## VU Software Engineering 1

# Abgabedokument

## Teilaufgabe 2 (Implementierung des Servers)

|  |  |
| --- | --- |
| **Nachname, Vorname:** | **Wendl, Christopher** |
| **Matrikelnummer:** | A01268564 |
| **E-Mail Adresse:** | [wendl.chris@gmx.at](mailto:wendl.chris@gmx.at) |
| **Datum:** | 08.12.2017 |

### Änderungen zu Abgabge 1

#### Kritik bei Abgabe 1:

Persistenz

* **Spielerbewegungen werden nicht gespeichert**

**Das Stimmt so nicht!!**

Wie auf folgendem Bild aus Abgabe 1 zu sehen ist, werden Spielerbewegungen in jeder Runde sehr wohl gespeichert. Genauer gesagt wird in jeder Runde die aktuelle und die letzte Position des Spielers gespeichert. (Dargestellt durch Standort\_alt und Standort\_neu)

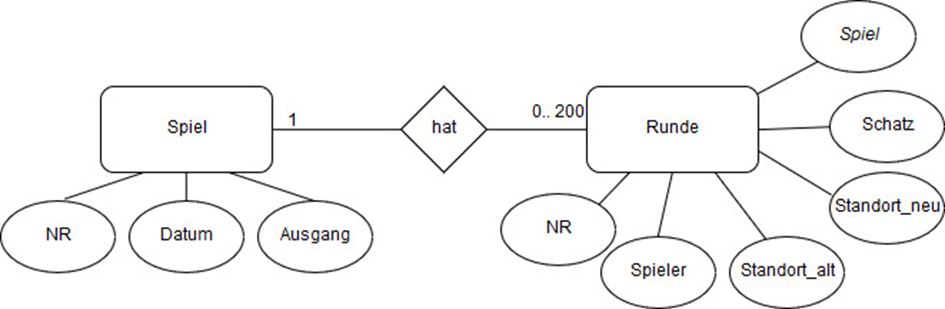


Abbildung 1 Abgabe 1 Datenbank

Um dies zu verdeutlichen wurden diese beiden Attribute nun umbenannt:

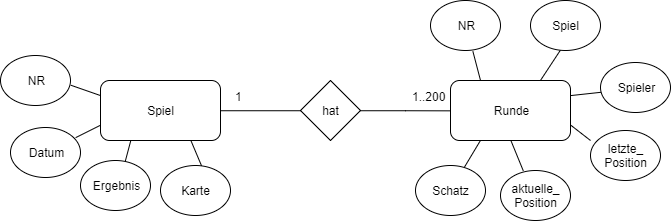


Abbildung 2 Abgabe 2 Datenbank Update

Die beiden Attribute waren in Abgabe 1 vom Typ Double. Die Idee hierbei war zu unterscheiden ob es sich um eine Bewegung entlang der X Achse (<1.0) oder entlang der Y Achse (>1.0) handelt.

Da die Karte aber eine fixe Größe von 8x8 Feldern hat ist Double als Datentyp nicht notwendig und Integer wird nun stattdessen verwendet.

Der Standort wird nun wie folgt erfasst:

* Die X-Achse beschreibt die Werte 0-7
* Die Y-Achse beschreibt die Werte 10-70
* Durch die fixe Größe kann nie einer der beiden Werte größer als 7 sein. D.h: Die Position ganz links unten (im Ursprung) hat den Wert 0 (genauer gesagt 00, das lässt sich aber nicht speichern und ist auch nicht notwendig). Die Position ganz rechts oben hat daher den Wert 77. Durch eine Integer-Division (d.h.: keine Nachkommastellen) geteilt durch 10 lässt sich der Wert auf der Y-Achse herauslesen. Den Wert auf der X-Achse bekommt man durch folgende Berechnung:

Aktuelle Position – Wert auf Y-Achse \* 10

Beispiel: Spieler steht auf Feld 47

* Also 5 Felder nach oben und 8 Felder nach rechts (0…4 & 0…7)
* Wert auf Y-Achse = 47 / 10 = 4 (Durch Java „trunctation“)
* Wert auf X-Achse = 47 -4\*10 = 7

Die Positionen wurden in Abgabe 1 bereits gespeichert. Lediglich der Datentyp war ungünstig gewählt.

* **Karten werden nicht gespeichert**

Da in sqlite keine Arrays gespeichert werden können wird die Karte in Abgabe 2 als Text gespeichert. Hierbei wird die JSON-Map zu einem String konvertiert und kann daher auch einfach wieder geparsed werden falls notwendig. Siehe Abbildung 2 Abgabe 2 Datenbank Update.

