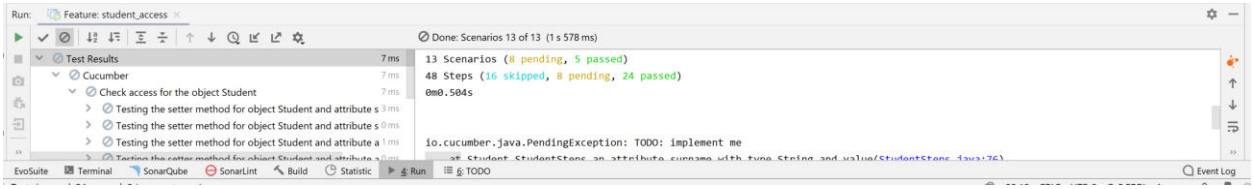

Figure G.6: Five passed scenarios

Implement the further steps analogous to the first tests. Below is an example of the whole *StudentSteps* file. Do not forget to implement the null check in the elements, who are not defined as nullable. **Hint:** Maybe the Java Compiler gives you an error in the null check statements. You can delete the test in the Gherkin feature file or write it as a comment and then add manual the exception to the exceptionsList. It depends if you want the Gherkintests completely or write as little code as possible.

Below is the fully implemented *StudentSteps* file.

```
//src/test/java/exampledomain/StudentSteps.java
public class StudentSteps {
    private StudentEntity student;
    private StudentIDValueObject studentId;
    private List<Exception> exceptionsList = new LinkedList<>();

    @Given("an attribute id {string} with type StudentId")
    public void an_attribute_id_with_type_StudentId(String string) {
        studentId = new StudentIDValueObject(string);
    }

    @When("the constructor Student is called with the values: {string},
      {string}, {biginteger}, {string}")
    public void the_constructor_student_is_called_with_the_values(String string,
     String string2, BigInteger int1, String string3) {
        student = new StudentEntity(studentId, string, string2, int1);
    }

    @Then("the attribute name is equals {string}")
    public void the_attribute_name_is_equals(String string) {
        assertEquals(string, student.getName());
    }

    @Then("the attribute surname is equals {string}")
    public void the_attribute_surname_is_equals(String string) {
        assertEquals(string, student.getSurname());
```

```
}

@Then("the attribute martikelNumber is equals {biginteger}")
public void the_attribute_martikelNumber_is_equals(BigInteger int1) {
    assertEquals(int1, student.getMartikelNumber());
}

@Then("the attribute id is equals {string}")
public void the_attribute_id_is_equals(String string) {
    assertEquals(studentId, student.getId());
}

@Given("an attribute name with type String and value {string}")
public void an_attribute_name_with_type_String_and_value(String string) {
    student = new StudentEntity(new StudentIDValueObject("123"), string,
            "surname", new BigInteger("2233"));
}

@When("name is set to {string}")
public void name_is_set_to(String string) {
    student.setName(string);
}

@When("name is set to null")
public void name_is_set_to_null() {
    try{
        student.setName(null);
    }catch (Exception e){
        exceptionsList.add(e);
    }
}

@Then("an exception illegal argument is thrown in Student")
public void an_exception_illegal_argument_is_thrown_in_student() {
    assertEquals(exceptionsList.get(0).getClass(),
     IllegalArgumentException.class);
}

@Given("an attribute surname with type String and value {string}")
public void an_attribute_surname_with_type_String_and_value(String string) {
    student = new StudentEntity(new StudentIDValueObject("123"), "name",
            string, new BigInteger("2233"));
}
```

```
@When("surname is set to {string}")
public void surname_is_set_to(String string) {
    student.setSurname(string);
}

@When("surname is set to null")
public void surname_is_set_to_null() {
    try{
        student.setSurname(null);
    }catch (Exception e){
        exceptionsList.add(e);
    }
}

@Given("an attribute address with type String and value {string}")
public void an_attribute_address_with_type_String_and_value(String string) {
    student = new StudentEntity(new StudentIDValueObject("123"), "name",
            "surname", new BigInteger("2233"));
    student.setAddress(string);
}

@When("address is set to {string}")
public void address_is_set_to(String string) {
    student.setAddress(string);
}

@Then("the attribute address is equals {string}")
public void the_attribute_address_is_equals(String string) {
    assertEquals(string, student.getAddress());
}

@When("address is set to null")
public void address_is_set_to_null() {
    try{
        student.setAddress(null);
    }catch (Exception e){
        exceptionsList.add(e);
    }
}
```

```
@Given("an attribute subject with type String and value {string}")
public void an_attribute_subject_with_type_String_and_value(String string) {
    student = new StudentEntity(new StudentIDValueObject("123"), "name",
            "surname", new BigInteger("2233"));
    student.setSubject(string);
}

@When("subject is set to {string}")
public void subject_is_set_to(String string) {
    try{
        student.setSubject(string);
    }catch (Exception e){
        exceptionsList.add(e);
    }
}

@Then("the attribute subject is equals {string}")
public void the_attribute_subject_is_equals(String string) {
    assertEquals(string, student.getSubject());
}

@When("subject is set to null")
public void subject_is_set_to_null() {
    try{
        student.setSubject(null);
    }catch (Exception e){
        exceptionsList.add(e);
    }
}
@Given("an attribute semester with type int and value {int}")
public void anAttributeSemesterWithTypeIntAndValue(int int1) {
    student = new StudentEntity(new StudentIDValueObject("123"), "name",
            "surname", BigInteger.valueOf(1));
    student.setSemester(int1);
}

@When("semester is set to {int}")
public void semesterIsSetTo(int int1) {
    student.setSemester(int1);
}
```

```
    @Then("the attribute semester is equals {int}")
    public void theAttributeSemesterIsEquals(Integer int1) {
        assertEquals(int1, (Integer)student.getSemester());
    }

    @When("semester is set to null")
    public void semesterIsSetToNull() {
        //student.setSemester(null) Compiler Error
        exceptionsList.add(new IllegalArgumentException());
    }
}
```

G.1.9 Step 7: Repeat steps 3-6 for check access for the object StudentId

Repeat the steps three to six for the second Gherkin feature file *Check access for the object StudentId*.

- Create the Gherkin feature file *studentId_access.feature* and copy the Gherkin feature file in your project. (Step 3)
- Create the file *StudentIdSteps.java* in *src/test/java/exampledomain* and add the pending step definitions to *StudentIdSteps.java*. (Step 4)
- The code is already implemented. (Step 5)
- Implement the step definitions analogous to the steps above. Use an instance variable to save the StudentIDValueObject between the different steps and check with *assertEquals()* if the id is the same. The code is shown below. (Step six)

```
//src/test/java/exampledomain/StudenIdSteps.java
public class StudentIdSteps {
    private StudentIDValueObject studentIDValueObject;

    @When("the constructor StudentId is called with the values: {string}")
    public void the_constructor_studentid_is_called_with_the_values(String string) {
        studentIDValueObject = new StudentIDValueObject(string);
    }
    @Then("the attribute id in StudentId is equals {string}")
    public void the_attribute_id_in_StudentId_is_equals(String string) {
        assertEquals(string, studentIDValueObject.getId());
```

```
    }
}
```

G.1.10 Step 8: Create the test range check for mutable attributes

In the eighth step you will generate and implement the range check for the *semester* attribute, who must be between one and 14.

- Repeat the step two with option *Range check for mutable attributes*. The tool will generate a Gherkin feature file called *Range check for semester*.
- Create the Gherkin feature file in your project with name *student_semester_range.feature* and copy the generated file.
- Add the pending step definitions *StudentSteps.java*.
- The step definitions are already implemented but you have to add a try and catch clause to the When statement `semster is set to {int}` because you want to catch the illegal argument exception, when the semester value is outside of the range from one to 14.

```
//src/test/java/exampledomain/StudentSteps.java
@When("semester is set to {int}")
    public void semesterIsSetTo(int int1) {
        try{
            student.setSemester(int1);
        }catch (Exception e){
            exceptionsList.add(e);
        }
}
```

- When you run the tests (`mvn test` or `gradle cucumber`), three scenarios pass and four scenarios fail.

- Correct the error in your code an make the other scenarios pass.

```
//src/main/java/exampledomain/StudentEntity.java
public void setSemester(int semester) {
    if(semester < 1 || semester > 14){
        throw new IllegalArgumentException();
```

```
        }
        this.semester = semester;
    }
```

- Well done!

G.1.11 Step 9: Create the test 1:1 association

In the last step you will create the 1:1 asscoiation test for the reference StudentId in the Entity Student.

