Dokumentation ASE-Projekt

Philipp Rall, TINF18B2, 5844601

1. Projektidee

Meine Idee ist mit Java ein kleines, lokales Desktop-Spiel „Tippduell" nach dem Vorbild von Typeracer zu entwickeln. D.h. das Spielziel besteht darin, einen vorgegebenen Text möglichst schnell korrekt abzutippen. In einem Wettkampfbereich soll hierzu zunächst die Schwierigkeit von der Spielleitung festgelegt werden, bevor ein entsprechender Text aus der Datenbank geladen wird. Anschließend tippt ein*e Spieler*in nach dem*der anderen den Text ab, der*die schnellste gewinnt die Runde. Neben dem Wettkampfbereich ist außerdem ein Bereich zur Verwaltung von Texten und Spielern sowie ein HighScore bzw. Statistikbereich pro Spieler*in angedacht, die Statistiken sollen ebenfalls in der Datenbank gespeichert werden. Auch ein Trainingsbereich für Spieler*innen ist geplant.

Generell dient die geplante Anwendung dem übergeordneten Zweck die Tippgeschwindigkeit der Spieler*innen auf kompetitive Art und Weise zu trainieren und zu verbessern.

Die vorliegende Softwarelösung dient zeitgleich als persönliche Beschäftigung und Untersuchung von Praxistauglichkeit gendergerechter Sprache.

2. Technologien

Die Anwendung ist in Java entwickelt, die Benutzeroberfläche basiert auf der in der Programmierenvorlesung erlernten Java-Swing-Technologie.
Als Build-Management-Tool wird Maven verwendet, als Datenbank eine SQLite-Datenbank.

3. Programmierprinzipien

SOLID

Single Responsibility Principle

Dieses Programmierprinzip sagt aus, dass jede Klasse nur eine Zuständigkeit, eine klar definierte Aufgabe besitzen soll. Das ermöglicht niedrige Komplexität und Kopplung. So besitzt die Klasse `Session` lediglich die Zuständigkeit der Anmeldelogik, die durch die beiden Methoden `login()` und `logoff` sowie das Attribut `loggedInPlayer` ausgedrückt wird. Ein weiteres Beispiel für die Einhaltung des Prinzips findet sich in den `Repository` -Klassen, von Game, Player, Stats oder Text. Diese haben jeweils nur die Aufgabe die Datenbankzugriffe für ein konkretes Repository-Objekt abzubilden.

Open Closed Principle

Das Open Closed Prinzip beschreibt, dass Klassen generell offen für Erweiterungen und geschlossen für Änderungen sein sollten. Durch Abstraktionen kann die Erweiterbarkeit gefördert werden, sodass bei Erweiterungen der bestehende Code nicht geändert werden muss. Konstrukte, die dies häufig nicht erfüllen

sind If-Else-Statements. Diese können nicht ohne Änderung um weitere Fälle erweitert werden. Dieser Fall lag auch in der `Login` Klasse vor, wie folgende Zeilen Code zeigen:

```
if (isDuringGame) {
    gameListener.startRound();
}else if(isAtAppStart){
    uiListener.drawUI();
}
```

In diesem Codeabschnitt wird die nächste Aktion nach erfolgter Anmeldung festgelegt. Diese ist abhängig davon, ob die Anmeldung während des Spiels oder zu Anwendungsstart erfolgt, was in einem If-Else-Statement anhand von booleschen Werten unterschieden wird. Würden in der weiteren Entwicklung weitere Anmeldedialoge benötigt werden, müsste diese Stelle geändert werden - sie erfüllt nicht das Open Closed Prinzip.
Deshalb wurde die Klasse überarbeitetet und die If-Else-Verzweigung durch Polymorphismus ersetzt, wie folgende Codestelle zeigt:

```
if(loginListener!=null){
    loginListener.goOn();
}
```

Die `Login` Klasse besitzt nun nur noch einen `loginListener`, das Interface `GameListener` erbt von `LoginListener`, UIListener wird durch `LoginListener` ersetzt. Somit kann unabhängig vom Kontext immer die Methode `goOn()` aufgerufen werden. Die konkrete gesetzte konkrete Implementierung entscheidet dann darüber, was bei `goOn()` passiert. Die Codestelle ist somit Open Closed. Die Änderung kann in diesem Commit nachvollzogen werden.

Liskov Substitution Principle

Das Liskov Substitution Principle besagt, dass Objekte eines abgeleiteten Typs als Ersatz für Instanzen ihres Basistyps funktionieren müssen ohne die Korrektheit des Programms zu ändern. Durch Einsatz des Prinzips können Invarianzen eingehalten werden. In Tippduell erben die Klassen `GameStats` und `PlayerStats` von der Klasse `Stats`. Zur Einhaltung des Liskov Substitution Principles müssen `Stats` -Objekte ohne Probleme durch Objekte der abgeleiteten Klassen ersetzt werden können. Dies ist möglich, da die Klasse `Stats` zu keinem Zeitpunkt instanziiert wird, sie ist abstrakt. Eine weitere Vererbung findet in der Anwendung keine Anwendung, daher kann dieses Prinzip nicht aufgezeigt werden.