* Neues String Feld Karte
* Datum zu String geändert

Die sqlite Statements ändern sich daher wie folgt:

Create Table Spiel (

Nummer Integer,

Datum String,

Ergebnis String,

Karte String,

Primary Key (Nummer)

);

Create Table Runde (

Nummer Integer,

Spieler String,

letzte\_Position Integer,

aktuelle\_Position Integer,

Schatz Boolean,

Spiel Integer,

Primary Key (Nummer),

Foreign Key (Spiel) References Spiel(Nummer)

);

Insert Into Spiel (Nummer, Datum, Ergebnis, Karte)

Values (1, „11-3-2017: 220000“, „Spieler1 hat gewonnen“,“{JSON\_MAP}“);

Insert Into Runde (Nummer, Spieler, letzte\_Position, aktuelle\_Position, Schatz)

Values (1, „Spieler1“, 11, 21, false);

### Datenbankumsetzung

Die Umsetzung der Datenbank mittels Hibernate und prepared-Statements ergab folgende Ergebnisse:

3 Klassen:

* Spiel.java
* Runde.java
* HibernateMain.java

Die Idee dahinter ist, dass der Server eine Instanz eines HibernateMain Objektes erstellt und über dieses auf die Datenbank zugreifen kann. Um das zu ermöglichen wurden verschiedene Methoden bereitgestellt:

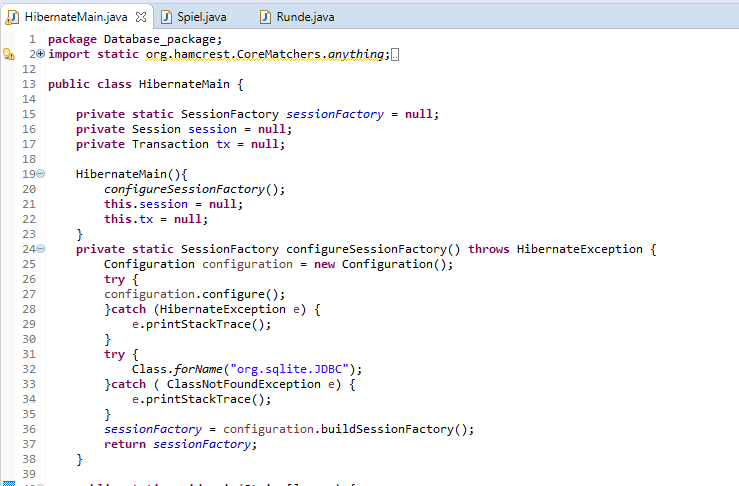


Abbildung 3 Konstruktor von HibernateMain,java

Wie in Abbildung 3 zu sehen ist, ruft der Konstruktor der Klasse HibernateMain zunächst die von den Slides bereitgestellte, leicht abgeänderte configureSessionFactory() Methode auf.

Diese Methode wurde mit zusätzlichen try-catch Klauseln versehen, da es zu Beginn probleme mit der hibernate.cfg.xml Konfigurationsdatei gab, welche unter Anderem nicht gefunden wurde.

Anschließend werden die Variablen session und tx der Klasse noch initialisiert und null gesetzt.

Folgende Methoden stehen dem Server nun zur Verfügung:



Abbildung 4 SaveGame Methode von HibernateMain

Die SaveGame() Methode speichert ein ihr übergebenes Objekt der Klasse Spiel in der Datenbank ab. Für Fehlerbehandlung während dieses Prozesses wurde eine try-catch Klausel verwendet.

Anmerkung: Jede dieser Methoden öffnet eine Session zu Beginn und schließt diese wieder am Ende. Ich habe mich gegen eine globale Session entschieden da, falls in anderen Methoden etwas schief läuft, dies zu schwerwiegenden eventuell nicht auffindbaren Fehlern führen könnte, wenn Methoden auf Sitzungen zugreifen die es beispielsweise nicht mehr gibt.

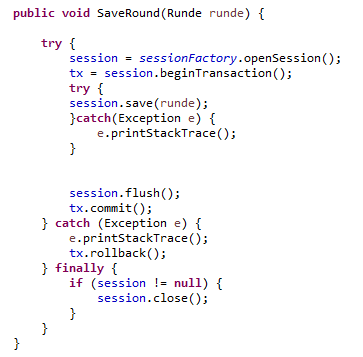


Abbildung 5 SaveRound Methode von HibernateMain

Diese Methode speichert eine Runde und ist ansonsten wie die SaveGame Methode aufgebaut.

