Programmentwurf Dualis-Bot

Name:	Schewe, Konrad
Matrikelnummer, Kurs:	3863127, TINF19B2
Abgabedatum:	16. Mai 2022

Inhaltsverzeichnis

Abkürzungsverzeichnis

CSV	Comma Separated Values
DIP	Dependency Inversion Principle
DRY	Don't Repeat Yourself
ISP	Interface Segreggation Principle
JAR	Java Archive
LSP	Liskov Substitution Principle
OCP	Open Closed Principle
SRP	Single Responsibility Principle

Abbildungsverzeichnis

1 Einführung

1.1 Übersicht über die Applikation

Die Duale Hochschule Baden-Württemberg nutzt das Tool Moodle als Studierendenverwaltungssystem. Das Portal Dualis ist ein Teil des Moodle-Softwarepakets. Mit Hilfe des Dualis Web-Client können Studierende Noten und weitere Informationen zum Studium abrufen. Die Anmeldung im Portal erfolgt mithilfe eines berechtigten Benutzerkontos mit Benutzername und Passwort.

Studierende möchten möglichst schnell wissen, wenn eine Note von der zugehörigen Sekretariatsstelle auf Dualis eingetragen wurde. Dualis bietet von Haus aus keine Möglichkeit der Benachrichtigung. Da man nicht genau weiß, wie schnell ein Dozent eine Klausur korrigiert hat, kann man auch nicht genau abschätzen, wann die Note erscheint. Dies sorgt für ein lästiges, ständiges Anmelden im Portal und der Suche nach dem Modul.

Die Idee ist es, ein Programm zu entwickeln, das sich automatisiert und regelmäßig im Portal anmeldet und sich die Daten von Dualis holt. Anschließend wird geprüft, ob eine neue Klausur eingetragen wurde. Ist dies der Fall, soll der gesamte Kurs benachrichtigt werden, aber auch die angemeldete Person selbst. Dabei enthält die Nachricht an die angemeldete Person zusätzlich die Note der Klausur, sodass ein Blick in das Portal nicht nötig ist.

Dualis bietet keine Schnittstelle an, über die die erforderlichen Daten erfasst werden können. Stattdessen gibt es das selbstentwickelte Plugin Dualis-Webscraper, das den Login ins Portal und das Auslesen der HTML-Elemente und damit der benötigten Daten ermöglicht. Es speichert die rohen gelesenen Informationen als Comma Separated Values (CSV). Erstellt werden dabei Listen mit Semestern, Modulen, Versuchen und Klausuren.

Der Dualis-Bot ruft dieses Plugin auf, erfasst auf Basis dieser Datengrundlage die neu eingetragenen Klausuren und sendet gegebenenfalls Nachrichten an Chats in Telegram über die Telegram API.

1.2 Wie startet man die Applikation?

Voraussetzungen:

- Das Dualis-Webscraping Plugin (dualis-webscraper.jar) liegt im Wurzel-Verzeichnis des Dualis-Bot (Entwicklung) oder im selben Verzeichnis der Dualis-Bot Java Archive (JAR).
- Das Wurzel-Verzeichnis, in der die Anwendung liegt, enthält die Verzeichnisse „old-results“ und „new-results“. Darin werden die Ergebnisse des Webscrapers gespeichert.
- Optional: Maven ist installiert
- Optional: IDE (IntelliJ) ist installiert

Wie wird die Applikation gestartet:

- Über die IDE (IntelliJ):
  1. Öffne das Modul 1-adapters.
  2. Lokalisiere die Klasse com.schewe.dualisbot.entrypoint.Main.
  3. Starte die main-Methode.
- Über die Kommandozeile (JAR):
  1. Öffne die Kommandozeile im Wurzelverzeichnis des Projekt.
  2. Führe „java -jar dualis-bot.jar“ aus.
- Über die Kommandozeile (Maven):
  1. Öffne die Kommandozeile im Wurzelverzeichnis des Projekt.
  2. Führe „mvn exec:java“ aus.

1.3 Wie testet man die Applikation?

Voraussetzungen:

- Maven ist installiert
- Optional: IDE (IntelliJ) ist installiert

Wie wird die Applikation getestet:

- Über die IDE (IntelliJ): Die Tests sind im Test-Verzeichnis der Module 2-application zu finden. Hier werden alle Test-Klassen gefunden und können einzeln ausgeführt werden.
- Über die Kommandozeile (Maven):
    1. Öffne die Kommandozeile im Wurzelverzeichnis des Projekt.
    2. Führe „mvn test“ aus.

2 Clean Architecture

2.1 Was ist Clean Architecture?

Die Clean Architecture gibt die Struktur einer Anwendung vor. Sie besteht aus mehreren Schichten:

- Abstraction Code (Schicht 4): Enthält Code für grundlegende Konzepte und Datenstrukturen, Algorithmen, die nichts mit der Anwendungsdomäne zu tun haben und daher nur selten berührt werden.
- Domain Code (Schicht 3): Enthält vor allem domänenspezifische Entitäten, die in der Application Schicht verwendet werden. Die Entitäten werden damit unabhängig von den Use Cases der eigentlichen Anwendung betrachtet und sollten ebenfalls selten verändert werden. Zusätzlich sind auch domänenspezifische Logiken zu finden, die eigentlich immer gleich sind.
- Application Code (Schicht 2): Enthält die Use Cases der Anwendung. Hier sollten die meisten Regeln der Anwendung zu finden sein (if-Abzweigungen). Hier wird der Fluss der Eingabe zu Ausgabe mithilfe der Entitäten der Domain Schicht geregelt. Der Application Code wird immer dann verändert, wenn sich die Anforderung an die Software ändert.
- Adapters (Schicht 1): Diese Schicht ermöglicht die Kommunikation der Anwendung nach außen. Sie interagiert mit anderen Anwendungen oder Plugins, wandelt von dort eingehende Daten in interne Formate und bietet Schnittstellen dafür an. Die konvertierten Daten können nun an die Application Schicht gereicht werden. Das Ziel dabei ist die Entkopplung von innen und außen. Die Adapters müssen immer dann angepasst werden, wenn sich eines der äußeren Einflüsse (Plugins) verändert. Dies passiert verhältnismäßig oft.
- Plugins (Schicht 0): Die Plugins gehören nicht zur eigentlichen Anwendung. Plugins interagieren nur mit der Adapter Schicht. Plugins treten beispielsweise als Frameworks, Datentransportmittel (z.B. Datenbanken, Benutzeroberfläche) auf.

Innere Schichten sind langlebig und sollten möglichst selten angepasst werden. Je weiter außen sich eine Schicht in der Anwendung befindet, desto häufiger wird die Schicht angepasst. Eine jeweils innere Schicht weiß nichts von den äußeren. Nur die äußeren Schichten dürfen die inneren verwenden. Diese Abhängigkeitsregel wird Dependency Rule genannt. Die Abbildung 2.1 veranschaulicht den oben beschriebenen Prozess.

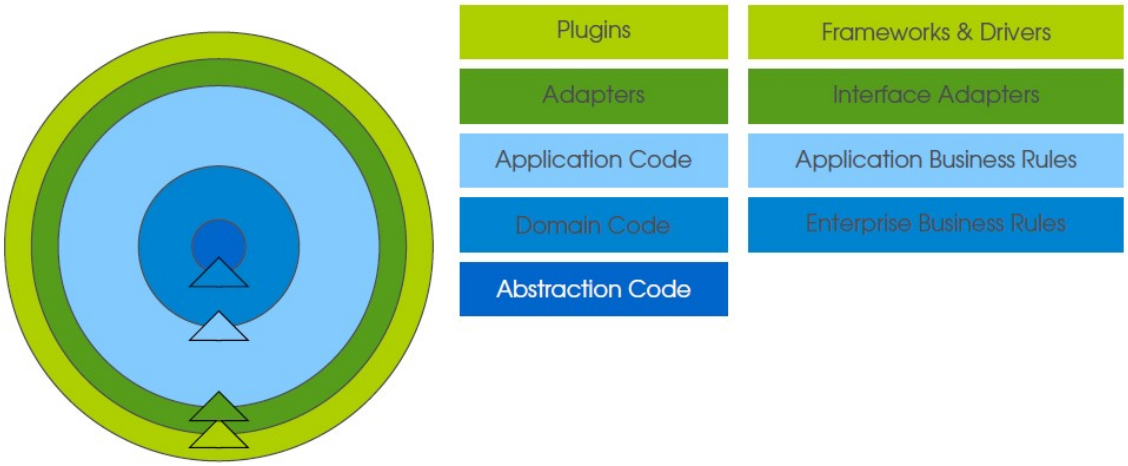

Abbildung 2.1: Clean Architecture mit Dependency Rule

2.2 Analyse der Dependency Rule

Im folgenden wird ein Positiv-Beispiel gezeigt, dass die Dependency-Rule einhält und ein Negativ-Beispiel, das sie verletzt.