Interface Segregation Principle

Dieses Programmierprinzip sagt aus, dass Anwender nicht von den Funktionen abhängig sein sollten, die sie nicht nutzen, und lässt sich durch passgenaue Interfaces realisieren. In der vorliegenden Anwendung werden ausschließlich passgenaue Interfaces verwendet, da diese jeweils nur von einer Klasse implementiert werden und lediglich im Rahmen des *Listener-Patterns* zum Einsatz kommen. Als Beispiel hiefür kann das `GameListener` -Interface angeführt werden, es wird nur von der Klasse `Game` implementiert und ist daher passgenau auf die benötigten Funktionen zugeschnitten. Somit ist dieses Prinzip erfüllt.

Dependency Inversion Principle

Das Dependency Inversion Principle verlangt, dass High-Level-Module nicht von Low-Level-Modulen, sondern beide von Abstraktionen abhängig sein sollten. Dies wird im Tippduell erreicht, da beispielsweise die Klassen `PlayerStats` und `GameStats` von der abstrakten Klasse `Stats` abhängen. Zudem ist generell, sofern möglich, eine Klasse in Tippduell abstrakt gestaltet.

GRASP

GRASP steht für General Responsibilty Assignment Software Patterns und bezeichnet eine Sammlung an Basisprinzipien. Von diesen soll zwei im Folgenden behandelt werden.

Low Coupling

Low Coupling verlangt eine geringe bzw. lose Kopplung zwischen Objekten, d.h. diese weisen nur geringe Beziehungen auf. Dadurch liegen nur geringere Abhängigkeiten vor, der Code wird verständlicher und einfach wiederverwendbar. Ein Beispiel für eine Klasse mit geringer Kopplung ist die Klasse `Round`, die lediglich eine Beziehung zu einem `GameListener` aufweist, um diesen über das Ende der Runde zu benachrichtigen, sowie eine Beziehung zu ihrer Benutzeroberfläche durch den Aufruf der statischen Methoden von `RoundUI` aufweist und zu `Text`. Diese Beziehung ist notwendig, um die Runde mit einem Text darzustellen.
Weitere Beispiele finden sich in den zahlreichen UI-Klassen, die jeweils für die Darstellung eines bestimmten Teils der Benutzeroberfläche zuständig sind und auch eine geringe Kopplung zu anderen Klassen aufweisen.

High Cohesion

Kohäsion ist allgemein ein Maß für den Zusammenhalt einer Klasse und beschreibt die semantische Nähe der Elemente einer Klasse. High Cohesion erfordert somit, eine hohe semantische Nähe aller Klassenelemente.
Eine hohe Kohäsion lässt sich vor allem in den UI-Klassen wiederfinden, da grundsätzlich für jeden Bestandteil der Benutzeroberfläche eine eigene Klasse existiert. So ist beispielsweise die Klasse `RoundUI` nur für die Darstellung der Benutzeroberfläche einer Runde zuständig und besitzt dafür die beiden Methoden `displayRoundFor` und `closeRound`. Ein anderes Beispiel stellt die Klasse `Registration` dar, die nur eine Methode `drawUI` aufweist, die die Benutzeroberfläche zu Registrierung zeichnet. Aber auch in Klassen wie `Rules` oder `Player` zeigt sich eine hohe Kohäsion.

Low Coupling und *High Cohesion* gehen beide in gewissermaßen mitdem Single-Responsibility-Prinzip einher. Besitzt jede Klasse nur eine Verantwortung, so liegt meist auch eine hohe Kohäsion innerhalb der Klasse und geringe Kopplung zu anderen Klassen vor.

DRY

DRY ist eine Abkürzung für *Don't Repeat Yourself!* und versucht jegliche unnötige Duplikation zu vermeiden. Eine Anwendung des Prinzips ist in der Darstellung der Anmeldebenutzeroberfläche in der Klasse `Login` zu finden. Diese muss an drei verschiedenen Stellen im Programm leicht verändert dargestellt werden (Programmstart, Rundenstart je Spieler, Spielende). Statt den Code dreimal zu wiederholen, wurde auf das Builder-Pattern gesetzt. D.h. es werden zunächst die Parameter der benötigten Darstellung gesammelt und es wird dann einmal in der `build`-Methode die Oberfläche erzeugt.

Ein weiteres Beispiel ist die Klasse `Game` bzw. auch das `GameUI`. Ein spiegelt grundsätzlich sowohl einen Wettkampf als auch ein Trainingsspiel wider und wird hierfür ebenfalls sinnvoll wiederverwendet. Durch das

Attribut `isCompetition` kann eine Unterscheidung vorgenommen werden, es muss kein Code dupliziert werden.

Als drittes Beispiel für DRY sollen die bereitgestellten Skripte zur Installation und Ausführung der Anwendung `init.sh` und `start.sh` angeführt werden. Automatisiertes Deployment stellt sicher, dass die einzelnen Deployment-Schritte nicht manuell wiederholt werden müssen verringert so bei häufigem Deployment Fehlerquellen.

4. Entwurfsmuster

Als Entwurfsmuster wurden im Tippduell zum einen das **Builder-Pattern** und zum anderen das **Observer-Pattern** bzw. **Listener-Pattern** angewandt.