Da Runden über einen Fremdschlüssel referenziert werden muss das zugehörige Spiel nicht als Objekt bekannt sein, oder mitübergeben werden. Es genügt ein Eintrag in Runde.Spiel um eine Runde einem Spiel zugehörig sein zu lassen.

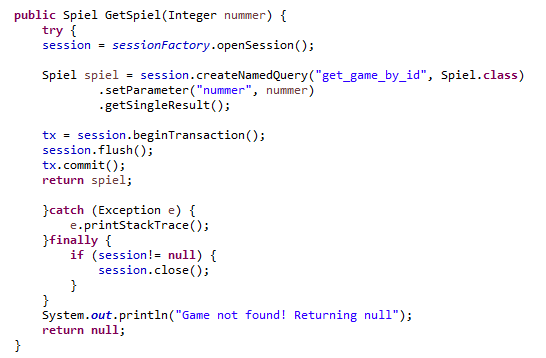


Abbildung 6 GetSpiel Methode von HibernateMain

Die Methode GetSpiel wird verwendet um anhand einer übergebenen Id nach einem Spiel zu suchen. Sollte ein Spiel gefunden werden, so wird es als Objekt der Klasse Spiel zurückgegeben. Wird kein Spiel gefunden, so wird null zurückgegeben und es muss daher immer erst geprüft werden, ob das Spiel existiert bevor darauf zugegriffen wird. Als Hinweis wird in der Konsole noch eine Meldung ausgegeben, dass kein Spiel gefunden wurde.



Abbildung 7 GetGewinner Methode von HibernateMain

Die GetGewinner Methode verhält sich im Prinzip wie die GetSpiel Methode indem sie ein Spiel anhand dessen ID sucht. Einziger Unterschied hier ist, dass diese Methode den String mit dem darin enthaltenen Gewinner zurück gibt anstelle des gesamten Spieles.

Diese Methode wurde hauptsächlich implementiert um den Anforderungen der Abgabe gerecht zu werden. Sie stellt zusammen mit der Abfrage in Spiel.java die erste beispielhafte Abfrage von zweien dar.



Abbildung 8 GetRunden Methode von HibernateMain

Diese Methode nimmt zwei Parameter an. Der erste Wert gibt die Nummer der Runde an ab der gesucht werden soll und der zweite Wert bis zu welcher Runde gesucht werden soll.

Sie gibt eine Liste vom Typ Runde zurück die alle gesuchten Runden enthält sofern sie existieren.

Diese Methode stellt zusammen mit der Query die zweite beispielhafte Abfrage dar.

Die Main-Methode wurde zum Testen der oben genannten Methoden benutzt. Ich kam zu folgenden Ergebnissen:

BeispielQuery1: (Gewinner eines Spieles)

(Siehe Main-Methode)

18:00:13.405 [main] DEBUG org.hibernate.internal.util.EntityPrinter - Listing entities:

18:00:13.405 [main] DEBUG org.hibernate.internal.util.EntityPrinter - Database\_package.Spiel{datum=Heute, ergebnis=Frank hat das Spiel gewonnen, karte=Testmap, nummer=2}

18:00:13.405 [main] DEBUG org.hibernate.resource.jdbc.internal.LogicalConnectionManagedImpl - Initiating JDBC connection release from afterTransaction

18:00:13.405 [main] DEBUG org.hibernate.resource.jdbc.internal.LogicalConnectionManagedImpl - Initiating JDBC connection release from afterTransaction

Frank hat das Spiel gewonnen

BeispielQuery2 : (Alle Runden von Runde 10 bis 20)

(Siehe Main-Methode)

18:00:13.415 [main] DEBUG org.hibernate.internal.util.EntityPrinter - Listing entities:

18:00:13.415 [main] DEBUG org.hibernate.internal.util.EntityPrinter - Database\_package.Runde{spieler=spieler, aktuelle\_Position=10, spiel=Database\_package.Spiel#2, nummer=19, letzte\_Position=0, schatz=false}

18:00:13.415 [main] DEBUG org.hibernate.internal.util.EntityPrinter - Database\_package.Runde{spieler=spieler, aktuelle\_Position=10, spiel=Database\_package.Spiel#2, nummer=20, letzte\_Position=0, schatz=false}

18:00:13.415 [main] DEBUG org.hibernate.internal.util.EntityPrinter - Database\_package.Runde{spieler=spieler, aktuelle\_Position=10, spiel=Database\_package.Spiel#2, nummer=17, letzte\_Position=0, schatz=false}