- Repeat the step two with option *1:1 relationships*. The tool will generate a Gherkin feature file called *Testing the 1:1 Reference for Student:StudentId*.
- Add the Gherkin feature file to your project with the name *student_studentId_reference.feature*.
- Add the pending step definitions to the *StudentSteps.java* file.
- Implement a class *Repository* to simulate the database of your application. It is not worthwhile to connect a database in this small example, for this reason we simulate it, as we only want to show how the corresponding step definitions can be implemented. Before you start implementing take a look at the step definitions: there is one new method required the *deleteStudent()* method. You also have to make sure, that you could add the *Student* and the *StudentId* to your *Repository* and check if they are still existing in the *Repository*. A possible implementation of the *Repository* is shown below:

```
//src/main/java/exampledomain/Repository.java
public class Repository {
    private static Repository repositoryInstance;
    public String value;
    List<StudentIDValueObject> studentIds = new LinkedList<>();
    List<StudentEntity> studentEntities = new LinkedList<>();

    private Repository(String value) {
        // The following code emulates slow initialization.
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        this.value = value;
    }

    public static Repository getInstance(String value) {
        if (repositoryInstance == null) {
```

```
            repositoryInstance = new Repository(value);
        }
        return repositoryInstance;
    }

    public void createStudent(StudentEntity student) {
        studentEntities.add(student);
        studentIds.add(student.getId());
    }

    public List<StudentIDValueObject> readStudentIds() {
        return studentIds;
    }
    public void deleteStudent(StudentEntity student) {
            //remove student
            removeStudent(student);
            //remove studentId
            removeId(student.getId());

    }
    private void removeStudent(StudentEntity studentEntity) {
        studentEntities.remove(studentEntity);
    }
    private void removeId(StudentIDValueObject id) {
        studentIds.remove(id);
    }
```

- After you have implemented the *Repository* program the steps definitions. In the Given statement create a *Student* and a *StudentId* with some default values and then add it to your *Repository*. Delete the *Student* from the *Repository* in the When statement and check if the corresponding *StudentId* from the *Student*, who was deleted is also deleted in the *Repository*. An example implementation is shown below.

```
//src/test/java/exampledomain/StudentSteps.java
@Given("a Student and a StudentId")
public void a_Student_and_a_StudentId() {
    studentId = new StudentIDValueObject("12344");
    student = new StudentEntity(studentId, "name", "surname",
   BigInteger.valueOf(1));
    repository.createStudent(student);
}

@When("Student is deleted")
public void student_is_deleted() {
    repository.deleteStudent(student);
}

@Then("StudentId is deleted as well")
public void studentid_is_deleted_as_well() {
     assertFalse(repository.getStudentIds().contains(studentId));
}
```

- When you run your tests (`mvn test` or `gradle cucumber`) the scenario will pass. Congratulations your first project runs with Cucumber!

For further information visit our Advanced Tutorial. It explains you how to generate the tests for your Aggregate lifecycle state, 1:n relationships and further features.

G.2 Advanced Tutorial: DDD-2-BDD

In this tutorial we explain how to define the step defintions for the DDD-2-BDD tests.

Overview

G.2.1 Example domain

In this tutorial we work with an example domain *Essay*. The domain represents a part of an essay submission and marking portal. The Context Mapper model(CML) is shown below. The example is inspired by a blogpost from John Ferguson Smart[14].

```
BoundedContext Essay_Domain{
    Aggregate Essay {
    knowledgeLevel = CONCRETE

    Entity Essay {
        aggregateRoot
        - @EssayId identifier !changeable
        String title required
        String essayBody required
        - @Mark marks
        - Collection<@Author> authors size="max=5" cascade="persists"
    }
```

14 https://johnfergusonsmart.com/feature-mapping-a-simpler-path-from-stories-to-executable-acceptance-criteria/

```
  enum States {
      aggregateLifecycle
      CREATED, SUBMITTED, OPENED, MARKED, FINALISED, RETURNED
  }

  ValueObject EssayId {
      int essayId
  }
  Entity Author {
      int authorId required
      String name length="150" required
      String className
      -List<@Essay> publicatedEssays cascade="persists"
      -List<@Essay> pendingEssayList cascade="persists,remove" size="10"

      def @EssayId createEssay(@Essay essay) : write [ -> CREATED];
      def void addCoAuthor(@Author coauthor): read-only;
      def void updateEssay(@Essay essay): read-only;
      def void submitEssay(@Essay essay) : write [ CREATED, RETURNED-> SUBMITTED];
  }
  ValueObject Mark{
      !immutable
      int reasoningScore range="1,10"
      int relevanceScore range="1,10,'must be between 1 - 10'"
      int languageScore range="min=1,max=10,message='must be between 1 - 10'"
  }
  Entity Corrector{
      int correctorId required
      String name required
      def @Essay openEssay(@EssayId essayId): write [SUBMITTED -> OPENED];
      def void markEssay(@EssayId essayId, @Mark mark): write [OPENED -> MARKED];
      def @Essay returnEssay(@EssayId essayId): write[MARKED ->RETURNED];
      def void finaliseEssay(@EssayId essayId): write[MARKED->FINALISED*];
  }
  Service EssayService {
      boolean canFinalise(@EssayId essayId): read-only;
      List<@Essay> getAllEssaysFromAuthor(@Author author): read-only;
      @Essay getEssayWithId(@Essay essayId): read-only;
}
```

```
    }
}


UserStory UC01_author{
    As an "Author"
    I want to create a "essay"
    I want to "add" an "coauthor"
    I want to read a "grades"
    I want to update an "essay"
    I want to read a "pendingessay"
    so that "I could manage all my essay in the tool and see my grades."
}
```

The domain has a RootEntity *Essay* and two Entities that represents the *Author* and *Corrector*. The domain has also two ValueObjects one for the *Grades* and one for the *EssayId*. The *Essay* has a lifecycle, that is defined in the enum *State* and marked with `aggregateLifecycle`. The different methods in *Author* and *Corrector* can change the aggregate State. They are marked with the keyword `write` and the corresponding state changes. The *Essay* has two `required` attributes: title and essayBody. It also has a reference to the *EssayId* who is `!changeable` and a reference to the ValueObject *Mark*. The authors of an *Essay* are contained in the list *authors* and can not be more than 5. The *authors* list is defined as `cascade="persists"`.
The *Author* has an authorId and a name, who both are required. It also has a classname and two List of *Essays*: the *publicatedEssays* and the *pendingEssayList*. The author has four methods: createEssay, addCoAuthor, updateEssay and submitEssay.
The *Corrector* has an correctorId and a name, who both are required. It also has four writing methods: openEssay, markEssay, returnEssay and finaliseEssay.
The ValueObject *Mark* is defined as `!immutable` and therefore the attributes have getter and setter methods. All three attributes: reasoningScore, relevanceScore and languageScore must be between one and ten.
The *EssayService* contains further read-only methods.

Warning: Each step definition must be unique. If two attributes have the same name, the step definitions are equal and it produces errors in the step definition implementations. Make sure that the name of the attributes are unique.

G.2.2 Implementation

Setup a new project like explained in the beginner tutorial and name it *essayexampleproject*. Below is a possible implementation for the example domain shown. You can copy the implementation to the folder *src/main/java/essayexampleproject/* or write your own code. The *Repository* is used to test the relationships.

Author

```
public class Author {
    private int authorId; //required
    private String name; //required
    private String className;
    private List<Essay> publicatedEssays = new LinkedList<>();
    //cascade="persists"
    private List<Essay> pendingEssayList = new LinkedList<>();
    // cascade="persists,remove"
    private List<Essay> allEssays = new LinkedList<>();

    public Author(int authorId, String name) {
        this.authorId = authorId;
        this.name = name;
    }
    //getter and setter for authorId, name, className,publicatedEssays,
     pendingEssayList, allEssays
    public Essay createEssay(String title, String essayBody){
        Essay essay = new Essay(new EssayId(), title, essayBody);
        if(essay.getState()!=null){
            throw new IllegalStateException();
        }
        allEssays.add(essay);
        essay.addAuthor(this);
        essay.setState(State.CREATED);
        return essay; //: write [ -> CREATED];
    }
    public void addCoAuthor(Author coauthor, EssayId id){
       //void
    }
    public void updateEssay(Essay essay){
        if(!essay.getState().equals(State.CREATED)){
            throw new IllegalStateException();
        }
    }
    public void submitEssay(Essay essay) {
        if(!(essay.getState().equals(State.CREATED)
          || essay.getState().equals(State.RETURNED))){
            throw new IllegalStateException();
        }
        essay.setState(State.SUBMITTED);//: write [ CREATED, RETURNED-> SUBMITTED];
    }
```

```
    public void deleteAuthor(Repository repository){
        //remove author
        repository.removeAuthor(this);
        // remove corresponding essays
        for (Essay essay: pendingEssayList) {
            repository.removeEssay(essay);
        }
}
```

Corrector

```
public class Corrector {
    private int correctorId; //required
    private String name; //required

    public Corrector(int correctorId, String name) {
        this.correctorId = correctorId;
        this.name = name;
    }
        //getter and setter for correctorId, name
    public void openEssay(Essay essay) {
         // write [SUBMITTED -> OPENED];
    }

    public void markEssay(Essay essay, Mark mark) {
        //write [OPENED -> MARKED];
    }

    public Essay returnEssay(Essay essay) {
        return essay; //write[MARKED ->RETURNED];
    }

    public void finaliseEssay(Essay essay) {
                //write[MARKED -> FINALISED]
    }
```