Builder-Pattern

Grundsätzlich handelt es sich beim Builder-Pattern um ein Muster zur einfachen und schrittweisen Erstellung von komplexen Objekten in unterschiedlichen Ausführungen. In der Anwendung wird das Builder-Pattern nicht direkt zur Erstellung von `Login` Objekten bzw. Login-UIs eingesetzt. Diese werden in drei verschiedenen Abwandlungen in der Anwendung eingesetzt, daher ist der Einsatz des Builder-Patterns sinnvoll. Zudem erhöht das Builder-Pattern deutlich die Lesbarkeit und damit Wartbarkeit des Codes.

Aufrufe ohne Builder-Pattern:

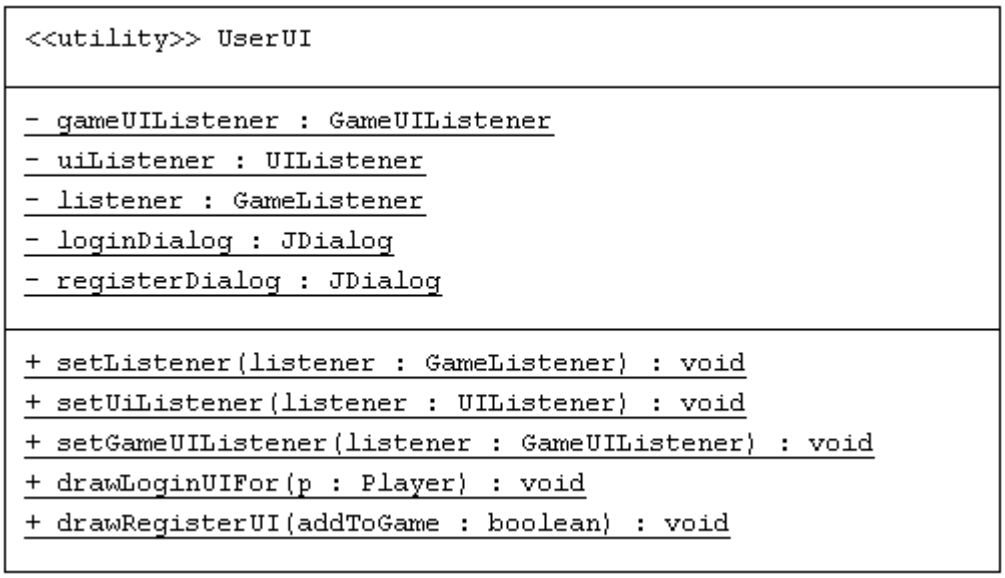
- Anmeldung in `ApplicationUI` zu Programmstart: `UserUI.setUIListener(this); UserUI.drawLoginFor(null);`
- Anmeldung in `Game` für nächsten Spieler: `UserUI.setListener(this); UserUI.drawLoginUIFor(nextPlayer);`
- Anmeldung in `Game` für Spielleiter: `UserUI.setListener(this); UserUI.drawLoginFor(originallyLoggedInPlayer);`

Aufrufe mit Builder-Pattern:

- Anmeldung in `ApplicationUI` zu Programmstart: `Login.create().withTitle("Anmeldung").atAppStart(this).withRegisterButton().build();`
- Anmeldung in `Game` für nächsten Spieler: `Login.create().withTitle("Anmeldung des nächsten Spielers").forPlayer(nextPlayer).duringGame(this).build();`
- Anmeldung in `Game` für Spielleiter: `Login.create().withTitle("Anmeldung des Spielleiters für Ergebnisse").forPlayer(originallyLoggedInPlayer).build();`