2.2.1 Positiv-Beispiel

Die Abbildung 2.2 zeigt die Klasse MessageFactory, die in der Application Schicht liegt, da sie den Anwendungsfall einer Nachrichtengenerierung abdeckt. Ihre Aufgabe ist die Erstellung einer Message (Domain Code), die abhängig von der Exam (Domain Code) generiert werden. Die Message Factory wird in der Main Klasse (Adapters) verwendet. Damit bleibt die Abhängigkeit von Adapters zu Application Code zu Domain Code erhalten. MessageFactory hat keine Abhängigkeit in die obere Schicht, die Adapters.

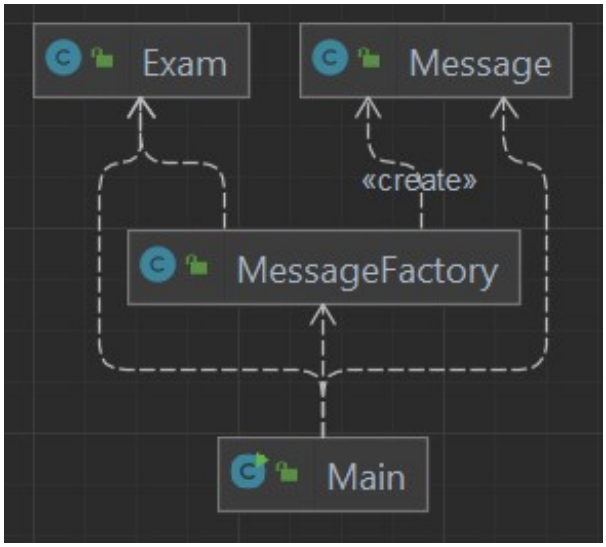

Abbildung 2.2: UML Klassendiagramm (MessageFactory): Positiv-Beispiel Dependency Rule

2.2.2 Negativ-Beispiel

Für das Negativ-Beispiel lässt sich leider kein UML zeigen. Das liegt daran, dass die hier erstellten UML-Diagramme von IntelliJ automatisch generiert wurden. Die Abhängigkeiten lassen sich schichtenweise nicht umkehren, da für jede Schicht ein extra Modul existiert und klare Abhängigkeiten zu anderen Modulen in Mavens POM definiert wurden. Listing 2.1 zeigt die Abhängigkeiten der POM aus dem Modul des Application Codes. Dabei ist nur der Domain Code gelistet, da dieser wiederum Zugriff auf Abstraction Code hat.

```
1 <dependencies>
2     <dependency>
3         <groupId>com.schewe.dualisbot</groupId>
4         <artifactId>3-domain</artifactId>
5         <version>1.0-SNAPSHOT</version>
6         <scope>compile</scope>
7     </dependency>
8 </dependencies>
```

Listing 2.1: Umsetzung Dependency Rule mithilfe der POM: Application Code

Gerne möchte ich aber beschreiben, wie ein Negativ-Beispiel aussehen könnte. In der positiven Abbildung 2.2 gehen alle Abhängigkeitspfeile nach oben. Nun könnte derselbe Prozess so implementiert werden, dass nicht die Main-Klasse die Generierung der Message aufruft, sondern dies über die Exam geschieht, die sowieso als Eingabe dient. Dabei würde die Dependency Rule verletzt werden, da Domain Code nicht auf Application Code zugreifen darf.

2.3 Analyse der Schichten

Im Folgenden werden zwei Klassen beschrieben, die aus zwei unterschiedlichen Schichten stammen. Das UML-Diagramm in Abbildung 2.3 enthält beide Klassen im Zusammenspiel.

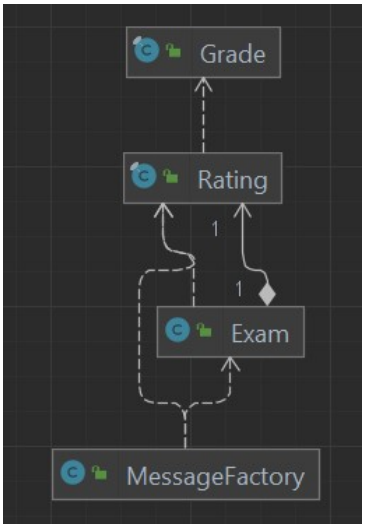

Abbildung 2.3: UML Klassendiagramm (Exam & Grade): Positiv-Beispiel Dependency Rule

2.3.1 Domain Code: Exam

Die Klasse Exam ist eine domänenspezifische Entität und gehört daher in die Domain Schicht. Sie muss nur selten angepasst werden, da sich die Eigenschaften einer Klausur so gut wie nie ändern. Laut Abbildung 2.3 wird Exam in der MessageFactory verwendet (Application Code). Eine Exam wird durch genau ein Rating bewertet. Rating ist kein primitiver Datentyp, da es nur bestimmte Werte enthalten darf. Dadurch eignet sich

die Implementierung als Value Object. Rating liegt weiterhin im Domain Code, besitzt allerdings Abhängigkeiten in den Abstraction Code. Ein Rating wird entweder durch eine Note oder durch eine Prozentzahl bestimmt. Beide liegen im Abstraction Code. Die Begründung dafür liefert die nächste Sektion.

2.3.2 Abstraction Code: Grade

Die Note Grade liegt im Abstraction Code. Sie ist ein Value Object, das Werte von 1-6 (float) annehmen kann. Grund dafür ist die eben genannte Einschränkung des Wertebereichs. Die Abhängigkeiten reichen hier nur in obere Schichten, genauer gesagt zum Rating. Im Gegensatz zu Rating liegt Grade aber im Abstraction Code. Das liegt daran, dass sich die Eigenschaften bzw. der Wertebereich einer Note niemals ändern wird. Dies gilt zumindest, solange das Notensystem in Deutschland weiterhin dem heutigen entspricht. Die Art einer Bewertung hingegen könnte sich an der Dualen Hochschule schneller ändern (z.B. Bewertung nur noch durch eine Prozentzahl).

3 SOLID

SOLID ist ein Akronym für die folgenden fünf Programmierprinzipien: Single-Responsibility-Prinzip, Open-Closed Prinzip, Liskov Substitution Prinzip, Interface Segregation Prinzip und Dependency Inversion Prinzip.

3.1 Analyse Single Responsibility Principle

Das Single Responsibility Principle (SRP) ist eine Entwurfsrichtlinie in der Softwarearchitektur. Im SRP soll jede Klasse maximal eine Verantwortung übernehmen bzw. einen Zweck erfüllen. Die Klasse hat somit nur eine Aufgabe, was dazu führt, dass sie nur einen Grund hat geändert zu werden.