Essay

```
public class Essay {
    private final EssayId essayID; //!changeable
    private String title;//required not nullable
    private String essayBody; //required not nullable
    private Mark mark;
    private List<Author> authors = new LinkedList<>();
    //size="max='5'" cascade="persists"
    private State state;

    public Essay(EssayId essayID, String title, String essayBody) {
        this.essayID = essayID;
        this.title = title;
        this.essayBody = essayBody;
    }
        //getter for essayId
    //getter and setter for title, essayBody, mark, authors, state
    public void addAuthor(Author coauthor) {
        authors.add(coauthor);
    }
    public void deleteAuthor(Author author){
        authors.remove(author);
    }

    public void deleteEssay(Repository repository){
        //remove essay
        repository.removeEssay(this);
        //remove essay from pendingList
        repository.removeMark(this.mark);
        for(Author author: repository.getAuthors()){
            author.getPendingEssayList().remove(this);
            author.getPublicatedEssays().remove(this);
        }
         }
}
```

EssayId

```
public class EssayId {
    private static int idCounter = 1;
    private final Integer id;
    public EssayId() {
        this.id = idCounter++;
    }
    public Integer getId() {
        return id;
    }
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        EssayId essayId = (EssayId) o;
        return id.equals(essayId.id);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);
    }
}
```

EssayService

```
public class EssayService {
    public boolean canFinalise(EssayId essayId){
        return false; //read-only:
    }
    List<Essay> getAllEssaysFromAuthor(Author author){
        return null; //read-only;
    }
    Essay getEssayWithId(Essay essayId){
        return null; //read-only;
    }
}
```

Mark

```
public class Mark {
    private int reasoningScore;
    private int relevanceScore;
    private int languageScore;

    public Mark(int reasoningScore, int relevanceScore, int languageScore) {
        this.reasoningScore = reasoningScore;
        this.relevanceScore = relevanceScore;
        this.languageScore = languageScore;
    }
        //getter and setter for reasoningScore, relevanceScore, languageScore
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Mark mark = (Mark) o;
        return reasoningScore == mark.reasoningScore &&
                relevanceScore == mark.relevanceScore &&
                languageScore == mark.languageScore;
    }

    @Override
    public int hashCode() {
        return Objects.hash(reasoningScore, relevanceScore, languageScore);
    }
}
```

State

```
public enum State {
    CREATED, SUBMITTED, OPENED, MARKED, FINALISED, RETURNED
}
```

Repository

```
public final class Repository {
    private static Repository repositoryInstance;
    public String value;
    List<Essay> essays = new LinkedList<>();
    List<Author> authors = new LinkedList<>();
    List<Mark> marks = new LinkedList<>();
    List<EssayId> essayIds = new LinkedList<>();
```

```
private Repository(String value) {
    // The following code emulates slow initialization.
    try {
        Thread.sleep(1000);
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
    this.value = value;
}

public static Repository getInstance(String value) {
    if (repositoryInstance == null) {
        repositoryInstance = new Repository(value);
    }
    return repositoryInstance;
}
public List<Essay> getEssays() {
    return essays;
}

public void setEssays(List<Essay> essays) {
    this.essays = essays;
    for(Essay essay: essays){
        authors.addAll(essay.getAuthors());
    }
}
    //getter and setter for authors, marks, essayIds
public void addAuthor(Author author){
    this.authors.add(author);
}
public void removeAuthor(Author author){
    this.authors.remove(author);
    for(Essay essay: essays){
        if(essay.getAuthors().contains(author)){
            List<Author> newAuthors= essay.getAuthors();
            newAuthors.remove(author);
            essay.setAuthors(newAuthors);
        }
    }
}
public void addEssay(Essay essay){
    this.essays.add(essay);
    this.essayIds.add(essay.getId());
```

```
        this.authors.addAll(essay.getAuthors());
    }
    public void removeEssay(Essay essay){
        this.essayIds.remove(essay.getId());
        this.essays.remove(essay);
    }
    public void addMark(Mark mark){ this.marks.add(mark);}
    public void removeMark(Mark mark){this.marks.remove(mark);}

    public Essay getById(EssayId essayId){
        for (Essay essay: essays) {
            if (essay.getId().equals(essayId)) {
                return essay;
            }
        }
        return null;
    }
}
```

G.2.3 Testing the constructor and setter methods for Entities and ValueObjects

- Generate the constructor and setter methods tests with the DDD-2-BDD tool.
- Import the feature file for the essay access test with the name *essay_access.feature* in *src/test/resources/exampleessayproject*.
- Rename the file *StepsDefintions.java* to *EssaySteps.java* in *src/test/java/esampleessayproject*.
- Now implement the step definitions like explained in the beginner tutorial. Make the test pass. Below is a possible implementation of the *EssaySteps.java* file shown.

```
//src/test/java/essayexampleproject/EssaySteps.java
    @Given("an attribute identifier {string} with type EssayId")
    public void an_attribute_identifier_with_type_essay_id(String string) {
        essayId = new EssayId();
    }
```

```
@When("the constructor Essay is called with the values: {string},
 {string}, {string}")
public void the_constructor_essay_is_called_with_the_values(String string,
String string2, String string3) {
    essay = new Essay(essayId, string, string2);
}

@Then("the attribute title in Essay is equals {string}")
public void the_attribute_title_in_essay_is_equals(String string) {
    assertEquals(essay.getTitle(), string);
}

@Then("the attribute essayBody in Essay is equals {string}")
public void the_attribute_essay_body_in_essay_is_equals(String string) {
    assertEquals(essay.getEssayBody(), string);
}

@Then("the attribute identifier in Essay is equals {string}")
public void the_attribute_identifier_in_essay_is_equals(String string) {
    assertEquals(essayId, essay.getId());
}

@Given("an attribute title with type String and value {string}")
public void an_attribute_title_with_type_string_and_value(String string) {
    essay = new Essay(new EssayId(), string, "body");
}

@When("title is set to {string}")
public void title_is_set_to(String string) {
    essay.setTitle(string);
}

@When("title is set to null")
public void title_is_set_to_null() {
    try {
        essay.setTitle(null);
    } catch (Exception e) {
        exceptionList.add(e);
    }
}

@Then("an exception illegal argument is thrown in Essay")
public void an_exception_illegal_argument_is_thrown_in_essay() {
```

```
    assertEquals(IllegalArgumentException.class,
     exceptionList.get(0).getClass());
}

@Given("an attribute essayBody with type String and value {string}")
public void an_attribute_essay_body_with_type_string_and_value(
 String string) {
    essay = new Essay(new EssayId(), "body", string);
}

@When("essayBody is set to {string}")
public void essay_body_is_set_to(String string) {
    essay.setEssayBody(string);
}

@When("essayBody is set to null")
public void essay_body_is_set_to_null() {
    try {
        essay.setEssayBody(null);
    } catch (Exception e) {
        exceptionList.add(e);
    }
}

@Given("an attribute marks with type Mark and value {string}")
public void an_attribute_marks_with_type_mark_and_value(String string) {
    essay = new Essay(new EssayId(), "title", "body");
    essay.setMark(new Mark(8, 8, 8));
}

@When("marks is set to {string}")
public void marks_is_set_to(String string) {
    mark = new Mark(7, 7, 7);
    essay.setMark(mark);

}

@Then("the attribute marks in Essay is equals {string}")
public void the_attribute_marks_in_essay_is_equals(String string) {
    assertEquals(mark, essay.getMark());
}

@When("marks is set to null")
```

```
public void marks_is_set_to_null() {
    try {
        essay.setMark(null);
    } catch (Exception e) {
        exceptionList.add(e);
    }
}

@Given("an attribute authors with type Collection<Author>
 and value {string}")
public void an_attribute_authors_with_type_collection_author
 _and_value(String string) {
    essay = new Essay(new EssayId(), "title", "body");
    List<Author> authorList = new LinkedList<>();
    authorList.add(new Author(10, "mustermann"));
    essay.setAuthors(authorList);
}

@When("authors is set to {string}")
public void authors_is_set_to(String string) {
    authorList.add(new Author(10, "Anna"));
    authorList.add(new Author(11, "Mario"));
    essay.setAuthors(authorList);
}

@Then("the attribute authors in Essay is equals {string}")
public void the_attribute_authors_in_essay_is_equals(String string) {
    assertEquals(authorList, essay.getAuthors());
}

@When("authors is set to null")
public void authors_is_set_to_null() {
    try {
        essay.setAuthors(null);
    } catch (Exception e) {
        exceptionList.add(e);
    }
}
```