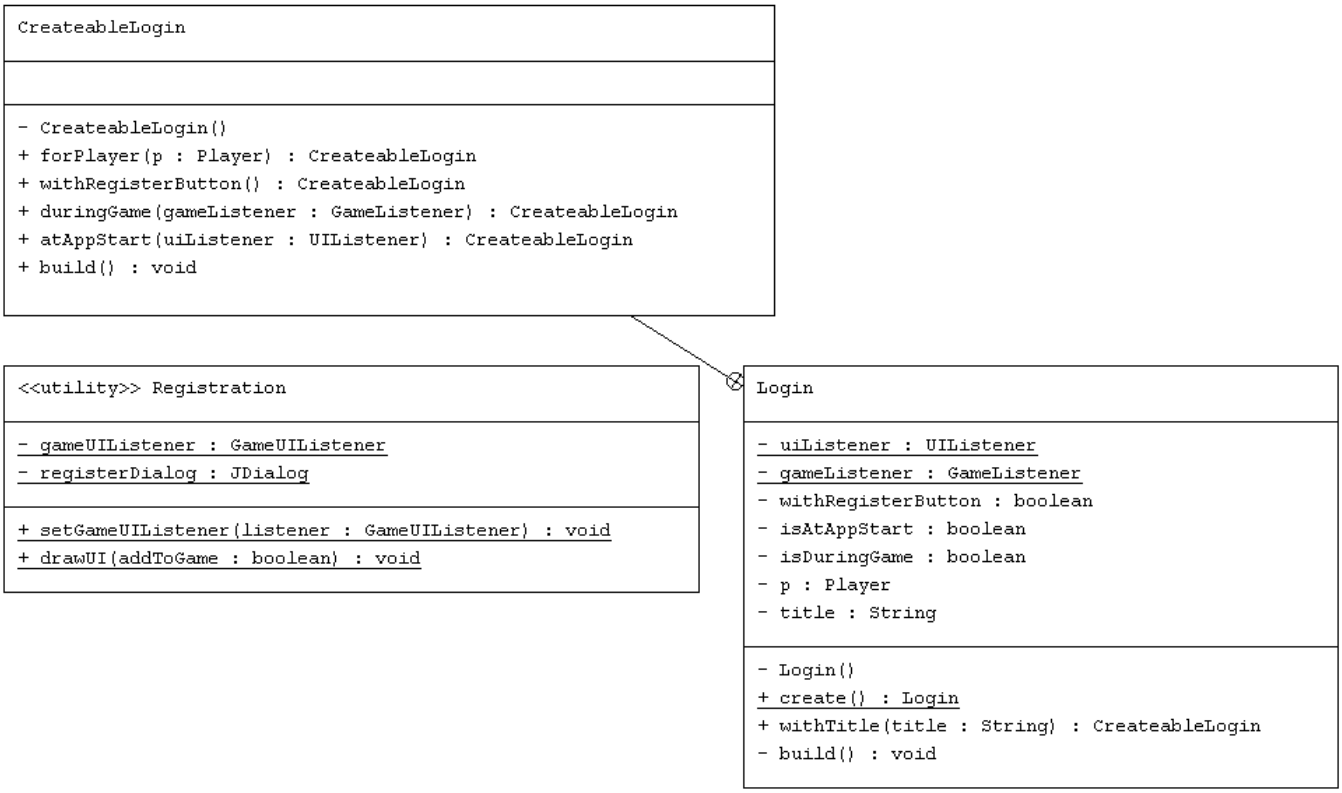
Durch das Builder-Pattern besteht nun auch die Möglichkeit einfach den Titel des Login-Dialogs anzupassen, so wird für den*die Spieler*in klarer, wofür der aktuelle Dialog dient. Durch das eingesetzte Builder-Pattern ist nun auch gewährleistet, dass immer ein Listener gesetzt wird, sofern dieser benötigt wird. Dies geschieht in einem Methodenaufruf (z.B. `duringGame(this)`) und nicht mehr in zwei, wie zuvor. Da in diesem Beispiel lediglich einen konkreten Erbauer gibt, kann auf weitere Interfaces verzichtet werden. Zudem gibt die `build`-Methode kein Objekt zurück, sondern zeichnet das entsprechende UI. Deshalb wird auch für das letztendliche Produkt kein Interface benötigt.

UML-Diagramm vorher:



Vor der Einführung des Patterns war die `UserUI`-Klasse für die Darstellung von Login und Registration verantwortlich, was zunächst auch einmal gegen das Single-Responsibility-Prinzip verstößt.

UML-Diagramm nachher:



Danach gibt es für die beiden Aufgaben zwei separate Klassen `Registration` und `Login`. Für `Registration` lohnt sich der Einsatz des Builder-Patterns nicht, da diese lediglich zwei verschiedene Ausprägungen aufweist.

Listener-Pattern

Es handelt sich hierbei um ein Verhaltensmuster, also einem Pattern zur Kommunikation zwischen Objekten und der Steuerung des Kontrollflusses einer Anwendung zur Laufzeit. Das Listener-Pattern ermöglicht eine automatische Reaktionen auf Zustandsänderungen und wird in der vorliegenden Anwendung für die Kommunikation zwischen Benutzeroberfläche und Applikationslogik eingesetzt.

Ein Interface wie beispielsweise `GameListener` gibt verschiedene Methoden vor, wie beispielsweise `startRoundFor(Player p)`, die die `Game` -Klasse implementiert. Die Klasse `Login` bekommt über die `duringGame(GameListener g)`-Methode dann eine `Game` Instanz als privaten Member gesetzt, über den das `Login` bei Bedarf die erwähnte Methode aufrufen kann. Die Klasse `Game` ist somit ein Observer, die Klasse `Login` ein Observable, das den Observer über die `startRoundFor(Player p)`-Methode benachrichtigen kann. Hierfür muss sich eine Instanz der Game-Klasse auf dem `Login` während des Buildprozesses registrieren. Da das Login nach dem Builder-Pattern erzeugt wird, liegt keine klassische Implementierung des Listener-Patterns vor, so gibt es z.B. keine Methode `addListener`. Das kommt auch daher, da jedes Login nur maximal einen `Game` -Observer besitzen kann.

5. Domain Driven Design

Analyse der Ubiquitous Language

Die Ubiquitous Language bezeichnet die Sprache, mit der Regeln, Prozesse und Konzepte einer Domäne ausgedrückt werden und legt einheitliches Vokabular zwischen Domänenexperten und Entwicklern fest. Die vorliegende Domäne ist die Messung und der Vergleich der Tippgeschwindigkeit in Form eines Wettkampfs. Im Folgenden sollen einige wesentliche Begriffe der festgelegten Projektsprache vorgestellt werden. Da für die technische Umsetzung die englische Sprache verwendet wurde, ist zu jedem Begriff der korrespondierende englische Begriff mit angegeben. Die Begriffe werden einheitlich zur Benennung von Klasse, Methoden oder Datenbanktabellen eingesetzt.