3.1.1 Positiv-Beispiel

Im Dualis-Bot findet man viele dieser Klassen, die nur eine Aufgabe besitzen. Beispiel ist der ExamsComparator, dessen einzige Aufgabe ist, zwei Listen mit Klausuren (Exams) zu vergleichen. Listing 3.1 zeigt den Quellcode des ExamsComparator. Weitere Methoden, die ebenfalls die Eingabelisten vergleichen und eine andere Ausgabe als getDifference() liefern, wären hier als Erweiterung denkbar.

```
1 public class ExamsComparator {
2
3     private final List<Exam> oldExams;
4     private final List<Exam> newExams;
5
6     public ExamsComparator(List<Exam> oldExams, List<Exam> ↙
        ↳ newExams){...}
7
8     public List<Exam> getDifference(){...}
9
```

```
10 }
```

Listing 3.1: Source Code ExamsComparator: Positiv-Beispiel SRP

3.1.2 Negativ-Beispiel

Um das Positiv-Beispiel aufzugreifen, verändern wir den ExamsComparator und fügen die Methode addNewExams(List<Exam>) hinzu. Der entstandene Code ist in Listing 3.2 zu sehen. Die neue Methode verstößt gegen das SRP, da das Hinzufügen neuer Klausuren den Verantwortungsbereich des ExamsComparator überschreitet.

```
1 public class ExamsComparator {
2
3     public void addNewExams(List<Exams> newExams){...}
4
5 }
```

Listing 3.2: Source Code ExamsComparator: Negativ-Beispiel SRP

3.2 Analyse Open Closed Principle

Das Open Closed Principle (OCP) beschreibt das Konzept, dass Klassen, Module, Funktionen, etc., offen für Erweiterungen aber geschlossen für Modifikationen sein sollen. Die initiale Motivation für dieses Prinzip ist das Vermeiden von ständigen Veränderungen in einer Klasse, wenn die Bedingungen sich ändern.

3.2.1 Positiv-Beispiel

Als Positiv-Beispiel kann hier auch wieder der ExamsComparator aus Listing 3.1 herangezogen werden. Der ExamsComparator hat bisher nur die Methode getDifference(), der die Klausuren der oldExams aus der Liste der newExams subtrahiert und so die Differenz berechnet und zurückgibt. Diese Methode wird immer so bleiben und wird nie modifiziert werden müssen, da getDifference() auf die Exam.equals() Methode zurückgreift (falls eine Änderung am Indikator einer Exam geschieht). Erweiterungen hingegen sind im

ExamsComparator gerne gesehen und einfach zu implementieren. Beispielsweise könnte eine Methode getIntersection() hinzugefügt werden, die die anstelle der Differenz die sich überschneidenden Klausuren berechnet.

3.2.2 Negativ-Beispiel

Ein schlechtes Beispiel für OCP ist die Klasse EntityMatcher. Der EntityMatcher hat die Aufgabe einem Modul seine zugehörigen Versuche mit den passenden Klausuren zuzuordnen. Er fügt also einige Entities zu der Modul-Aggregation zusammen. Der Code ist in Listing 3.3 zu sehen. Würde das Aggregat Modul nun erweitert, müsste man die Methode im Detail anpassen. Erweiterbar wäre die Klasse dennoch um weitere Methoden, die andere Aggregate zusammenfügen.

```
1 public class EntityMatcher implements ModuleMatcher {
2
3     public static void matchModules(List<Module> modules, ↙
          ↳ List<Attempt> attempts, List<Exam> exams){...}
4
5 }
```

Listing 3.3: Source Code EntityMatcher: Negativ-Beispiel OCP

3.3 Analyse Interface Segreggation Principle

Das Interface Segregation Prinzip empiehlt zu große Schnittstellen in mehrere kleinere Schnittstellen aufzuteilen, falls die implementierende Klasse unnötige Methoden haben müsste.

3.3.1 Positiv-Beispiel

Ein Beispiel in dem das Interface Segreggation Principle (ISP) umgesetzt ist, ist der EntityMatcher. Dieser implementiert den ModuleMatcher, der vorgibt, die Methode matchModules() zu verwenden. Anstelle eines EntityMatcherInterfaces, das Matching-Methoden für alle Aggregates enthält und so überflüssige Methoden hat, werden die

Matching-Interfaces in kleinere Teile für jedes Aggregate zerlegt. Die Implementierung des Interfaces ist in Listing 3.3 zu betrachten.

3.3.2 Negativ-Beispiel

Um das Positiv-Beispiel zu einem Negativ-Beispiel umzukehren, muss lediglich das Interface ModuleMatcher in EntityMatcherInterface umbenannt werden (siehe Listing 3.4).

```
1 public class EntityMatcher implements EntityMatcherInterface ↙
      ↳ {...}
```

Listing 3.4: Source Code EntityMatcher: Negativ-Beispiel ISP

Das Interface EntityMatcherInterface enthält nun nicht nur die abstrakte Methode matchModules(), sondern beispielsweise auch matchInstructors(). Diese Erweiterung ist in Listing 3.5 dargestellt.

```
1 public interface EntityMatcherInterface {
2
3     static void matchModules() {}
4
5     static void matchInstructors() {}
6
7 }
```

Listing 3.5: Source Code EntityMatcherInterface: Negativ-Beispiel ISP

Gegebenenfalls passt das aber gar nicht in den Kontext des EntityMatchers und dieser muss deswegen irrelevante Methoden implementieren.

4 Weitere Prinzipien

4.1 GRASP

General Responsibility Assignment Software Patterns bezeichnet eine Menge von Entwurfsmustern, mit denen die Zuständigkeit bestimmter Klassen objektorientierter Systeme festgelegt wird. Diese Sektion setzt sich mit Low Coupling und High Cohesion im Dualis-Bot auseinander.

4.2 Analyse GRASP: Geringe Kopplung

Geringe Kopplung ist eines der Hauptziele von gutem Design. Kopplung bezeichnet hier das Maß für die Abhängigkeit eines Moduls von anderen Modulen. Eine gering Kopplung impliziert also eine geringe Abhängigkeit eines Moduls von seiner Umgebung. Grund für eine niedrige Kopplung ist die Eventualität einer Austauschbarkeit eines Moduls durch ein anderes.

Im Dualis-Bot kann die Applikation in zwei thematische Module getrennt werden:

- Dualis: Kümmert sich um die Verwertung der Ergebnisse des Dualis-Webscrapers.
- Telegram: Sorgt sich um das Erstellen von Nachrichten und die Kommunikation mit Telegram.

Wie gering die Kopplung dieser beiden Module tatsächlich ist, wird in den Positiv- und Negativ-Beispielen besprochen.

4.2.1 Positiv-Beispiel

Abbildung 4.1 zeigt ein UML Klassendiagramm von einem relevanten Ausschnitt der Anwendung. Der Wurzelknoten liegt in der Main-Klasse. Von hier gehen zwei Hauptstränge ab, die streng separiert bleiben. Der linke Strang gehört zum Modul Telegram, der rechte

zu Dualis. Da es dazwischen keine Berührungspunkte gibt, beweist dies, dass eine geringe Kopplung besteht.

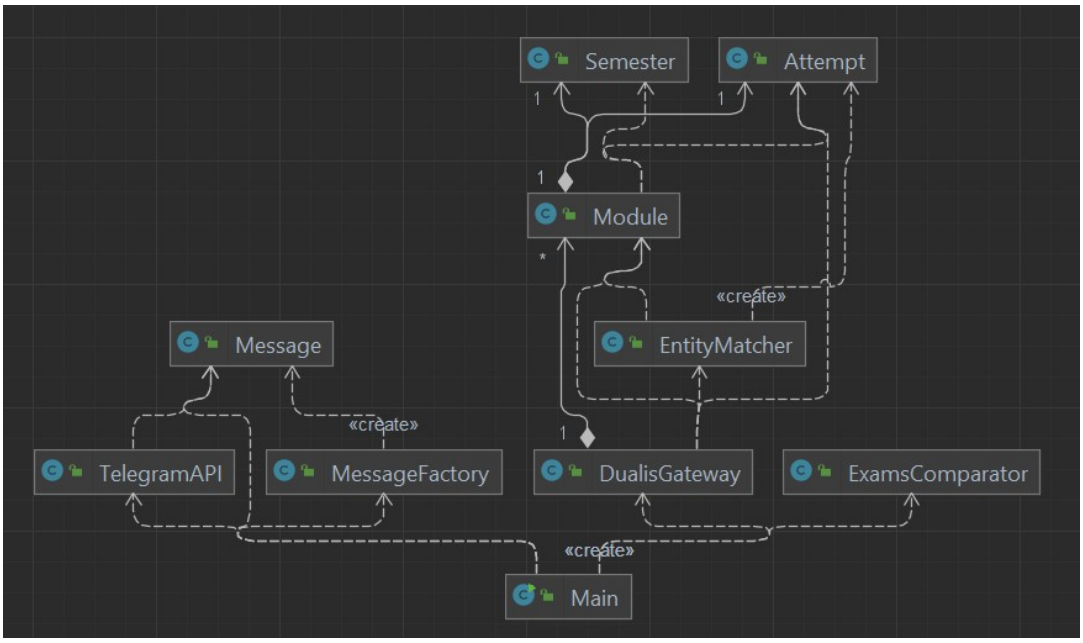

Abbildung 4.1: UML Klassendiagramm (Themenbereiche Dualis & Telegram): Positiv-Beispiel Geringe Kopplung