- Repeat the steps above for the *author*, *corrector*, *essayId* and *mark*. Create for each element a new step definition file.

```
//src/test/java/essayexampleproject/AuthorSteps.java
    @When("the constructor Author is called with the values:
     {int}, {string}")
    public void the_constructor_author_is_called_with_the_values
     (Integer int1, String string) {
        author = new Author(int1, string);
    }

    @Then("the attribute authorId in Author is equals {int}")
    public void the_attribute_author_id_in_author_is_equals(int int1) {
        assertEquals(int1, author.getAuthorId());
    }

    @Then("the attribute name in Author is equals {string}")
    public void the_attribute_name_in_author_is_equals(String string) {
        assertEquals(string, author.getName());
    }

    @Given("an attribute authorId with type int and value {int}")
    public void an_attribute_author_id_with_type_int_and_value(Integer int1) {
        author = new Author(int1, "max muster");
    }

    @When("authorId is set to {int}")
    public void author_id_is_set_to(Integer int1) {
        author.setAuthorId(int1);
    }

    @When("authorId is set to null")
    public void author_id_is_set_to_null() {
        //author.setAuthorId(null)
        exceptionList.add(new IllegalArgumentException());
    }

    @Then("an exception illegal argument is thrown in Author")
    public void an_exception_illegal_argument_is_thrown_in_author() {
        assertEquals(IllegalArgumentException.class,
          exceptionList.get(0).getClass());
    }

    @Given("an attribute name with type String and value {string}")
```

```
public void an_attribute_name_with_type_string_and_value(String string) {
    author = new Author(101, string);
}

@When("name is set to {string}")
public void name_is_set_to(String string) {
    author.setName(string);
}

@When("name is set to null")
public void name_is_set_to_null() {
    try {
        author.setName(null);
    } catch (Exception e) {
        exceptionList.add(e);
    }
}

@Given("an attribute className with type String and value {string}")
public void an_attribute_class_name_with_type_string_and_value
 (String string) {
    author = new Author(101, "max muster");
    author.setClassName(string);
}

@When("className is set to {string}")
public void class_name_is_set_to(String string) {
    author.setClassName(string);
}

@Then("the attribute className in Author is equals {string}")
public void the_attribute_class_name_in_author_is_equals(String string) {
    assertEquals(author.getClassName(), string);
}

@When("className is set to null")
public void class_name_is_set_to_null() {
    try {
        author.setClassName(null);
    } catch (Exception e) {
        exceptionList.add(e);
    }
}
```

```
@Given("an attribute publicatedEssays with type List<Essay>
  and value {string}")
public void an_attribute_publicated_essays
_with_type_list_essay_and_value(String string) {
    author = new Author(101, "max muster");
    List<Essay> essayList = new LinkedList<>();
    essayList.add(new Essay(new EssayId(), "title", "body"));
    author.setPublicatedEssays(essayList);
}

@When("publicatedEssays is set to {string}")
public void publicated_essays_is_set_to(String string) {
    publicatedEssays.add(new Essay(new EssayId(), "titleOne", "bodyOne"));
    publicatedEssays.add(new Essay(new EssayId(), "titleTwo", "bodyTwo"));
    author.setPublicatedEssays(publicatedEssays);
}

@Then("the attribute publicatedEssays in Author is equals {string}")
public void the_attribute_publicated_essays_in_author_is_equals
 (String string) {
    assertEquals(publicatedEssays, author.getPublicatedEssays());
}

@When("publicatedEssays is set to null")
public void publicated_essays_is_set_to_null() {
    try {
        author.setPublicatedEssays(null);
    } catch (Exception e) {
        exceptionList.add(e);
    }
}

@Given("an attribute pendingEssayList with type List<Essay>
 and value {string}")
public void an_attribute_pending_essay_list_with_type_list
 _essay_and_value(String string) {
    author = new Author(101, "max muster");
    List<Essay> essayList = new LinkedList<>();
    essayList.add(new Essay(new EssayId(), "title", "body"));
    author.setPendingEssayList(essayList);
}
```

```
    @When("pendingEssayList is set to {string}")
    public void pending_essay_list_is_set_to(String string) {
        pendingEssayList.add(new Essay(new EssayId(), "titleOne", "bodyOne"));
        pendingEssayList.add(new Essay(new EssayId(), "titleTwo", "bodyTwo"));
        author.setPendingEssayList(pendingEssayList);
    }

    @Then("the attribute pendingEssayList in Author is equals {string}")
    public void the_attribute_pending_essay_list_in_author_is_equals
     (String string) {
        assertEquals(pendingEssayList, author.getPendingEssayList());
    }

    @When("pendingEssayList is set to null")
    public void pending_essay_list_is_set_to_null() {
        try {
            author.setPendingEssayList(null);
        } catch (Exception e) {
            exceptionList.add(e);
        }
    }
```

//src/test/java/essayexampleproject/CorrectorSteps.java

```
    @When("the constructor Corrector is called with the values: {int},
      {string}")
    public void the_constructor_corrector_is_called_with_the_values
     (Integer int1, String string) {
        corrector = new Corrector(int1, string);
    }

    @Then("the attribute correctorId in Corrector is equals {int}")
    public void the_attribute_corrector_id_in_corrector_is_equals(int int1) {
        assertEquals(corrector.getCorrectorId(), int1);
    }

    @Then("the attribute correctorName in Corrector is equals {string}")
    public void the_attribute_name_in_corrector_is_equals(String string) {
        assertEquals(corrector.getName(), string);
    }

    @Given("an attribute correctorId with type int and value {int}")
    public void an_attribute_corrector_id_with_type_int_and_value
     (Integer int1) {
        corrector= new Corrector(int1, "name");
```

```
    }

    @When("correctorId is set to {int}")
    public void corrector_id_is_set_to(Integer int1) {
        corrector.setCorrectorId(int1);
    }

    @When("correctorId is set to null")
    public void corrector_id_is_set_to_null() {
        //corrector.set(null) -> compiler error
        exceptionList.add(new IllegalArgumentException());
    }

    @Then("an exception illegal argument is thrown in Corrector")
    public void an_exception_illegal_argument_is_thrown_in_corrector() {
        assertEquals(IllegalArgumentException.class
         , exceptionList.get(0).getClass());
    }
    @Given("an attribute correctorName with type String and value {string}")
    public void an_attribute_corrector_name_with_type_string_
    and_value(String string) {
        corrector= new Corrector(101, string);
    }

    @When("correctorName is set to {string}")
    public void corrector_name_is_set_to(String string) {
        corrector.setName(string);
    }

    @When("correctorName is set to null")
    public void corrector_name_is_set_to_null() {
        try{
            corrector.setName(null);
        }catch (Exception e){
            exceptionList.add(e);
        }
    }

//src/test/java/essayexampleproject/EssayIdSteps.java
    @When("the constructor EssayId is called with the values: {int}")
    public void the_constructor_essay_id_is_called_with_
     the_values(Integer int1) {
        essayId = new EssayId();
    }
```

```
    @Then("the attribute essayId in EssayId is equals {int}")
    public void the_attribute_essay_id_in_essay_id_is_equals(Integer int1) {
        //kind of useless test because the constructor is empty
        // delete it or leave it (that the tests are complete)
        assertEquals(essayId.getId(), essayId.getId());
    }

//src/test/java/essayexampleproject/MarkdSteps.java
    @Given("an attribute reasoningScore with type int and value {int}")
    public void an_attribute_reasoning_score_with_type_int_
     and_value(Integer int1) {
        mark=new Mark(int1, 5, 7);
    }

    @When("reasoningScore is set to {int}")
    public void reasoning_score_is_set_to(Integer int1) {
         mark.setReasoningScore(int1);
    }

    @Then("the attribute reasoningScore in Mark is equals {int}")
    public void the_attribute_reasoning_score_in_mark_is_equals(int int1) {
        assertEquals(int1, mark.getReasoningScore());
    }

    @When("reasoningScore is set to null")
    public void reasoning_score_is_set_to_null() {
        //mark.setReasoningScore(null)
        exceptionList.add(new IllegalArgumentException());
    }

    @Then("an exception illegal argument is thrown in Mark")
    public void an_exception_illegal_argument_is_thrown_in_mark() {
        assertEquals(IllegalArgumentException.class,
          exceptionList.get(0).getClass());
    }

    @Given("an attribute relevanceScore with type int and value {int}")
    public void an_attribute_relevance_score_with_type_int_
    and_value(Integer int1) {
        mark=new Mark(5, int1, 7);
    }

    @When("relevanceScore is set to {int}")
```

```
public void relevance_score_is_set_to(Integer int1) {
     mark.setRelevanceScore(int1);
}

@Then("the attribute relevanceScore in Mark is equals {int}")
public void the_attribute_relevance_score_in_mark_is_equals(int int1) {
    assertEquals(mark.getRelevanceScore(), int1);
}

@When("relevanceScore is set to null")
public void relevance_score_is_set_to_null() {
    //mark.setRelevanceScore(null)
    exceptionList.add(new IllegalArgumentException());
}

@Given("an attribute languageScore with type int and value {int}")
public void an_attribute_language_score_with_type_int_
 and_value(Integer int1) {
    mark=new Mark(5, 7, int1);
}

@When("languageScore is set to {int}")
public void language_score_is_set_to(Integer int1) {
    mark.setLanguageScore(int1);
}

@Then("the attribute languageScore in Mark is equals {int}")
public void the_attribute_language_score_in_mark_is_equals(int int1) {
    assertEquals(mark.getLanguageScore(), int1);
}

@When("languageScore is set to null")
public void language_score_is_set_to_null() {
    //mark.setLanguageScore(null)
    exceptionList.add(new IllegalArgumentException());
}
```

G.2.4 Range check for attributes

- Generate the range tests with the DDD-2-BDD tool.
- Import the feature files with the name *[object]_range.feature*.