18:00:13.415 [main] DEBUG org.hibernate.internal.util.EntityPrinter - Database\_package.Runde{spieler=spieler, aktuelle\_Position=10, spiel=Database\_package.Spiel#2, nummer=18, letzte\_Position=0, schatz=false}

18:00:13.415 [main] DEBUG org.hibernate.internal.util.EntityPrinter - Database\_package.Runde{spieler=spieler, aktuelle\_Position=10, spiel=Database\_package.Spiel#2, nummer=15, letzte\_Position=0, schatz=false}

18:00:13.415 [main] DEBUG org.hibernate.internal.util.EntityPrinter - Database\_package.Runde{spieler=spieler, aktuelle\_Position=10, spiel=Database\_package.Spiel#2, nummer=16, letzte\_Position=0, schatz=false}

18:00:13.415 [main] DEBUG org.hibernate.internal.util.EntityPrinter - Database\_package.Runde{spieler=spieler, aktuelle\_Position=10, spiel=Database\_package.Spiel#2, nummer=13, letzte\_Position=0, schatz=false}

18:00:13.415 [main] DEBUG org.hibernate.internal.util.EntityPrinter - Database\_package.Runde{spieler=spieler, aktuelle\_Position=10, spiel=Database\_package.Spiel#2, nummer=14, letzte\_Position=0, schatz=false}

18:00:13.415 [main] DEBUG org.hibernate.internal.util.EntityPrinter - Database\_package.Runde{spieler=spieler, aktuelle\_Position=10, spiel=Database\_package.Spiel#2, nummer=11, letzte\_Position=0, schatz=false}

18:00:13.415 [main] DEBUG org.hibernate.internal.util.EntityPrinter - Database\_package.Runde{spieler=spieler, aktuelle\_Position=10, spiel=Database\_package.Spiel#2, nummer=12, letzte\_Position=0, schatz=false}

18:00:13.415 [main] DEBUG org.hibernate.internal.util.EntityPrinter - Database\_package.Runde{spieler=spieler, aktuelle\_Position=10, spiel=Database\_package.Spiel#2, nummer=10, letzte\_Position=0, schatz=false}

18:00:13.415 [main] DEBUG org.hibernate.internal.util.EntityPrinter - Database\_package.Spiel{datum=Heute, ergebnis=Frank hat das Spiel gewonnen, karte=Testmap, nummer=2}

18:00:13.415 [main] DEBUG org.hibernate.resource.jdbc.internal.LogicalConnectionManagedImpl - Initiating JDBC connection release from afterTransaction

18:00:13.415 [main] DEBUG org.hibernate.resource.jdbc.internal.LogicalConnectionManagedImpl - Initiating JDBC connection release from afterTransaction

Runde Nummer: 10

Spieler: spieler

Letzte Position: 0

Aktuelle Position: 10

Schatz: Nein

Spiel Nummer: 2

Runde Nummer: 11

Spieler: spieler

Letzte Position: 0

Aktuelle Position: 10

Schatz: Nein

Spiel Nummer: 2

Runde Nummer: 12

Spieler: spieler

Letzte Position: 0

Aktuelle Position: 10

Schatz: Nein

Spiel Nummer: 2

Runde Nummer: 13

Spieler: spieler

Letzte Position: 0

Aktuelle Position: 10

Schatz: Nein

Spiel Nummer: 2

Runde Nummer: 14

Spieler: spieler

Letzte Position: 0

Aktuelle Position: 10

Schatz: Nein

Spiel Nummer: 2

Runde Nummer: 15

Spieler: spieler

Letzte Position: 0

Aktuelle Position: 10

Schatz: Nein

Spiel Nummer: 2

Runde Nummer: 16

Spieler: spieler

Letzte Position: 0

Aktuelle Position: 10

Schatz: Nein

Spiel Nummer: 2

Runde Nummer: 17

Spieler: spieler

Letzte Position: 0

Aktuelle Position: 10

Schatz: Nein

Spiel Nummer: 2

Runde Nummer: 18

Spieler: spieler

Letzte Position: 0

Aktuelle Position: 10

Schatz: Nein

Spiel Nummer: 2

Runde Nummer: 19

Spieler: spieler

Letzte Position: 0

Aktuelle Position: 10

Schatz: Nein

Spiel Nummer: 2

Runde Nummer: 20

Spieler: spieler

Letzte Position: 0

Aktuelle Position: 10

Schatz: Nein

Spiel Nummer: 2

18:00:42.689 [pool-1-thread-1] DEBUG org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl - Connection pool now considered primed; min-size will be maintained

Die Einträge sind Test-Einträge und daher alle gleich, lediglich die Nummer der Runde unterscheidet sich.