4.2.2 Negativ-Beispiel

Im Positiv-Beispiel aus Abbildung 4.1 sind nicht alle Klassen abgebildet. Tatsächlich teilen sich Modul Dualis und Telegram nicht nur die Main-Klasse, sondern auch die Exam. Das Modul benötigt diese, um darin die vom Dualis-Webscraper gelesen Daten zu speichern. Modul Telegram benötigt Informationen aus den erstellten Exams, um daraus Nachrichten zu generieren. Über diese Klasse entsteht eine Kopplung, die gelöst werden sollte. Die Main-Klasse könnte beispielsweise ein Transfer-Objekt (Adapters) erstellen, das nicht zu einem der Domänenentitäten gehört und von beiden Modulen verwendet werden kann. Dadurch entsteht kein so großes Netz, wie in Abbildung 4.2, sondern eine geordnete Modulstruktur.

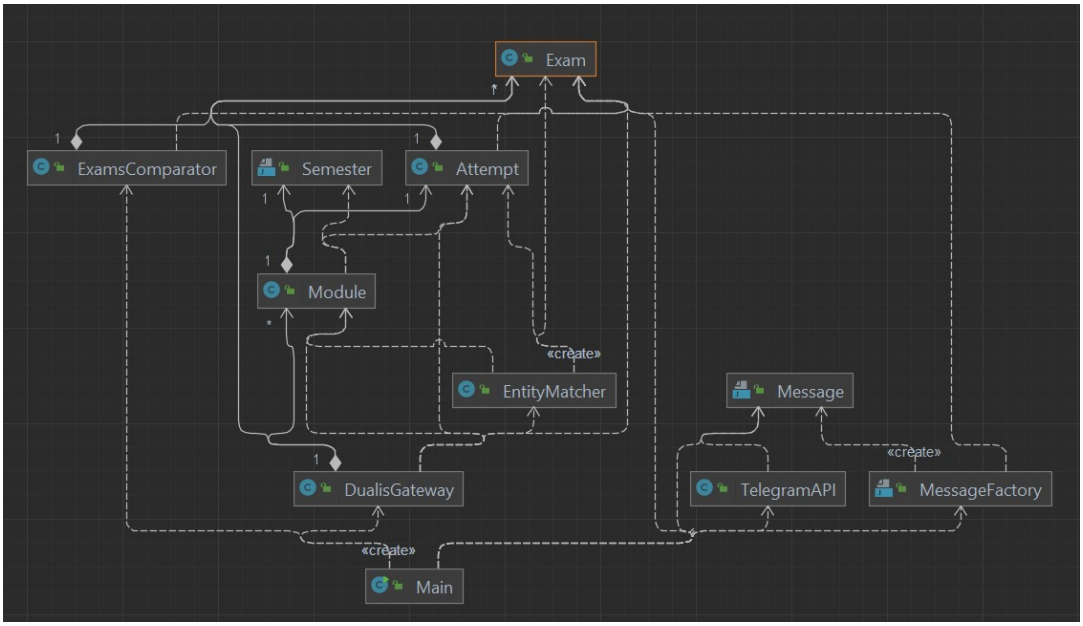

Abbildung 4.2: UML Klassendiagramm (Themenbereiche Dualis & Telegram): Negativ-Beispiel Geringe Kopplung

4.3 Analyse GRASP: Hohe Kohäsion

Hohe Kohäsion ist wichtig, um Komplexität von Gesamtsystemen zu begrenzen, indem man Klassen gut überschaubar organisiert. Eine Kohäsion ist dann präsent, wenn die Klassen innerhalb eines Moduls eine starke Bindung haben. Abbildung 4.3 zeigt das vollständige UML des Moduls Dualis. Durch die vielen Abhängigkeiten wird klar, dass hier eine starke Vernetzung und damit eine hohe Kohäsion herrscht. Stärker vernetzte Klassen liegen im Zentrum, weniger vernetzte Klassen werden an den Rand des Diagramms gedrängt. Die Clean Architecture wird dabei immer streng eingehalten.

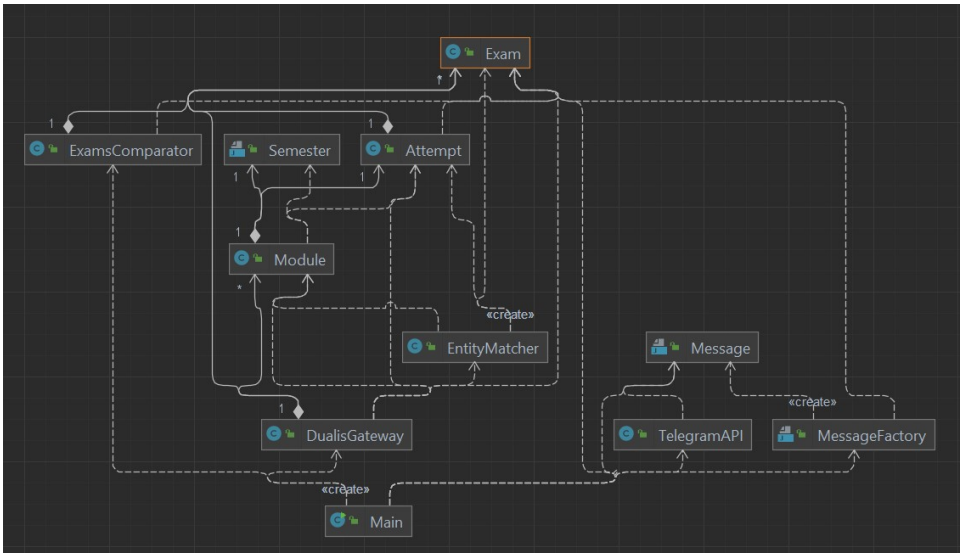

Abbildung 4.3: UML Klassendiagramm (Themenbereich Dualis): Beispiel Hohe Kohäsion

Erst durch das Prinzip von High Cohesion ist Low Coupling möglich. Es wird im Code sofort klar, welche Komponente welche Aufgabe erfüllen soll.

4.4 Don't Repeat Yourself

Don't Repeat Yourself (DRY) ist ein Prinzip in der Softwareentwicklung, die darauf abzielt Wiederholungen im Code zu vermeiden, indem man es mit Abstraktionen oder Normalisierungen ersetzt. Eines der Beispiele, wie Codeduplikation vermieden wird, ist die Auslagerung von frequentiert genutzten Methoden, die im selben Kontext stehen.

Eine solche Duplikation tritt in der ModulesCSV auf. Der Code-Ausschnitt in Listing 7.1 konvertiert den String value in ein float und gibt das anschließend in das Value Object Grade. Dieser Prozess geschieht sowohl für die ModulesCSV, als auch für die AttemptsCSV.

Dieser duplizierte Code kann ausgelagert werden. Für derartige Konvertierungen werden von nun an Converter erstellt, die die Methode convert() implementieren müssen. Darin ist dann der duplizierte Code enthalten. Als Beispiel ist in Listing 7.2 der neu erstellte GradeConverter zu sehen.