- Run the the tests. Some step definitions are already implemented.
- Add the missing step definitions to the *MarkSteps.java* file.
- Also add a try and catch clause to the When statements which set the attributes with a range. (`@When("reasoningScore is set to {int}")`, `@When("relevanceScore is set to {int}")`, `@When("language is set to {int}")`)

```
//src/test/java/essayexampleproject/MarkSteps.java
   @When("reasoningScore is set to {int}")
    public void reasoning_score_is_set_to(Integer int1) {
        try{
            mark.setReasoningScore(int1);
        }catch (Exception e){
            exceptionList.add(e);
        }
    }
    @When("relevanceScore is set to {int}")
    public void relevance_score_is_set_to(Integer int1) {
        try{
            mark.setRelevanceScore(int1);
        }catch (Exception e){
            exceptionList.add(e);
        }
    }
    @When("languageScore is set to {int}")
    public void language_score_is_set_to(Integer int1) {
        try{
            mark.setLanguageScore(int1);
        }catch (Exception e){
            exceptionList.add(e);
        }
    }
```

- Now implement the pending step definitions as shown below.

```
//src/test/java/essayexampleproject/MarkSteps.java
@Then("an exception {string} is thrown in Mark")
public void an_exception_is_thrown_in_mark(String string) {
     assertEquals(exceptionList.get(0).getMessage(), string);
}
```

- Run the three test files. Some tests will fail. Now implement the ranges in the class *Mark*. A possible implementation is shown below:

```
//src/main/java/essayexampleproject/Mark.java
    public Mark(int reasoningScore, int relevanceScore, int languageScore) {
        this.setReasoningScore(reasoningScore);
        this.setRelevanceScore(relevanceScore);
        this.setLanguageScore(languageScore);
    }
           public void setReasoningScore(int reasoningScore) {
        if(reasoningScore < 1 || reasoningScore > 10){
            throw new IllegalArgumentException();
        }
        this.reasoningScore = reasoningScore;
    }
    public void setRelevanceScore(int relevanceScore) {
        if(relevanceScore < 1 || relevanceScore > 10){
            throw new IllegalArgumentException("must be between 1 - 10");
        }
        this.relevanceScore = relevanceScore;
    }
        public void setLanguageScore(int languageScore) {
        if(languageScore < 1 || languageScore > 10){
            throw new IllegalArgumentException("must be between 1 - 10");
        }
        this.languageScore = languageScore;
    }
```

- The tests will pass.

G.2.5 Size tests for collection

- Generate the collection size tests with the DDD-2-BDD tool.
- Import the feature files with the name *essay_authors_collection_size.feature* and *author_pendingEssayList_collection_size.feature*.
- Run the the tests of the *essay_authors_collection_size.feature*.
- Add the missing step definitions to the *EssaySteps.java* file.
- Implement the step definitions. Create in the Given statement a list with `{int}` authors and save it to a global *authorList* and set it to the *essay*. In the When statement remove the last element of the authorList and set the list new in the *essay*. In the Then statement check if the size of the *authorList* is equals the input size. In the second Then statement make sure that the correct error was thrown. In the second When statement add an author to the *authorList* and set the list new in *essay*.

```
//src/test/java/essayexampleproject/EssaySteps.java
    @Given("a collection authors with size {int}")
    public void a_collection_authors_with_size(Integer int1) {
        essay = new Essay(new EssayId(), "title", "body");
        for(int i = 0; i < int1; i++){
            authorList.add(new Author(i,"authorName"+i));
        }
        essay.setAuthors(authorList);
    }
    @When("an element is removed from the collection authors")
    public void an_element_is_removed_from_the_collection_authors() {
        try{
            authorList.remove(authorList.size()-1);
            essay.setAuthors(authorList);
        }catch (Exception e){
            exceptionList.add(e);
        }
    }
    @Then("the collection authors has the size {int}")
    public void the_collection_authors_has_the_size(int int1) {
        assertEquals(int1, essay.getAuthors().size());
    }
    @Then("an index out of bound exception is thrown in authors")
    public void an_index_out_of_bound_exception_is_thrown_in_authors() {
        assertEquals(IndexOutOfBoundsException.class,
          exceptionList.get(0).getClass());
    }
    @When("an element is added to the collection authors")
    public void an_element_is_added_to_the_collection_authors() {
        try{
            authorList.add(new Author(1,"name"));
            essay.setAuthors(authorList);
        }catch (Exception e){
            exceptionList.add(e);
        }
    }
}
```

- Run the test and they will fail. Implement the size check of the collection *authors* in *Essay.java*.

```
//src/main/java/essayexampleproject/Essay.java
public void setAuthors(List<Author> authors) {
        if (authors == null){
```

```
            throw new IllegalArgumentException();
        }
        if(authors.size()>5){
            throw new IndexOutOfBoundsException();
        }else{
            this.authors = authors;
        }
    }
```

- The tests pass.
- Run the the tests of the *author_pendingEssayList_collection_size.feature*.
- Add the missing step definitions to the *AuthorSteps.java* file.
- Implement the step definitions analogous to the previous collection size as shown below.

```
//src/test/java/essayexampleprojectn/AuthorSteps.java
        @Given("a collection pendingEssayList with size {int}")
    public void a_collection_pending_essay_list_with_size(Integer int1) {
        author = new Author(1, "Hanna");
        for(int i = 0; i < int1; i++){
            pendingEssayList.add(new Essay(new EssayId(), "title", "body"));
        }
        author.setPendingEssayList(pendingEssayList);
    }

    @When("an element is removed from the collection pendingEssayList")
    public void an_element_is_removed_from_the_collection_pending_
     essay_list() {
        try{
            pendingEssayList.remove(pendingEssayList.size()-1);
            author.setPendingEssayList(pendingEssayList);
        }catch (Exception e){
            exceptionList.add(e);
        }
    }

    @Then("the collection pendingEssayList has the size {int}")
    public void the_collection_pending_essay_list_has_the_size(int int1) {
        assertEquals(int1, author.getPendingEssayList().size());
    }

    @Then("an index out of bound exception is thrown in pendingEssayList")
    public void an_index_out_of_bound_exception_is_thrown_in_
```

```
 pending_essay_list() {
    assertEquals(IndexOutOfBoundsException.class,
      exceptionList.get(0).getClass());
}

@When("an element is added to the collection pendingEssayList")
public void an_element_is_added_to_the_collection_pending_essay_list() {
    try{
        pendingEssayList.add(new Essay(new EssayId(), "title", "body"));
        author.setPendingEssayList(pendingEssayList);
    }catch (Exception e){
        exceptionList.add(e);
    }
}
```

- Implement the corresponding code in the *Author.java* class.

```
//src/main/java/essayexampleproject/Author.java
    public void setPendingEssayList(List<Essay> pendingEssayList) {
        if(pendingEssayList == null){
            throw new IllegalArgumentException();
        }
        if(pendingEssayList.size()>10) {
            throw new IndexOutOfBoundsException();
        }
        this.pendingEssayList = pendingEssayList;
    }
```

- Run the tests and they will pass.

G.2.6 1:1 relationships

- Generate the 1:1 relationships tests with the DDD-2-BDD tool.
- Import the feature files with the name *essay_essayId_reference.feature* and *essay_mark_reference.feature*.
- Run the the tests of the *essay_mark_reference.feature*.
- Add the missing step definitions to the *EssaySteps.java* file.
- Implement the step definitions. Add a *respository* instance to the instance variables of the *EssaySteps* class. Also add a *newMark* variable to the *EssaySteps* class. In the Given statement create a new *mark* and a new *essay* and add the *essay* to the *respository*. In the first When statement delete the *essay*. In the Then statement check

if the mark is still containing in the *repository*. In the second When statement change the *mark* of the *essay* and save the new mark to the variable *newMark*. In the Then statment check if the *mark* is not in the *repository* and the *essay* has the *newMark*.

```
//src/test/java/essayexampleproject/EssaySteps.java
    @Given("a Essay and a Mark")
public void a_essay_and_a_mark() {
    mark = new Mark(2,3,4);
    essay = new Essay(new EssayId(), "title", "body");
    essay.setMark(mark);
    repository.addEssay(essay);
}
@When("Essay is deleted")
public void essay_is_deleted() {
    essay.deleteEssay(repository);
}
@Then("Mark is deleted as well")
public void mark_is_deleted_as_well() {
    assertFalse(repository.getMarks().contains(mark));
}
@When("Essay changes its reference to a new Mark")
public void essay_changes_its_reference_to_a_new_mark() {
    newMark = new Mark(5,5,5);
    essay.setMark(newMark);
}
@Then("the old Mark is deleted")
public void the_old_mark_is_deleted() {
    assertFalse(repository.getMarks().contains(mark));
    assertEquals(newMark, repository.getById(essay.getId()).getMark());
}
```

- Run the tests, they pass.
- Run the the tests of the *essay_essayId_reference.feature*.
- Add the missing step definitions to the *EssaySteps.java* file. (`@When("Essay is deleted")` is already implemented)
- Implement the step definitions. (The *essayId* is not changeable and therefore it has only one test scenario.)

```
@Given("a Essay and a EssayId")
public void a_essay_and_a_essay_id() {
    essayId = new EssayId();
    essay = new Essay(essayId, "title", "body");
    repository.addEssay(essay);
```

```
}
@Then("EssayId is deleted as well")
public void essay_id_is_deleted_as_well() {
    assertFalse(repository.getEssayIds().contains(essayId));
}
```

- Run the tests. They pass.