- **Spieler*in** bzw. **Player**: Eine Person, die sich in der Anwendung registriert und damit die Möglichkeit besitzt diese zu nutzen. Da die wesentliche Funktionalität der Anwendung in Spielen besteht, wurde diese Bezeichnung statt der allgemeinen Bezeichner eines*einer Nutzer*in gewählt.
- **Spiel** bzw. **Game**: Ein Spiel ist entweder ein Wettkampfs- oder ein Trainingsdurchlauf in der Anwendung.
- **Wettkampf** bzw. **Competition**: Ein Spiel mit mehreren Spieler*innen.
- **Training**: Ein Spiel mit nur einem*einer Spieler*in.
- **Text**: Ein Text, der im Rahmen eines Spiels abgetippt werden muss.
- **Runde** bzw. **Round**: Der Teilvorgang eines Spiels, in dem ein*e Spieler*in einen Text abtippt. Ein Training besitzt nur eine Runde, ein Wettkampf eine Runde pro Spieler*in.
- **Anmeldung** bzw. **Login**: Vorgang zur Identifzierung und Authentifzierung eines*einer Spieler*in. Kommt sowohl bei Anwendungsstart, als auch vor jeder Runde und am Ende eines Spiels zum Einsatz.
- **Registrierung** bzw **Registration**: Vorgang zur Erstellung eines*einer neuen Spieler*in.
- **Statistik** bzw. **Stats**: Statistische Daten über die Nutzung der Anwendung, beinhaltet zwei Teilbereiche:
    - **Spielstatistiken** bzw **GameStats**: Allgemeine Statistiken zum Spiel seit lokaler Installation.
    - **Spielerstatistiken** bzw **PlayerStats**: Spieler*in-bezogene Statistiken.

Weitere Analysen und Begründungen

Value Objects

Value Objects sind einfache, unveränderliche Objekte ohne eigene Identität. Sie werden nur durch ihren Wert beschrieben und besitzen keinen Lebenszyklus. Im Folgenden sollen einige Value Objects der Anwendung vorgestellt werden:

- **Player**: Ein `Player` kapselt das Wertekonzept eines*einer Spieler*in un weist die Eigenschaften `username` und `fullname` auf. Ein Player-Objekt besitzt in der Anwendung keine Identität, zwei Player sind identisch, sofern deren Benutzernamen und vollständige Namen übereinstimmen. Ändert ein*e Spieler*in den eine Eigenschaft, wird in der Anwendung immer ein neues gültiges Player-Objekt erzeugt. Somit hat ein `Player` auch keinen Lebenszyklus.
- **Rules**: Ein `Rules` Value Object kapselt das Wertekonzept eines Regelwerkes mit Eigenschaften wie der Schwierigkeit, der minimalen und maximalen Textlänge. Auch ein `Rule`-Objekt besitzt keinen Lebenszyklus und wird für jedes Spiel neu anhand der vom*von der Spieler*in festgelegten Parameter erzeugt.
- **Text**: Ein `Text` Value Object kapselt das Wertekonzept eines Textes mit Eigenschaften wie dem Texttitel, dem Textinhalt und der Textlänge. Ein `Text` wird in der Anwendung nicht verändert, sondern beim Auslesen aus der Datenbank jedes Mal neu erzeugt. Somit besitzt auch ein `Text` keinen Lebenszyklus und lässt sich als Value Object ausmachen.
- **StatsEntry**: Dieses Value Object `StatsEntry` kapselt die Werte eines Statistikeintrags eines Spiels. Auch dieses Value Object wird beim Auslesen aus der Datenbank jedes Mal neu erzeugt, nicht verändert und besitzt keinen Lebenszyklus.
- **PlayerStats** und **GameStats**: Diese Value Objects kapseln jeweils die Werte der spieler*inbezogenen oder allgemeinen Statistik und besitzen keine Identität oder Lebenszeit.

Die Unveränderlichkeit der behandelten Value Objects wurde auch in der Implementierung durch finale Klassen mit finalen Feldern und überschriebenen equals() sowie hashCode() Methoden umgesetzt. Die entsprechenden Dateien sind hier verlinkt: Player ,Rules ,Text, StatsEntry

Entities

Eine Entity besitzt im Gegensatz zu Value Objects eine eindeutige ID innerhalb der Domäne sowie weist einen Lebenszyklus auf, während dessen sie sich verändern kann. Die ID kann entweder ein natürlicher oder ein selbst generierter Surrogatsschlüssel sein.
Im Folgenden sollen einige Entitäten der Anwendung aufgeführt werden:

- **Competition**: Eine `Competition` besitzt einen selbst generierten Surrogatsschlüssel als ID, nämlich ein inkrementeller Zähler `competitionId`.
- **Training**: Ein `Training` besitzt ähnlich wie die `Competition` eine generierte selbst inkrementierende `trainingId` zur Identifikation. Beide werden in der Anwendung als `Game` -Objekt erzeugt, verändern ihre Eigenschaften (wie beispielsweise das Feld `playersLeft`) und weisen damit einen Lebenszyklus auf. Aus Datenbankperspektive betrachtet sind `Player` und `Text` auch Entities, da sich deren Persistierung durchaus ändern und auch identifiziert werden kann. Außerdem wird die Persistierung der beiden nicht jedes Mal neu erzeugt wird. Innerhalb der Anwendung sind diese beiden Objekte allerdings klar als Value Objects zu erkennen.