Für die Verwendung des ausgelagerten Codes ist nun lediglich eine Zeile nötig. Listing 7.3 verwendet den GradeConverter, um den value String in die Note umzuwandeln.

5 Unit Tests

Testen ist ein wichtiger Teil des Softwareentwicklungsprozesses. Durch ständige Tests können Fehler bereits vor Release entdeckt werden und zudem zuverlässiger erkannt werden. Beim Unit Testing werden individuelle Units oder Komponenten getestet. Der Zweck von Unit Tests ist das Verhalten von den einzelnen Units auf richtiges Verhalten zu validieren.

5.1 10 Unit Tests

MessageFactoryTest.generateMessagesForFirstAttempt()
Testet, ob für eine Erstklausur die korrekte Telegram-Nachricht generiert wird.

MessageFactoryTest.generateMessagesForSecondAttempt()
Testet, ob für Zweit- / Dritt- / ... -versuche die korrekte Telegram-Nachricht generiert wird.

ExamsComparatorTest.getDifferenceExisting()
Testet, ob der ExamsComparator eine vorhandene Klausurendifferenz tatsächlich bestimmen kann.

ExamsComparatorTest.getDifferenceNotExisting()
Testet, ob der der ExamsComparator bei der Differenz zweier identischer Klausurenlisten eine leere Liste / Differenz zurückgibt.

EntityMatcherTest.matchModulesAttemptIncluded()
Testet, ob ein Attempt mit einer zu einem Modul passenden ModulId tatsächlich zu dem Modul-Aggregat hinzugefügt wird.

EntityMatcherTest.matchModulesAttemptNotIncluded()
Testet, ob ein Attempt mit einer zu einem Modul unpassenden ModulId nicht zu dem

Modul-Aggregat hinzugefügt wird.

EntityMatcherTest.matchModulesExamIncluded()
Testet, ob eine Exam mit einer zu einem Attempt passenden Attempt-Nummer tatsächlich zu dem Modul-Aggregat mit dem Attempt hinzugefügt wird.

EntityMatcherTest.matchModulesExamNotIncluded()
Testet, ob eine Exam mit einer zu einem Attempt unpassenden Attempt-Nummer nicht zu dem Modul-Aggregat mit dem Attempt hinzugefügt wird.

5.2 ATRIP: Automatic

Tests sollten automatisch gestartet werden. Ergebnisse von Tests sollen automatisch auf Erfolg oder Misserfolg geprüft werden. Dieser Punkt ist im Dualis-Bot erfüllt, da unter Nutzung von JUnit, Tests automatisch ausgeführt werden können und die Testergebnisse in einem Testdurchlauf am Ende für jeden Test angezeigt werden. Das maven-surefire-plugin aus Listing 5.1 sorgt dafür, dass alle Tests im Projekt gleichzeitig mit dem Befehl „mvn test“ ausgeführt werden sollen. Dann erhält man eine Übersicht, der erfolgreichen und fehlgeschlagenen Tests.

```
1 <plugin>
2     <groupId>org.apache.maven.plugins</groupId>
3     <artifactId>maven-surefire-plugin</artifactId>
4     <version>2.22.0</version>
5 </plugin>
```

Listing 5.1: Plugin Maven-Surefire: ATRIP Automatic

Ein detailreiches Testergebnis mit Code Coverage wird durch Jacoco (siehe Listing 5.2) möglich.

```
1 <plugin>
2     <groupId>org.jacoco</groupId>
3     <artifactId>jacoco-maven-plugin</artifactId>
4     <version>0.7.2.201409121644</version>
5     <executions>
```

```
6           <execution>
7               <goals>
8                   <goal>prepare-agent</goal>
9               </goals>
10          </execution>
11          <execution>
12              <id>generate-code-coverage-report</id>
13              <phase>test</phase>
14              <goals>
15                  <goal>report</goal>
16              </goals>
17          </execution>
18      </executions>
19  </plugin>
```

Listing 5.2: Plugin Jacoco: ATRIP Automatic

5.3 ATRIP: Thorough

Dieses Prinzip von ATRIP konzentriert sich auf die Coverage. Bugs tendieren oftmals dazu sich in einer Region zu sammeln, dennoch kann es dazu kommen, dass Bugs sehr verteilt sind und unentdeckt bleiben. Um dies zu vermeiden, sollte man möglichst alle Schlüsselpfade und Szenarios gründlich testen, damit keine Bugs unter dem Radar verloren gehen.

Die Methode MessageFactory.generateMessagesFor() enthält in ihrer Implementierung genau eine Abzweigung, die abhängig von der Versuchsnummer (Attempt-Nummer > 1), der Telegram-Nachricht einen Hinweis hinzufügt. Die genannte Abzweigung ist in Code-Beispiel 5.3 zu betrachten.

```
1  public class MessageFactory{
2
3      public static List<Message> generateMessagesFor(Exam exam){
4          ...
5          if(exam.getAttemptNumber().getNumber() > 1){
```

```
6                 textGroup += "\nHinweis: Es handelt sich um eine ↙
                      ↳ Nachklausur.";
7             }
8             ...
9         }
10
11    }
```

Listing 5.3: MessageFactory: Hinweis bei Nachklausuren

Das Hinzufügen dieses Hinweises muss in den Tests geprüft werden. Listing 5.4 zeigt das korrekte Testsetup mit 2 Tests (ein Test für jede Abzweigung). Einmal wird das Hinzufügen für Zweitversuche getestet, einmal das Weglassen des Hinweises.

```
1  public class MessageFactoryTest{
2
3      public void generateMessagesForFirstAttempt(){...}
4
5      public void generateMessagesForSecondAttempt(){...}
6
7  }
```

Listing 5.4: MessageFactoryTest: Positiv-Beispiel ATRIP Thorough

Listing 5.5 zeigt wie es nicht gemacht werden sollte. MessageFactoryTest enthält nur einen Test und prüft die Abzweigung nicht.

```
1  public class MessageFactoryTest{
2
3      public void generateMessagesFor(){...}
4
5  }
```

Listing 5.5: MessageFactoryTest: Negativ-Beispiel ATRIP Thorough

5.4 ATRIP: Professional

Professionalität für Tests beinhaltet die Menge an Tests, die im Bestfall oftmals die Menge an Produktionscode überragt, und Tests, die angemessen benannt wurden. Zurzeit lässt die Menge an Tests zu wünschen übrig, jedoch ist ein Teil der Professionalität durch die Namensgebung bereits gegeben. So beschreibt bereits der Methodenname grob, was der Testzweck der Methode ist (z.B generateMessagesForFirstAttempt()). Indem man die Professionalität der Tests hoch hält, stellt man sicher, dass Tests sowohl zuverlässig laufen als auch einfach zu warten und verstehen sind.

Bei einem Blick auf 5.4 finden sich zwei Tests generateMessagesForFirstAttempt() und generateMessagesForSecondAttempt(). Die Namensgebung für generateMessagesForFirstAttempt() ist professionell; die für generateMessagesForSecondAttempt() nicht ganz. Hier steht im Name der Zweitversuch im Vordergrund. Getestet wird zwar der Zweitversuch, die Verantwortlichkeit deckt aber alle Nachklausuren (Versuche >= 2) ab. Daher wäre ein Methodenname, wie generateMessagesForPostAttempt() gegebenenfalls aussagekräftiger.

5.5 Code Coverage

Die Code Coverage im Dualis Bot wird mithilfe des Plugins Jacoco berechnet. Die Einbindung des Plugins findet sich in Listing 5.2. Laut Jacoco wurde eine Code Coverage von 45%. Eigentlich dürfte die Coverage nicht so hoch sein. Grund dafür ist, dass Jacoco nur die Test-Abdeckung von Methoden miteinbezieht. Allerdings findet sich auch viel Logik in den Konstruktoren der Klassen. Eine so hohe Coverage ist außerdem möglich, weil viele Abzweigungen getestet werden. In der Coverage fehlen hauptsächlich die equals()-Implementierungen auf den Entitäten im Domain Code.