G.2.7 1:n aggregation relationships

- Generate the 1:n aggregation relationships tests with the DDD-2-BDD tool.
- Import the feature file with the name *author_publicatedEssays_reference.feature* and *essay_authors_reference.feature*.
- Run the the tests of the *author_publicatedEssays_reference.feature*. Some step definitions are already defined.
- Add the missing step definitions to the *AuthorSteps.java* file.
- Implement the step definitions. First add the *repository* and an object *essay* to the instance variables. In the Given statement create a new *author* and a new *essay*. Add the *essay* to the *publicatedEssay* list. Set the *publicatedEssay* list in *author*. Add the author and the essay to the repository. Delete the author in the When statement. In the Then statement check if the *publicatedEssays* are still in the repository.

```
//src/test/java/essayexampleproject/AuthorSteps.java
@Given("an entity Author with the references publicatedEssays")
public void an_entity_author_with_the_references_publicated_essays() {
    author = new Author(101, "testAuthor");
    essay=new Essay(new EssayId(), "title", "body");
    publicatedEssays.add(essay);
    author.setPublicatedEssays(publicatedEssays);
    repository.addAuthor(author);
    repository.addEssay(essay);
}
    @When("the entity Author is deleted")
public void the_entity_author_is_deleted() {
    author.deleteAuthor(repository);
}
@Then("the references publicatedEssays are not deleted.")
public void the_references_publicated_essays_are_not_deleted() {
    assertTrue(repository.getEssays().containsAll(publicatedEssays));
}
@When("an element Essay in the collection publicatedEssays is deleted")
```

```
public void an_element_essay_in_the_collection_publicated_essays
 _is_deleted() {
    essay.deleteEssay(repository);
}
@Then("the element Essay is removed from the reference collection
  publicatedEssays")
public void the_element_essay_is_removed_from_the_reference_
 collection_publicated_essays() {
    assertFalse(publicatedEssays.contains(essay));
    assertFalse(repository.getEssays().contains(essay));
}
```

- Run the the tests of the *essay_authors_reference.feature*. Some step definitions are already defined.
- Add the missing step definitions to the *EssaySteps.java* file.
- Implement the step definitions.

```
//src/test/java/essayexampleproject/EssaySteps.java
    @Given("an entity Essay with the references authors")
public void an_entity_essay_with_the_references_authors() {
    essay = new Essay(new EssayId(), essayTitle, essayBody);
    author = new Author(10, "Hanna");
    authorList.add(author);
    authorList.add(new Author(11, "Max"));
    essay.setAuthors(authorList);
    repository.addEssay(essay);
}

@When("the entity Essay is deleted")
public void the_entity_essay_is_deleted() {
    essay.deleteEssay(repository);
}

@Then("the references authors are not deleted.")
public void the_references_authors_are_not_deleted() {
    assertTrue(repository.getAuthors().containsAll(authorList));
}

@When("an element Author in the collection authors is deleted")
public void an_element_author_in_the_collection_authors_is_deleted() {
    repository.removeAuthor(author);
}
```

```
@Then("the element Author is removed from the reference collection
  authors")
public void the_element_author_is_removed_from_the_reference_
 collection_authors() {
    assertFalse(essay.getAuthors().contains(author));
}
```

- Run the tests and they pass.

G.2.8 1:n composition relationships

- Generate the 1:n composition relationship tests with the DDD-2-BDD tool.
- Import the feature file with the name *author_pendingEssays_reference.feature*.
- Run the the tests of the *author_pendingEssays_reference.feature*. Some step definitions are already defined.
- Add the missing step definitions to the *AuthorSteps.java* file.
- Implement the step definitions. In the Given statement create a new *author* and new *essay*. Add the *essay* to the *pendingEssayList* and then set the *pendingEssayList* in the *author*. Add the *essay* and the *author* to the repository. In the first Then statement check if the *pendingEssayList* is not contained in the repository. In the When statement delete the *essay*. In the second Then statement check if the deleted *essay* is also removed from the *pendingEssayList* from *author*.

```
////src/test/java/essayexampleproject/AuthorSteps.java
    @Given("an entity Author with the references pendingEssayList")
public void an_entity_author_with_the_references_pending_essay_list() {
    author=new Author(101,"author");
    essay = new Essay(new EssayId(), "title", "body");
    pendingEssayList.add(essay);
    author.setPendingEssayList(pendingEssayList);
    repository.addEssay(essay);
    repository.addAuthor(author);
}
@Then("the references pendingEssayList are deleted")
public void the_references_pending_essay_list_are_deleted() {
    assertFalse(repository.getEssays().containsAll(pendingEssayList));
}

@When("an element Essay in the collection pendingEssayList is deleted")
public void an_element_essay_in_the_collection_pending_essay_list_
 is_deleted() {
```

```
        essay.deleteEssay(repository);
    }

    @Then("the element Essay is removed from the reference collection
      pendingEssayList")
    public void the_element_essay_is_removed_from_the_reference_
     collection_pending_essay_list() {
        assertFalse(author.getPendingEssayList().contains(essay));
    }
```

- Run the tests and they pass.

G.2.9 Testing the Aggregate lifecycle

- Generate the test aggregate lifecycle with the DDD-2-BDD tool.
- Import the .feature file into your project with the name *aggregate_lifecycle_essay.feature*
- Create a cucumber parameter type[15] in the *EssaySteps.java* file.

```
//src/test/java/essayexampleproject/EssaySteps.java
@ParameterType("CREATED|SUBMITTED|OPENED|MARKED|FINALISED|RETURNED")// regexp
public State state(String state){ // type, name (from method)
    switch (state){
        case ("CREATED"):
            return State.CREATED;
        case ("SUBMITTED"):
            return State.SUBMITTED;
        case("OPENED"):
            return State.OPENED;
        case("MARKED"):
            return State.MARKED;
        case("FINALISED"):
            return State.FINALISED;
        case("RETURNED"):
            return  State.RETURNED;
        default:
            throw new IllegalStateException("Unexpected value: " + state);
    }
}
```

- Run the test and all will fail.

15 https://cucumber.io/docs/cucumber/cucumber-expressions/

- Import one given and one then statement. Replace the state (for example: CREATED) with {state} and add the State to the parameter list. You can remove the created String in the method name but it is not required.

```
//src/test/java/essayexampleproject/EssaySteps.java
//before
@Given("an aggregate Essay with the state CREATED")
public void an_aggregate_essay_with_the_state_created() {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}
@Then("the aggregate state change to CREATED")
public void the_aggregate_state_change_to_created() {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}
//after
@Given("an aggregate Essay with the state {state}")
public void an_aggregate_essay_with_the_state(State state) {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}
@Then("the aggregate state change to {state}")
public void the_aggregate_state_change_to(State state) {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}
```

- Run the test again and copy the rest of the step definitions.
- Now all steps should be pending.
- Add a default object for *author* and *corrector* to the instance variables of the *EssaySteps.java*. Then implement the step definitions. In the @Given("an aggregate Essay with the state {state}") statement an essay object is created and the input state is set. In the When statements execute the operations. The *author* and *corrector* are used in this example because the methods are declared in these classes. Do not forget to add a try and catch clause to the When statements. The Then statements checks with Junit if the state is like expected or the correct error was thrown. Below is a possible implementation shown.

```
//src/test/java/essayexampleproject/EssaySteps.java
@Given("an aggregate Essay with the state {state}")
public void an_aggregate_essay_with_the_state(State state) {
    essay=new Essay(new EssayId(), essayTitle, essayBody);
```

```
        essay.setState(state);
    }
    @When("the operation createEssay is executed")
    public void the_operation_create_essay_is_executed() {
        try{
            essay=author.createEssay(essayTitle, essayBody);
        }catch (Exception e){
            exceptionList.add(e);
        }
    }
    @When("the operation submitEssay is executed")
    public void the_operation_submit_essay_is_executed() {
        try{
            author.submitEssay(essay);
        }catch (Exception e){
            exceptionList.add(e);
        }
    }
    @When("the operation openEssay is executed")
    public void the_operation_open_essay_is_executed() {
        try{
            corrector.openEssay(essay);
        }catch (Exception e){
            exceptionList.add(e);
        }
    }
    @When("the operation markEssay is executed")
    public void the_operation_mark_essay_is_executed() {
        try{
            corrector.markEssay(essay, new Mark(5,5,5));
        }catch (Exception e){
            exceptionList.add(e);
        }
    }
    @When("the operation returnEssay is executed")
    public void the_operation_return_essay_is_executed() {
        try{
            essay=corrector.returnEssay(essay);
        }catch (Exception e){
            exceptionList.add(e);
        }
    }
    @When("the operation finaliseEssay is executed")
```

```
public void the_operation_finalise_essay_is_executed() {
    try{
        corrector.finaliseEssay(essay);
    }catch (Exception e){
        exceptionList.add(e);
    }
}
@Then("the aggregate state change to {state}")
public void the_aggregate_state_change_to(State state) {
    assertEquals(essay.getState(), state);
}
@Then("a illegal state exception is thrown")
public void a_illegal_state_exception_is_thrown() {
    assertEquals(IllegalStateException.class,
     exceptionList.get(0).getClass());
}
```