Aggregates

Aggregate gruppieren Entities und Value Objects zu gemeinsam verwalteten Einheiten, jede Entität gehört dabei zu einem Aggregat. In jedem Aggregat übernimmt eine Entity die Rolle des Aggregat Roots, alle Zugriffe auf das Aggregat erfolgen über das Aggregat Root. Folgende Aggregate lassen sich in der Anwendung ausmachen:

- **Player**: Dieses Aggregat beinhaltet lediglich das `Player` Value Object, Zugriffe auf das Aggregat erfolgen über den Benutzernamen der Datenbankentität des Players.
- **Text**: Dieses Aggregat beinhaltet lediglich das `Text` Value Object, Zugriffe auf das Aggregat erfolgen über den Texttitel der Datenbankentität des Texts.
- **Stats**: Dieses Aggregat beinhaltet die Entitäten *Competition* und *Training*, die beide als `Game` modelliert werden. Ein `Game` weist dabei einen bis mehrere *Player* sowie einen *Text* auf, der durch die *Rules* des `Game` charakterisiert wird.

Repositories

Repositories kapseln allgemein betrachtet die Logik für die Persistierung und Erzeugung von Entities, Value Objects und Aggregates. Sie vermitteln somit zwischen Domäne und Datenmodell und stellen der Domäne Methoden für den technischen Zugriff auf den Persistenzspeicher auf Granularität von Aggregates bereit. Sie können somit als Anti-Corruption-Layer zur Persistenzschicht angesehen werden.
Folgende Repositories werden in der Anwendung verwendet, sie sind nach den zugehörigen Aggregates benannt:

- **PlayerRepository**: Dieses Repository ist für die Verwaltung und den Zugriff auf Spieler*innen im Persistenzspeicher zuständig und beinhaltet die dazu notwendigen CRUD-Methoden.
- **TextRepository**: Dieses Repository ist für die Verwaltung und den Zugriff auf Texte im Persistenzspeicher zuständig und beinhaltet die dazu notwendigen CRUD-Methoden.
- **StatsRepository**: Dieses Repository ist für die Verwaltung und den Zugriff auf Spiel und Spieler*innenstatistiken zuständig und beinhaltet die dazu notwendigen CRUD-Methoden.

Hier finden sich sämtliche Repositories der Anwendung.

Architektur

Das vorliegende Programm wurde in einer Schichtenarchitektur mit den drei Schichten *Presentation*, *Domain* und *Data* entwickelt. Jede Schicht deckt dabei einen unterschiedlichen Aufgabenbereich ab:

- **Presentation**: Klassen zur Darstellung der Benutzeroberfläche, z.B. `GameUI`, `SettingsUI` oder `RoundUI`.
- **Domain**: Klassen für die eigentliche Spiellogik, steuern Spielfluss maßgeblich, z.B. `Game` oder `Round`.
- **Data**: Klassen für grundlegende Objekte der Anwendung, z.B. `Player`, `Text` oder `Stats` mit den dazugehörigen Repositories. Die Repositories wie z.B. das `PlayerRepository` beinhalten Code zum Verwalten der dazugehörigen Datenbankobjekte. In dieser Schicht werden alle Datenbankzugriffe realisiert, diese können dadurch für die obigen Schichten durch Methodenaufrufe auf den Repositories abstrahiert werden. Dabei gilt allgemein, dass jede Schicht nur von den unterliegenden Schichten abhängt und somit unabhängig von den überliegenden Schichten ist. Konkret bedeutet das, dass beispielsweise die *Data*-Schicht unabhängig von den anderen beiden Schichten ist, die oberen beiden Schichten aber Methoden der *Data*-Schicht verwenden.

Die Entscheidung viel auf diese Art der Schichtenarchitektur, da die Anwendung die drei klassischen Schichten *Datenbank*, *Datenverarbeitung* und *Präsentation* besitzt und diese somit ideal durch eine Schichtenarchitektur abgebildet werden können.

6. Unit Tests

Unit Tests testen einzelne Komponenten des Systems und stellen deren Funktionalität unabhängig von anderen Komponenten sicher. Dazu ist es wichtig, andere Teile durch Stellvertreter, sogenannte Mocks, zu ersetzen um eine Unabhängigkeit zu gewährleisten. Im Tippduell wurden 21 Testfällen realisiert. Zum Testen wird das Testing-Framework *JUnit5* verwendet, Mocks werden mit dem Mocking-Framework *mockito* erzeugt.

Tests

Als beispielhafte Tests soll hier der Test `CheckCurrentInputCharTest` angeführt werden, da diese eine wesentliche Funktionalität des Spiels testet.