5.6 Fakes und Mocks

Mocking ist eine Methode um das Verhalten von echten Objekten oder Methoden zu simulieren. Mocks werden vor allem in Unit Tests genutzt.

Im Dualis-Bot wird ein Mock beispielweise im ExamsComparatorTest verwendet. Vor jedem Test werden die Mock-Objekte in beforeAll() zurückgesetzt und neu generiert. Erstellt werden Listen, die gegebenenfalls bereits Klausuren enthalten. Die Klausuren, mit denen vergleichen werden soll, werden erst in den Tests selbst erzeugt. Die Klausurenlisten sind in diesem Fall die Mock-Objekte, die eigentlich in echt gar nicht existieren. Der beschriebene Prozess ist in Code-Schnipsel 5.6 zu sehen.

```
1  public class ExamsComparatorTest{
2
3      List<Exam> oldExams;
4      Exam oldExam;
5      List<Exam> newExams;
6
7      @BeforeEach
8      public void beforeAll() throws Exception {
9          oldExams = new ArrayList<>();
10         oldExam = new Exam("Betriebssysteme", new ↙
               ↳ Percentage(0.5f), new Rating(new ↙
               ↳ Percentage(0.83f)), new ↙
               ↳ NaturalNumber(1),"T3INF2005");
11         oldExams.add(oldExam);
12         newExams = new ArrayList<>();
13         newExams.add(oldExam);
14     }
15
16     public void getDifferenceExisting(){...}
17
18     public void getDifferenceNotExisting(){...}
19
20 }
```

Listing 5.6: MessageFactoryTest: Negativ-Beispiel ATRIP Thorough

6 Domain Driven Design

„Domain-Driven Design is an approach to software development that centers the development on programming a domain model that has a rich understanding of the processes and rules of a domain.“ (Martin Fowler: Domain Driven Design). Martin Fowler beschreibt Domain Driven Design als einen Softwareentwicklungsansatz, der im Entwicklungsprozess ein Domain Modell im Fokus hat, welches das einfache Verstehen von Regeln und Prozessen ermöglicht. Die folgende Sektion wird sich ausgiebig mit dem Domain Driven Design auseinandersetzen und erläutert inwiefern der Dualis-Bot das Domain Driven Design implementiert.

6.1 Ubiquitous Language

Die Ubiquitous Language ist eine Sprache, die das Ziel verfolgt eine allgemeingültige Sprache anzubieten, die es ermöglicht, dass jede involvierte Partei, z.B. Entwickler, Domänenexperten, Designer, etc., die entwickelte Software einfach zu verstehen. Um die Ubiquitous Language zu entwickeln, muss die Businessdomäne vollständig verstanden werden, sonst kann eine unklare Ubiquitous Language. Die folgenden Paragraphen beschreiben die Ubiquitous Language und ihre Begriffe genauer.

Semester
Der Begriff „Semester“ bezeichnet ein Semester für den Studiengang eines Studienjahres. Es findet eine Unterscheidung zwischen Winter- und Sommersemester statt.

Module
Der Begriff „Module“ bezeichnet ein Modul im Studienplan eines Studienganges. Ein Modul kann mit einer Note bestanden werden, falls alle enthaltenen Klausuren (Exam) in einem der Versuche (Attempt) ebenfalls bestanden wurden. Für ein Modul erhält man Credits.

Attempt

Der Begriff „Attempt“ bezeichnet den Versuch ein Modul zu bestehen. Ein Versuch wird dabei mit einer Nummer gekennzeichnet. Für jeden Versuch müssen alle im Modul enthaltenen Klausuren bestanden werden.

Exam
Der Begriff „Exam“ bezeichnet eine Klausur eines Moduls. Dabei erhält man für eine Klausur eine Bewertung, die entweder eine Prozentzahl oder eine Note ist.

Message
Der Begriff „Message“ bezeichnet eine Nachricht, die an die Telegram API geschickt werden kann. Sie enthält einen Text und einen Chatraum, an den die Nachricht versendet werden soll.

Die genannten Begriffe gehören zur Ubiquitous Language, da sie Business-Entitäten sind, die essentiell zur Verarbeitung der Informationen des Dualis-Webscrapers oder das Senden an die Telegram API sind.

6.2 Entities

Die eben genannten Begriffe der Ubiquitous Language sind ebenfalls die Entities des Dualis-Bots. Semester, Module, Attempt und Exam gehören der Domäne Dualis an; Message der Domäne Telegram. Abbildung 6.1 veranschaulicht das Zusammenspiel der Entitäten. Die Aufteilung in die Domänen wird sichtbar. Ein Modul hat gehört genau zu einem Semester. Ein Modul kann mehrere Versuche haben. Zu einem Versuch gehören mehrere Klausuren eines Moduls.

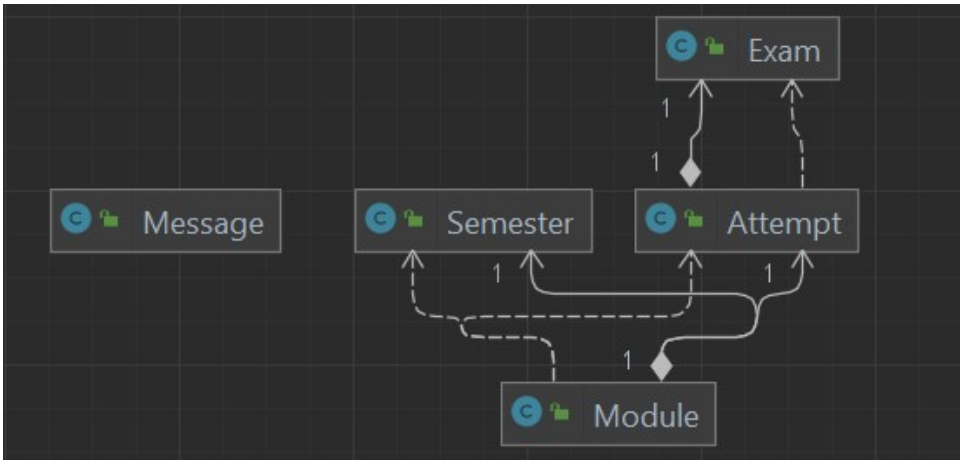

Abbildung 6.1: UML Klassendiagramm Entities

6.3 Value Objects

Value Objects bilden ein grundlegendes Attribut einer Entität ab. Value Objects werden benutzt, um beispielweise den Werebereich dieses Attributes einzugrenzen, sodass keine belibigen Werte möglich sind. Das sorgt dafür, dass Entwickler dabei keine Fehler machen können. Der umschlossene Wert Value Objects ist unveränderlich (final). Der Wert wird über den Konstruktor in das Objekt gegeben und kann nur über eine neue Objekterzeugung manipuliert werden.

Im Dualis-Bot wird eine Klausur mit einem Rating bewertet. Ein Rating besteht entweder aus einer Note oder aus einer Prozentzahl der erreichten Punkte. Das zugehörige UML ist in Abbildung 6.2 zu sehen. Dabei sind Grade und Percentage beide Value Objects, da sie einen festen Wertebereich haben.

Abbildung 6.2: UML Klassendiagramm Rating Value Objects

Im Folgenden wird genauer auf die Implementierung der Grade eingegangen. Der Code wird in Listing 6.1 gezeigt. Der Wertebereich einer Note wird im Konstruktor auf 1 bis 6 eingeschränkt. Der Wert ist nur im Konstruktor veränderbar. Manipulative Methoden würden ein neues Grade-Objekt erzeugen und zurückgeben.

```
1  public final class Grade {
2
3      private final float grade;
4
5      public Grade(final float grade) throws Exception {
6          if(grade >= 1 && grade <= 6)
7              this.grade = grade;
8          else
9              throw new Exception("A value for a grade has to be ↙
                   ↳ between 1.0 and 6.0.");
10     }
11
12     public float getGrade() {...}
13
14     ...
15
16 }
```

Listing 6.1: Grade: Beispiel Value-Object

6.4 Repositories

Repositories sind für das Management der Entitätsobjekte zuständig. Im Dualis-Bot werden die Entitätsobjekte komplett vom Dualis-Webscraper bereitgestellt. Eine Manipulation (Create, Update, Delete) ist nicht nötig. Lediglich die Abfrage (Read) muss bereitgestellt werden. Für die Abfrage gibt es keine speziellen Anforderungen. Die vom Webscraper gelesen Informationen müssen auch nicht gefiltert werden.