- The tests will fail.
- Add the functionality to the *Corrector.java* class. A possible implementation of the methods in the *Corrector.java* class is shown below.

```
//src/main/java/essayexampleproject/Corrector.java
public void markEssay(Essay essay, Mark mark) {
    if(!essay.getState().equals(State.OPENED)){
        throw new IllegalStateException();
    }
    essay.setState(State.MARKED); //write [OPENED -> MARKED];
}
public Essay returnEssay(Essay essay){
    if(!essay.getState().equals(State.MARKED)){
        throw new IllegalStateException();
    }
    essay.setState(State.RETURNED);
    return essay; //write[MARKED ->RETURNED];
}
public void finaliseEssay(Essay essay)  {
    if(!essay.getState().equals(State.MARKED)){
        throw new IllegalStateException();
    }
    essay.setState(State.FINALISED); //write[MARKED -> FINALISED]
}
```

- The tests will pass if you have everything implemented.

G.2.10 Entity identity checks

- Generate the Entity identity tests with the DDD-2-BDD tool.
- Import the feature files with the name *author_idenity.feature*, *corrector_identity.feature* and *essay_identity.feature*.
- Run the the tests of the *author_identity.feature*.
- Add the missing step definitions to the *AuthorSteps.java* file.
- Implement the step definitions. Add a second *author* to the instance variable *authorTwo*, a boolean *areEqual* and a variable, who save the *authorId*. In the first Given statement create an new *author* with random/default values. In the When statement save the authorId to the instance variable *authorId*. In the Then statment compare the *authorIds* with Junit. In the second Given statement create two authors with the same attributes and the given *authorIds* int1 and int2.
  Warning: the attributes of *author* and *authorTwo* must be identically in the `@Given("a Author with ID {int} and a Author with ID {int}")` statement.
  In the second When statement compare the two authors with equals and save it to the variable *areEqual*. In the second and third Then statement check if *areEqual* is true or false. In the third When statement compare all attributes of the *author* and *authorTwo* and save the result to *areEqual*. In the last Then statement test if *areEqual* is true.

```
//src/test/java/essayexampleproject/AuthorSteps.java
      @Given("an entity Author")
  public void an_entity_author() {
      author = new Author(101, "Anna");
  }
  @When("the operation getID is executed on the Author")
  public void the_operation_get_id_is_executed_on_the_author() {
      authorId = author.getAuthorId();
  }
  @Then("the identifier of the Author is returned.")
  public void the_identifier_of_the_author_is_returned() {
      assertEquals(authorId, author.getAuthorId());
  }
  @Given("a Author with ID {int} and a Author with ID {int}")
  public void a_author_with_id_and_a_author_with_id(Integer int1,
   Integer int2) {
      author = new Author(int1, "Anna");
      author.setClassName("Computer Science 2018");
      authorTwo = new Author(int2, "Anna");
      authorTwo.setClassName("Computer Science 2018");
  }
```

```
@When("two Author are tested for equality")
public void two_author_are_tested_for_equality() {
    areEqual = author.equals(authorTwo);
}
@Then("the two Author are equal")
public void the_two_author_are_equal() {
    assertTrue(areEqual);
}
@Then("the two Author are not equal")
public void the_two_author_are_not_equal() {
    assertFalse(areEqual);
}
@When("the attributes of two Author are compared")
public void the_attributes_of_two_author_are_compared() {
    areEqual = author.getAuthorId()==authorTwo.getAuthorId() &&
    author.getClassName().equals(authorTwo.getClassName()) &&
           author.getPendingEssayList().
     equals(authorTwo.getPendingEssayList()) &&
    author.getPendingEssayList().
     equals(authorTwo.getPublicatedEssays());
}
@Then("the attributes of the two Author are all equal")
public void the_attributes_of_the_two_author_are_all_equal() {
    assertTrue(areEqual);
}
```

- Some tests will fail. Implement the methods *equals()* and *hashCode()* to the *Author.java* class. An example implementation is shown below.

```
//src/main/java/essayexampleproject/Author.java
    @Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Author author = (Author) o;
    return authorId == author.authorId &&
            Objects.equals(name, author.name) &&
            Objects.equals(className, author.className) &&
            Objects.equals(publicatedEssays, author.publicatedEssays) &&
            Objects.equals(pendingEssayList, author.pendingEssayList) &&
            Objects.equals(allEssays, author.allEssays);
}

@Override
```

```
    public int hashCode() {
        return Objects.hash(authorId, name, className, publicatedEssays,
          pendingEssayList, allEssays);
    }
```

- The tests will pass.
- Run the tests of the *corrector_identity.feature*.
- Add the missing step definitions to the *CorrectorSteps.java* file.
- Implement the step definitions.

```
//src/test/java/essayexampleproject/CorrectorSteps.java
    @Given("an entity Corrector")
public void an_entity_corrector() {
    corrector=new Corrector(10, "Hans");
}
@When("the operation getID is executed on the Corrector")
public void the_operation_get_id_is_executed_on_the_corrector() {
    correctorId = corrector.getCorrectorId();
}
@Then("the identifier of the Corrector is returned.")
public void the_identifier_of_the_corrector_is_returned() {
    assertEquals(correctorId, corrector.getCorrectorId());
}
@Given("a Corrector with ID {int} and a Corrector with ID {int}")
public void a_corrector_with_id_and_a_corrector_with_id( Integer int1,
 Integer int2) {
    corrector = new Corrector(int1,"Fritz");
    correctorTwo = new Corrector(int2, "Fritz");
}
@When("two Corrector are tested for equality")
public void two_corrector_are_tested_for_equality() {
    areEqual = corrector.equals(correctorTwo);
}
@Then("the two Corrector are equal")
public void the_two_corrector_are_equal() {
    assertTrue(areEqual);
}
@Then("the two Corrector are not equal")
public void the_two_corrector_are_not_equal() {
    assertFalse(areEqual);
}
@When("the attributes of two Corrector are compared")
```

```
    public void the_attributes_of_two_corrector_are_compared() {
        areEqual = corrector.getCorrectorId()==
     (correctorTwo.getCorrectorId()) &&
        corrector.getName().equals(correctorTwo.getName());
    }
    @Then("the attributes of the two Corrector are all equal")
    public void the_attributes_of_the_two_corrector_are_all_equal() {
        assertTrue(areEqual);
    }
```

- Add the *equals()* and *hashCode()* methods to the *Corrector.java* class.
- Run the the tests of the *essay_identity.feature*.
- Add the missing step definitions to the *EssaySteps.java* file.
- Implement the step definitions. The *essayId* is generated in the constructor. Therefore it is not possible to set it. To test the Id compare the integer in the `@Given("a Essay with ID {int} and a Essay with ID {int}")` statement, if they are equal, copy the *essay* to variable *essayTwo*, if not, create a new essay with the same attributes.

```
//src/test/java/essayexampleproject/EssaySteps.java
    @Given("an entity Essay")
    public void an_entity_essay() {
        essay=new Essay(new EssayId(), "title", "body");
    }
    @When("the operation getID is executed on the Essay")
    public void the_operation_get_id_is_executed_on_the_essay() {
        essayId = essay.getId();
    }
    @Then("the identifier of the Essay is returned.")
    public void the_identifier_of_the_essay_is_returned() {
        assertEquals(essayId.getClass(), essay.getId().getClass());
    }
    @Given("a Essay with ID {int} and a Essay with ID {int}")
    public void a_essay_with_id_and_a_essay_with_id(Integer int1,
     Integer int2) {
        //possible workaround if the essay id is generated in the constructor
        essay= new Essay(new EssayId(), "title", "body");
        if(int1.equals(int2)){
            essayTwo = essay;
        }else{
            essayTwo= new Essay(new EssayId(), "title", "body");
        }
        essay.setMark(new Mark(4,5,5));
```

```
        essayTwo.setMark(new Mark(4,5,5));
        List<Author> authors = new LinkedList<>();
        authors.add(new Author(100, "Fritz"));
        essay.setAuthors(authors);
        essayTwo.setAuthors(authors);
    }
    @When("two Essay are tested for equality")
    public void two_essay_are_tested_for_equality() {
        areEqual = essay.equals(essayTwo);
    }
    @Then("the two Essay are equal")
    public void the_two_essay_are_equal() {
        assertTrue(areEqual);
    }
    @Then("the two Essay are not equal")
    public void the_two_essay_are_not_equal() {
        assertFalse(areEqual);
    }
    @When("the attributes of two Essay are compared")
    public void the_attributes_of_two_essay_are_compared() {
        // compare all attributes here
        areEqual = essay.getId().equals(essayTwo.getId()) &&
        essay.getTitle().equals(essayTwo.getTitle()) &&
        essay.getAuthors().equals(essayTwo.getAuthors()) &&
        essay.getEssayBody().equals(essayTwo.getEssayBody()) &&
        essay.getMark().equals(essayTwo.getMark());
    }
    @Then("the attributes of the two Essay are all equal")
    public void the_attributes_of_the_two_essay_are_all_equal() {
        assertTrue(areEqual);
    }
```

- Add the *equals()* and *hashCode()* methods to the *Essay.java* class.
- Now all tests of the files *author_idenity.feature*, *corrector_identity.feature* and *essay_identity.feature* will pass.