CheckCurrentInputCharTest

Dieser Test testet die Methode `checkCurrentInputChar` der Klasse `Round`, die die Funktionalität repräsentiert, einen eingegebenen Buchstaben des Nutzers auf Übereinstimmung mit dem aktuell erforderlichen Buchstaben des Textes zu überprüfen. Dafür wird in diesem Test eine Instanz der Klasse `Round` mit einem gemockten `Text` -Objekt erzeugt, das wie folgt erzeugt wurde:

```
Text text = mock(Text.class);
when(text.getText()).thenReturn("Testtext");
```

Dadurch kann eine Unabhängigkeit von der tatsächlichen Klasse `Text` erzielt und dennoch das Szenario erfolgreich getestet werden. Der Test umfasst vier Testfälle, die die Komponente bei korrekt und inkorrekt eingegebenen Char sowie bei korrekt und inkorrekt eingegebener Char-Folge überprüft.
Da die Methode lediglich Chars als Eingabe akzeptiert, müssen falsche Eingabetypen nicht getestet werden. Sie sind nicht möglich.

A-TRIP-Regeln

Die A-TRIP-Regeln können als Eigenschaften guter Tests angesehen werden. Im Folgenden sollen diese kurz vorgestellt und deren Einhaltung im Projekt erläutert werden.

- **Automatic**: Tests müssen einfach durch einen Befehl ausführbar sein und ihre Ergebnisse selbst überprüfen. Im vorliegenden Projekt können alle Tests durch den Befehl `mvn test` gestartet werden, in der Konsole sind daraufhin die Ergebnisse der Tests, also ob sie bestanden oder fehlgeschlagen haben, auf einen Blick sichtbar. Somit gilt diese Eigenschaft als erfüllt. Als Nachweis hier ein Screenshot aus der Konsole:

```
[INFO] Results:
[INFO]
[INFO] Tests run: 21, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  29.725 s
[INFO] Finished at: 2021-05-31T16:05:50+02:00
[INFO] ------------------------------------------------------------------------

Process finished with exit code 0
```

- **Thorough**: Tests müssen alles Notwendige überprüfen, d.h. alle relevanten Testfälle abdecken. Dies ist durch ausführliche Auseinandersetzung mit den Testszenarien sichergestellt worden.

- **Repeatable**: Tests müssen beliebig wiederholbar sein und immer das gleiche Ergebnis liefern. Dies wird im vorliegenden Projekt durch den Einsatz von Mock-Objekten erreicht.
- **Independent**: Tests dürfen keine Abhängigkeit zu anderen Tests haben. Dies wird im vorliegenden Projekt durch SetUp-Methoden, die vor jedem Test durchgeführt werden sichergestellt. So besitzt jeder Test, unabhängig von dem Zeitpunkt seiner Ausführung, die gleiche Testumgebung.
- **Professional**: Tests unterliegen denselben Qualitätsstandards wie Produktivcode. Hierzu werden im Projekt diesselben Prinzipien, wie z.B. DRY, auch für Tests angewandt und beispielsweise Coderedundanzen in SetUp-Methoden mit @BeforeEach-Annotation ausgelagert. So muss der Code nur einmal geschrieben werden und wird dennoch vor jedem Testszenario ausgeführt.

Code Coverage

Dieses Projekt weist keine besonders hohe Code Coverage auf, wofür in diesem Abschnitt einige Gründe angeführt werden sollen.

- Die innere Komplexität der vorhanden Klassen nicht sehr hoch ist, somit besteht kein hohes Testbedürfnis bzw. wenig Möglichkeiten für sinnvolle Tests.
- Es existieren viele Klassen, die lediglich Daten kapseln oder ein Objekt der realen Welt modellieren und keine Logik aufweisen (z.B. `Player`, `Stats` oder `Text`). Das testen dieser Klassen würde sich auf das Testen von Konstruktoren, Gettern und Equals-Methoden beschränken, was nicht als sinnvoll angesehen wird.
- Durch Anwendung des Single-Responsibility-Prinzips weisen viele Klassen private Methoden auf, die nicht leicht getestet werden können. Zudem ist die Sinnhaftigkeit von Tests privater Methoden diskutabel, da diese meist keine eigenständige Komponente darstellen.
- Es ist sehr viel Swing-UI-Code vorhanden, der sich nur schwer testen lässt. Insgesamt sind 30% aller Klassen (inklusive UI-Klassen) und 9% der Codezeilen durch Unittests abgedeckt.

7. Refactoring

Large Class & Long Method

Ein Code Smell, nämlich eine Large Class mit Long Methods, kann in der `SettingsUI` -Klasse hierdurch identifiziert werden. Diese Klasse beinhaltet den UI-Code für die Nutzerverwaltung und Textverwaltung mit jeweils einer langen Methode für die beiden Bereiche. Allein durch diese Beschreibung wird klar, dass die Klasse nicht dem Single Responsiblitiy Prinzip folgt. Deshalb soll in drei Schritten ein Refactoring durchgeführt werden, der Erfolg des Refactorings der Long Methods wird durch die Codezeilen pro Methode gemessen.