Daher wäre eine Implementierung eines Repositories überflüssig. Bei neuen Anforderungen, wie einer Maipulation oder Filterung der Daten, könnte ein Module- oder Exam-Repository erstellt werden. Dementsprechend werden die Daten direkt von der Schnittstelle DualisGateway mit den entsprechenden CSVReadern bereitgestellt.

6.5 Aggregates

Aggregates sind ein Zusammenschluss von Entitäten. In diesem Zusammenschluss ist genau geregelt, welche Entität auf welche andere verweisen darf, um so die Abhängigkeiten abzubilden. Dadurch soll ein zu komplexes Abhängigkeitsnetz verhindert werden. Zudem regelt ein Aggregate, über welche Entität auf das Aggregate zugegriffen werden darf und über welche nicht. Zusammenfassend reduzieren Aggregates also die Abhängigkeitskomplexität.

Im Dualis-Bot gibt es die folgenden Aggregate:

- Module-Aggregate: Das Modul enthält Verweise auf das zugehörige Semester. Es enthält passende Versuche, die wiederum Klausuren enthalten. Auf die Entitäten des Modul-Aggregats darf nur über das Modul selbst zugegriffen werden.
- Semester-Aggregate: Ein Semester-Aggregate enthält nur ein Semester. Über das Semester-Aggregate kann nicht auf zugehörige Module zugegriffen werden.
- Exam-Aggregate: Ein Exam-Aggregate enthält nur eine Klausur. Über das Exam-Aggregate kann nicht auf zugehörige Versuche oder Klausuren zugegriffen werden.
- Message-Aggregate: Das Message-Aggregate ist das einzige der Telegram-Domäne und enthält alle Entities der Domäne.

7 Refactoring

7.1 Code Smells

[jeweils 1 Code-Beispiel zu 2 Code Smells aus der Vorlesung; jeweils Code-Beispiel und einen möglichen Lösungsweg bzw. den genommen Lösungsweg beschreiben (inkl. (Pseudo-)Code)]

Code Smells sind Charakteristika die im Programmcode eines Programms auf tiefere Probleme hinweisen. Smells sind gewisse Strukturen im Code, die Verstöße gegen fundamentale Designprinzipien aufzeigen. Diese Sektion wird einige Code Smells im Code im Dualis-Bot aufzeigen und zeigen, wie diese Code Smells behoben wurden.

7.1.1 Code Smell: Duplicated Code

Duplizierter Code bezeichnet identischen oder sehr ähnlichen Code, der mehrfach an verschiedenen Stellen genutzt wird. Im Dualis-Bot gibt es mehrfach verwendeten Code in den Adapters, die die CSV Dateien des Dualis-Webscrapers in Entitäten konvertieren.

Der Code-Ausschnitt in Listing 7.1 konvertiert den String value in ein float und gibt das anschließend in das Value Object Grade. Dieser Prozess geschieht sowohl für die ModulesCSV, als auch für die AttemptsCSV.

```
1 ...
2 float floatGrade = Float.parseFloat(value);
3 grade = new Grade(floatGrade);
4 ...
```

Listing 7.1: Konvertierung String zu Grade: Duplizierter Code

Dieser duplizierte Code kann ausgelagert werden. Für derartige Konvertierungen werden von nun an Converter erstellt, die die Methode convert() implementieren müssen. Darin ist dann der duplizierte Code enthalten. Als Beispiel ist in Listing 7.2 der neu erstellte GradeConverter zu sehen.

```
1  public class GradeConverter extends Converter implements ↙
       ↳ ConverterInterface<Grade> {
2
3      public GradeConverter(String text) {
4          super(text);
5      }
6
7      @Override
8      public Grade convert() throws Exception {
9          float floatGrade = Float.parseFloat(text);
10         return new Grade(floatGrade);
11     }
12
13 }
```

Listing 7.2: GradeConverter: Auslagerung

Für die Verwendung des ausgelagerten Codes ist nun lediglich eine Zeile nötig. Listing 7.3 verwendet den GradeConverter, um den value String in die Note umzuwandeln.

```
1  grade = new GradeConverter(value).convert();
```

Listing 7.3: GradeConverter: Auslagerung

7.1.2 Code Smell: Large Class / Long Methods

Große Klassen sind im Code sehr unschön, da das für Entwickler nicht sehr leicht verständlich ist und Verantwortlichkeiten auch so eher nicht eingehalten werden. Stattdessen könnte man Möglichkeiten der Auslagerung nutzen, um diese Probleme zu lösen. Laut Listing 7.4 werden die Felder der CSV je nach zugehörigem Header im Switch nach anderen Regeln in die Entitäten konvertiert.

```
1  switch(header[i]){
2      case "module_id":
3          moduleId = value;
4      break;
5      case "attempt":
6          int attemptInt = Integer.parseInt(value);
```

```
7           attemptNumber = new NaturalNumber(attemptInt);
8       break;
9       case "grade":
10          float floatGrade = Float.parseFloat(value);
11          grade = new Grade(floatGrade);
12      break;
13      case "status":
14          passed = value.equals("bestanden");
15      break;
16      default:
17          throw new Exception("Unknown header in attempts CSV: " ↙
                ↳ + header[i]);
18  }
```

Listing 7.4: AttemptsCSV: Unschöne Switch Statements

Im ersten Schritt möchten wir den Code jeder Konvertierung auslagern. Laut Code Smell: Duplicated Code können diese Konvertierungen auch mehrfach vorkommen. Daher lohnt sich das erneut. In Listing 7.5 ist der Code mit selber Funktionalität und Converter-Pattern zu sehen.

```
1  switch(header[i]){
2      case "module_id":
3          moduleId = new EmptyConverter(value).convert();
4          break;
5      case "attempt":
6          attemptNumber = new ↙
               ↳ NaturalNumberConverter(value).convert();
7          break;
8      case "grade":
9          grade = new GradeConverter(value).convert();
10         break;
11     case "status":
12         passed = new StatusConverter(value).convert();
13         break;
14     default:
15         throw new Exception("Unknown header in attempts CSV: " ↙
               ↳ + header[i]);
```

```
16  }
```

Listing 7.5: AttemptsCSV: Unschöne Switch Statements

Alle verwendeten Converter implementieren das Interface ConverterInterface. Damit wird es möglich, die Header auf einen festen Datentypen ConverterInterface zu mappen. Anstelle eines Switch Statements kann nun eine Map<String, ConverterInterface> verwendet werden, der die Logik übernimmt und gleichzeitig in allen CSVReadern verwendet werden kann. Damit wurde die Verantwortlichkeit der Konvertierung ausgelagert und die Klasse etwas verkürzt.

PS: Ich habe übrigens versucht, das Switch Statement zu entfernen - es ist mir aber nicht gelungen. Eine dynamische Auswahl der Converter ist zwar möglich, aber eine Zuweisung zum erstellten Attempts-Objekt macht es extremst komplex.

7.2 Refactorings

Refactorings sollen dafür sorgen Code schöner zu gestalten, sodass andere Entwickler ihn besser verstehen können. Das Refactoring sollte dabei ein kontinuierlicher Prozess sein, der in der Entwicklung dauerhaft durchgeführt wird.