Further information about the test generation can be found in the test generation documentation file G.3.

G.3 How are the tests generated?

In the next sections we explain you how we have generated the tests. Each section is split in the following subsections:

- Recognition in the Context Mapper (CML) model
- What is tested
- Generated test cases
- Limitations
- Test coverage: low(only sunny cases), medium(some edge cases included), high(all edge cases included)

What are mutable attributes:

- attributes in an `Entity` with no keyword or the keyword `required`
- attributes in a `ValueObject` that is declared as `!immutable` and the attributes have no further keyword or `required`
- mutable attributes are not final and usually have a setter method

G.3.1 Testing the constructor and setter methods for Entities and ValueObjects

Recognition in the CML model: attributes and references of an `Entity` or a `ValueObject`

```
Entity Essay {
    - @EssayId identifier !changeable
    String title required
    String essayBody required
    - @Mark marks
    - Collection<@Author> authors size="max=5" cascade="persits"

}
ValueObject EssayId {
    int essayId
}
ValueObject Mark {
    !immutable
    int reasoningScore
    int relevanceScore !changeable
```

```
    int languageScore
}
```

What is tested: if the visibility of the `Entity` and `ValueObject` are implemented correctly

Generated test cases:

- constructor test with default values (in an `Entity` with the attributes with the key words `required` or `!changeable`, in a `ValueObject` all attributes if the `ValueObject` is `immutable` else the attributes with the key words `required` or `!changeable` attributes)
- setter test with default values for the mutable attributes
- setter test with `null` for the the mutable attributes

Limitations: The tests are generated with default values. The user has to set the values of the attributes. No test cases are generated for the edge cases, except the `null` test.

Test coverage: low

G.3.2 Range check for attributes

Recognition in the CML model: attribute with the keyword `range`

```
int points range="1" //only min
int points range="1,10" //min and max
int points range="1,10,'must be between 1 - 10'" //min, max and exception
int points range="min=1" //min
int points range="min=1,max=10" //min and max
int points range="min=1,max=10,message='must be between 1 - 10'"
 //min, max and exception
int points range="max=10" //only max
```

What is tested: testing whether the `ranges` are correctly implemented

Generated test cases:

- data inside the `range` (randomly generated)
- data outside the `range` throw an illegal argument exception or a personal exception if the message is set in the range (randomly generated for maximum and minimum)
- boundary data inside the `range` for maximum and minimum, with given range data
- boundary data outside the `range` for maximum and minimum throw an illegal argument exception or a personal exception if the message is set in the `range`, with given `range` data add one to the maximum or subtract one to the minimum

Limitations: In the current version are only tests generated for Integers. If the values are immutable, the object must be initialized in the steps definitions and the value for the range test is set with the initialization. The user must be sure to set the other attributes correctly, so that the range could be tested individual.

Test coverage: high

G.3.3 Size tests for collection

Recognition in the CML model: collection reference with the keyword `size`

```
- List<@Student> students size="1" //max
- List<@Student> students size="max=24" //max
```

What is tested: testing if the size of a collection is correctly implemented

Generated test cases:

- remove an object from a collection with `size` 1
- remove an object from a collection with `size` 0 throws an illegal argument error
- add an object to a collection with the `size` maximum minus one
- add an object to a collection with the `size` maximum throw an illegal argument error
- if the `size` is greater than one:
  - add an object to a collection with a randomly generated `size` between zero and maximum
  - remove an object from a collection with a randomly generated `size` between zero and maximum

Limitations: Only the maximum is supported in the current version. The default value for minimum is 0. We expect that the Collections are always mutable.

Test coverage: high

G.3.4 1:1 relationships

Recognition in the CML model: Reference in an `Entity` to a `ValueObject`

```
Entity Student{
    - @StudentId studentId
}
ValueObject StudentId{
    String id key
}
```

What is tested: testing whether the delete and change method of the object reference is correctly implemented

Generated test cases:

- testing whether the reference object is deleted when the main object is deleted
- if the reference is mutable
  - testing whether the old reference object is deleted when the reference is changed

Limitations: Take a look at the tutorials to find out, how you can implement the step definitions. These tests are close to the database and can be hard to implement with Cucumber.

Test coverage: medium

G.3.5 1:n aggregation relationships

Recognition in the CML model: Reference in an `Entity` with the keywords `cascade="persists"`

```
Entity Student{
    - Collection<@Essay> publicatedEssays cascade="persists"
    - List<@Essay> publicatedEssays cascade="persists"
    - Set<@Essay> publicatedEssays cascade="persists"
    - Bag<@Essay> publicatedEssays cascade="persists"
}
```

What is tested: testing whether the delete object and the object reference are correctly implemented

Generated test cases:

- testing whether the main object is not deleted when the reference object is deleted

- testing whether the reference object is removed from the collection when the reference object is deleted

Limitations: Take a look at the tutorials to find out, how you can implement the step definitions. These tests are close to the database and hard to implement with Cucumber.

Test coverage: medium

G.3.6 1:n composition relationships

Recognition in the CML model: Reference in an `Entity` with the keywords `cascade="persists,remove"`

```
Entity Student{
    - Collection<@Essay> publicatedEssays cascade="persists,remove"
    - List<@Essay> publicatedEssays cascade="persists,remove"
    - Set<@Essay> publicatedEssays cascade="persists,remove"
    - Bag<@Essay> publicatedEssays cascade="persists,remove"
}
```

What is tested: testing whether the delete object method and the object reference are correctly implemented

Generated test cases:

- testing whether the reference objects are deleted when the main object is deleted
- testing whether the reference object is removed from the collection when the reference object is deleted

Limitations: Take a look at the tutorials to find out, how you can implement the step definitions. These tests are close to the database and hard to implement with Cucumber.

Test coverage: medium

G.3.7 Testing the Aggregate lifecycle

Recognition in the CML model: an `Entity` who is the `aggregateRoot`, an `enum` with the keywords `aggregateLifecycle`, methods who are declared as `write` methods inside an `Entity` or `Service`

```
Entity Essay {
    aggregateRoot
    //..
}
enum States {
    aggregateLifecycle
    CREATED, SUBMITTED, OPENED, MARKED, FINALISED, RETURNED
}
Entity Author {
     def @EssayId createEssay(@Essay essay) : write [ -> CREATED];
     def boolean submitEssay(@Essay essay) : write [ CREATED, RETURNED-> SUBMITTED];
}
Entity Corrector {
     def @Essay openEssay(@EssayId essayId): write [SUBMITTED -> OPENED];
     def boolean markEssay(@EssayId essayId, @Mark mark): write [OPENED -> MARKED];
     def @Essay returnEssay(@EssayId essayId): write[MARKED ->RETURNED];
     def boolean finaliseEssay(@EssayId essayId): write[MARKED->FINALISED*];
}
```

What is tested: testing whether the delete object and the object reference are correctly implemented

Generated test cases:

- for the start method (for example : `write [ -> CREATED]`) one test whether the state is set correctly when the method is executed
- for the other methods:
  - check whether an illegal state exception is thrown, when the method is executed with the wrong state
  - check whether the state is correctly set after the method was executed

Limitations: Multiple outputs are not directly supported. (If you have multiple output states the Then statements has two possible states. The two states are split with an "or" in the Then statement. Make sure to implement the preconditions correctly in the When statement and check both states in the Then statement.)

Test coverage: medium if multiple outputs are used else high

G.3.8 Entity Identity checks

Recognition in the CML model: keyword `Entity`

```
Entity Student{
    //...
}
```

What is tested: testing whether the identifier/identity and the `equals` method of the `Entity` are correctly implemented

Generated test cases:

- check whether the `Entity` object has an identity
- assert if equality check for two `Entities` is correctly implemented and returns true if their identifiers are equal.
- assert if equality check for two `Entities` is correctly implemented and returns false if their identifiers are different
- check whether two `Entities` with the same identifier have identical attributes (as they should have as they are the same object)

Limitations: These test are not usable for all possible implementation of the identifier/identity of an Entity. In the tutorial is explained how the Cucumber steps work with two different implementations.

Test coverage: high