1. Zunächst muss die Klasse `SettingsUI` in zwei Klassen `UserManagementUI` und `TextManagementUI` aufgeteilt werden. Diese Aufteilung ist in diesem Commit zu sehen. So kann sichergestellt werden, dass jede Klasse nur eine Aufgabe innehat.
2. Anschließend kann das Long Method-Problem mittels Extract-Method-Refactoring angegangen werden und in `UserManagementUI` die lange Methoden `getUserManagementUI` in kleinere Methoden aufgeteilt werden, sodass jede Methode nur eine Aufgabe erfüllt. Dieser Zwischenstand ist in diesem Commit sichtbar. So konnte die `getUserManagementUI` Methode mit 149 Zeilen Code in neun Methoden mit jeweils maximal 38 Zeilen Code aufgeteilt werden. Das steigert die Lesbarkeit und Wartbarkeit erheblich. Ein ähnlicher Effekt konnte in der `TextManagementUI` -Klasse erzielt werden, dort ließ sich die lange Methode mit ursprünglich 151 Codezeilen in elf kleine Methoden mit maximal 36 Codezeilen aufgeteilt werden. Siehe hierzu diesen Commit.

3. Allerdings ist das Ergebnis im Bereich des User-Managements noch nicht zufriedenstellend, es fällt auf, dass Methoden wie `changeUsernameIfNotEmpty` eigentlich kein UI-Code, sondern Domain-Code sind. Deshalb sollen diese Methoden in einer neuen Klasse `UserManagement` in der domain-Schicht zusammengefasst werden. Außerdem ist eine Umbenennung sinnvoll, da diese nach dem Refactoring lediglich einen Wahrheitswert zurückliefern, ob die Änderung erfolgreich war bzw. ob die Eingabefelder im UI geleert werden sollen. Die Methoden heißen deshalb z.B. `isUsernameChaned`. Siehe hierzu diesen Commit.

Code Style

Zur objektiven, metrikbasierten Identifikation von Code Smells wird für dieses Projekt das Analysetool *Codacy* eingesetzt. Anhand des Quality-Scores dort, kann der konkrete Erfolg des Refactorings bewertet werden. Im zweiten Refactoring sollen die in Codacy aufgezeigten Issues bezüglich Code-Style und Fehleranfälligkeit angegangen werden. Zu Beginn des Refactorings liegen 55 Issues vor, davon 43 Code-Style-Issues und 12 Error-Prone-Issues. Hier kann das Codacy-Dashboard eingesehen werden. Ziel des Refactorings ist es, alle Issues zu lösen, um einen guten Code-Style und geringe Fehleranfälligkeit zu ermöglichen.

Folgende Probleme konnten dabei behoben werden:

- Entfernung bzw. Ersetzung ungenutzer Imports oder unnötiger Import-Alls (*)
- Entfernung von Logik-Inversionen, aus `!(a==b)` wird `(a!=b)`
- Reduzierung der Anzahl globaler Variablen, Umwandlung zu lokalen Variablen, wenn sinnvoller (z.B. `rule` in `Game` oder `endTime` in `Round`)
- Objektvergleiche mit != wurden durch equals ersetzt
- Anteil der Final-Attribute konnte erhöht werden
- Kombination von unnötig verschachtelten If-Statements wie in `UserManagement`

Siehe zu den Ergebnissen des Refactorings diesen Commit sowie diesen Commit, der den Code-Style ebenfalls verbessert. Es konnten alle Codacy-Issues gelöst werden, wie folgender Ausschnitt des Codacy-Dashboards zeigt:

Außerdem wurden noch Testmethoden umbenannt, da es nicht sinnvoll ist, dass alle Testmethoden des Tests `CheckTextLeftTest` mit `checkTextLeftAfter` starten. Die Umbenennung zu z.B. `forCorrectInputs` ermöglicht eine bessere Lesbarkeit. Das Ergebnis ist in diesem Commit sichtbar.

Duplicated Code

Als drittes angewandtes Refactoring soll hier die Beseitigung von duplicated Code in der Klasse `StatsRepository` angeführt werden. Vor dem Refactoring lag wie hier ersichtlich der Codeteil zur Formatierung der Datenbankinhalte und Erzeugung eines `HistoryEntry`-Objektes dreimal in verschiedenen Methoden in nahezu gleicher Ausführung vor.
Dem konnte durch die Einführung einer neuen statischen `getFormattedStatsEntry`-Methode entgegengewirkt werden. Somit wurde das *Extract Method*-Refactoring angewandt. Zudem wurde die Methode `getHighscoreList` erweitert zu `getHighscoreListFor` und kann dadurch die Highscoreliste für das gesamte Spiel oder eine*n einzelnen Spieler*in liefern, was ebenfalls den duplicated Code reduziert.
Des Weiteren wurde im Rahmen dieses Refactorings die Klasse `HistoryEntry` zu `StatsEntry` umbenannt, weil diese faktisch nicht nur Einträge der Historie, sondern auch Einträge des Highscores widerspiegelt und die Benennung somit nicht mehr der Verwendung entsprach.

Die Ergebnisse des Refactorings sind hier ersichtlich.