7.2.1 Streams

Der Entity Matcher enthält die Methode matchModules(), die in Listing 7.6 dargestellt ist. Anfangs sind modules, attempts und exams Listen, die nicht miteinander Verknüpft sind. Allerdings soll das Module-Aggregat erstellt werden. Dann hat ein Modul zugehörige Versuche und ein Versuch zugehörige Klausuren. Dabei entstehen in dieser Implementierung drei for-Schleifen ineinander, die sehr unübersichtlich werden. Die 5 Einrückungsstufen sorgen ebenfalls dafür (Wegen der If-Anweisungen). Das ist für einen Entwickler sehr schlecht lesbar.

```
1  public static void matchModules(List<Module> modules, ↙
       ↳ List<Attempt> attempts, List<Exam> exams){
2       for(Module module : modules){
3           List<Attempt> moduleAttempts = new ArrayList<>();
```

```
4           for(Attempt attempt : attempts){
5               if(attempt.getModuleId().equals(module.getModuleId())){
6                   moduleAttempts.add(attempt);
7               }
8               for(Exam exam : exams){
9                   List<Exam> attemptExams = new ArrayList<>();
10                  if(exam.getModuleId().equals(module.getModuleId()) ↙
                        ↳ && exam.getAttemptNumber().getNumber() ↙
                        ↳ == attempt.getNumber().getNumber()){
11                      attemptExams.add(exam);
12                  }
13                  attempt.setExams(attemptExams.toArray(new ↙
                        ↳ Exam[0]));
14              }
15          }
16          module.setAttempts(moduleAttempts.toArray(new ↙
                ↳ Attempt[0]));
17      }
18  }
```

Listing 7.6: EntityMatcher.matchModules(): Tiefe Verschachtelungen

Eine Verbesserung dieser Methode bietet Listing 7.7. Der Code etwas kürzer. Es gibt nur noch zwei for-Schleifen. Eine für die Iteration über die Module und einer über die Versuche. Dadurch entsteht eine maximale Einrückungsstufe von 2/3. Daher ist der Code einfacher zu Lesen. Auch das Verständnis wird wegen Nutzung der Streaming API besser. Die Bedingungen für ein Matchen kommen klarer heraus.

```
1       public static void matchModules(List<Module> modules, ↙
            ↳ List<Attempt> for(Attempt attempt : attempts) {
2           attempt.setExams(exams.stream().filter(exam ->
3               exam.getModuleId().equals(attempt.getModuleId()) &&
4               exam.getAttemptNumber().getNumber() == ↙
                    ↳ attempt.getNumber().getNumber()
5       ).collect(Collectors.toList()).toArray(new Exam[0]));
6       }
7       for(Module module : modules) {
8           module.setAttempts(attempts.stream().filter(attempt ->
```

```
 9                 attempt.getModuleId().equals(module.getModuleId())
10             ).collect(Collectors.toList()).toArray(new Attempt[0]));
11         }
12     }
```

Listing 7.7: EntityMatcher.matchModules(): Auflösung der Verschachtelung

7.2.2 Switch Statements

Im gesamten Projekt wurden Switch-Statements nur in den CSV-Konvertierungen vorgenommen. Listing 7.5 zeigt, wie einzelne Zeilen mithilfe ihrer zugehörigen Header in das korrekte Format verwandelt werden. Um duplizierten Code zu verwenden, wurden Converter eingesetzt, welche die Duplikate ersetzen. Das Switch-Statement bleibt weiterhin bestehen. Das schlechte an diesem Switch ist, dass für unterschiedliche CSV-Header andere Converter verwendet und gleichzeitig in unterschiedliche Variablen gespeichert werden sollen. Dadurch wird das Switch-Statement schwer zu warten.

Um dies zu ändern und die Bedingungen modularer zu gestalten, wird eine Map verwendet, in der zu jedem CSV-Header der zugehörige Wert gespeichert wird. Dadurch kann man über den Header-Key auf die Werte zugreifen, passend umwandeln und der jeweiligen Variable zuweisen.

Dieses Switch-Refactoring ist unter dem Commit 5f805c9a1bb686cb30489d7b4661ebe36a4d2af3 zu finden.

8 Entwurfsmuster

Bei der Entwicklung des Dualis-Bots wurden einige Entwurfsmuster verwendet. Diese Sektion zeigt einige verwendete Muster und erklärt deren Zweck und Grund.

8.1 Factory Pattern

Das Factory Pattern ist eines der meist implementierten Entwurfsmustern in Java. Anstatt dem Nutzer selbst die Erzeugung eines Objektes zu erlauben, sorgt eine Factory dafür. Vorteil ist, dass die Factory komplexe Logik verwenden kann, um dieses Objekt zu erzeugen. Oft wird dieser Code mehrfach benötigt. Daher sorgt die Factory ebenfalls dafür, dass es keinen duplizierten Code gibt.

Im Dualis-Bot wird die MessageFactory dafür verwendet, um eine Telegram-Nachricht (Message) basierend auf einer Klausur (Exam) zu generieren. Listing 8.1 zeigt den dafür benötigten Code. Die Methode „generateMessagesFor()“ bereitet die Parameter für zwei Messages vor, erstellt zwei Message-Instanzen, gibt diese in einer Liste zurück. Abhängig von der Versuchsnummer der Klausur wird der Text der zu sendenden Message verändert. Um diese Logik für jede Message-Generierung zu umschließen, bietet sich es an dieses Factory-Pattern zu verwenden.

```
1  public class MessageFactory {
2
3      public static List<Message> generateMessagesFor(Exam exam){
4          List<Message> messages = new ArrayList<>();
5
6          String textPrivate =
7          "Neue Note\n" +
8          "Klausur: " + exam.getName() + "\n" +
9          "Bewertung: " + exam.getRating().getRating();
10         Message messagePrivate = new Message(textPrivate, ↙
               ↳ Chatroom.PRIVATE);
11         messages.add(messagePrivate);
```

```
12
13          String textGroup =
14          "Neue Note\n" +
15          "Klausur: " + exam.getName();
16          if(exam.getAttemptNumber().getNumber() > 1){
17          textGroup += "\nHinweis: Es handelt sich um eine ↙
                ↳ Nachklausur.";
18          }
19          Message messageGroup = new Message(textGroup, ↙
                ↳ Chatroom.GROUP);
20          messages.add(messageGroup);
21
22          return messages;
23      }
24
25  }
```

Listing 8.1: MessageFactory: Beispiel Factory Pattern

8.2 Singleton Pattern

Das Singleton Pattern gehört ebenfalls zu den erzeugenden Entwurfsmustern. Es sorgt dafür, dass von einer Klasse nur genau ein Objekt erzeugt werden kann. Dies sorgt für einige Vorteile. Im Dualis-Bot wird das Singleton-Pattern im Dualis-Gateway verwendet. Das Dualis-Gateway liest während seiner Erzeugung (Konstruktor) alle CSV-Dateien des Dualis-Webscrapers. Es wandelt die gelesen Informationen in die Entitäten der Domäne um (ausgelagert). Die geladenen Entitäten werden im Dualis-Gateway-Objekt gespeichert und können abgefragt werden. Durch das Singleton Pattern soll vermieden werden, dass Daten durch Erzeugung des Dualis-Gateways mehrfach gelesen werden, da dadurch Laufzeitprobleme entstehen.

Listing 8.2 zeigt relevante Ausschnitte des Dualis-Gateways. Um das Singleton-Pattern umzusetzen, wird der Konstruktor *private*, die Klasse besitzt ein Attribut der DualisGateway-Datenstruktur, das direkt als Objekt instanziiert wird. Dies ist das einzige Objekt, das

jemals im Programm existieren darf. Auf die DualisGateway-Instanz kann nun über die statische Methode getInstance() zugegriffen werden.

```
1  public class DualisGateway {
2
3      private List<Module> oldModules = new ArrayList<>();
4      private List<Exam> oldExams = new ArrayList<>();
5      private List<Module> newModules = new ArrayList<>();
6      private List<Exam> newExams = new ArrayList<>();
7
8      private static DualisGateway instance = new DualisGateway();
9
10     private DualisGateway(){
11         ...
12         oldModules = modulesCsv.getModules();
13         oldExams = examsCsv.getExams();
14         ...
15         newModules = modulesCsv.getModules();
16         newExams = examsCsv.getExams();
17         ...
18     }
19
20     public static DualisGateway getInstance(){
21         return instance;
22     }
23
24     ...
25
26 }
```

Listing 8.2: DualisGateway: Beispiel Singleton Pattern