Main-Methode:

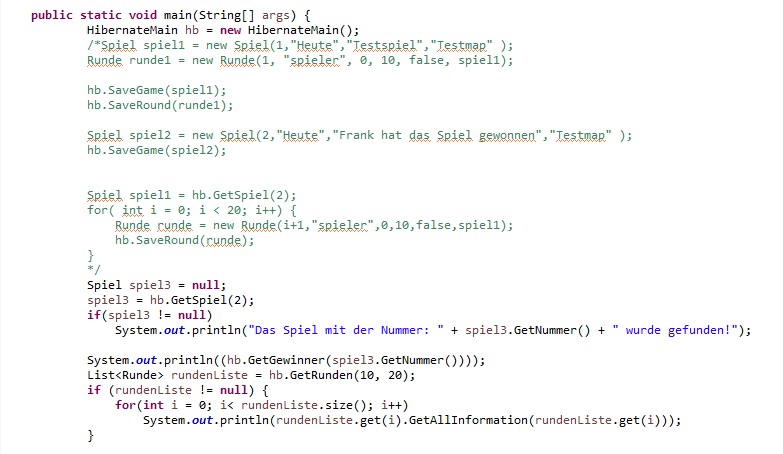


Abbildung 9 Main-Methode von HibernateMain

Die Main Methode wird hauptsächlich zum Testen der Datenbank benutzt und ist nicht essentiell für das Spiel. Später kann sie also auch entfernt werden.

Der Auskommentierte Code wurde beim ersten Start ausgeführt und legt die Testdaten an. In diesem Fall zwei Spiele und insgesamt 21 Runden.

Prepared Statements:

Die folgenden prepared Statements existieren in den Klassen Spiel.java bzw. Runde.java:

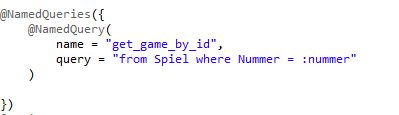


Abbildung 10 Prepared Statement in Spiel.java

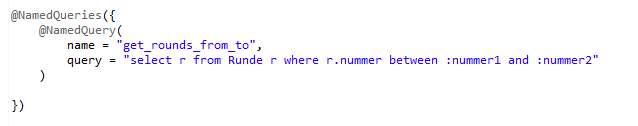


Abbildung 11 Prepared Statement in Runde.java

#### Netzwerkkommunikation

Folgende Rest-Routes wurden in Abgabe1 definiert:

**Gegenüberstellung Anfragen und Antworten**

|  |  |
| --- | --- |
| **Client** | **Server** |
| POST  RegisterNewPlayerRequest | Antwort:  PlayerRegisteredResponse  Error 409:  PlayerAlreadyRegistered |
| GET  GameStatus | Antwort:  GameStatusResponse:   * Wait * Turn * NewData |
| GET:  GameInformation | Antwort  GameInformationResponse:[Data] |
| POST:  GameMapData | Antwort:  GameMapDataValid Error 406:  GameMapNotValid |
| POST  GameTurnData | Antwort:  GameTurnDataValid  Error:406  GameTurnDataNotValid |

Implementierung einzelner Routes:

1. RegisterNewPlayerRequest

EntryPoint:



Abbildung 12 REST-Route /RegisterNewPlayerRequest

Wie im Code aus Abbildung 12 gezeigt wird die Route /RegisterNewPlayerRequest über die POST Methode aufgerufen und erwartet sich ein JSON Objekt. Dieses JSON Objekt wird zu einem RegisterNewPlayer Objekt geparsed.

Schlägt dies Fehl bekommt der User der die Anfrage gesendet hat einen HTTP 400 Fehler.

Ist die Umwandlung jedoch erfolgreich wird geprüft ob es ein Objekt der Klasse RunGame.java gibt.

* Die Klasse RunGame.java sammelt alle Informationen der User und speichert sie. RegisterNewPlayer etwa wird in einer Liste dieses Typs gespeichert.

Gibt es kein Objekt dieser Klasse wird eines instanziiert und ein neuer Spieler registriert.

* Die Klasse ServerController hat ein Objekt der Klasse RunGame (run genannt) als Attribut.

Gibt es bereits ein Objekt der Klasse RunGame, so gibt es auch mindestens einen registrierten Spieler, da der Konstruktor der Klasse RunGame einen Spieler zum Registrieren verlangt.

Es wird nun geprüft ob bereits zwei Spieler registriert wurden. Falls sich ein dritter Spieler registrieren möchte bekommt dieser einen HTTP 409 Fehler.

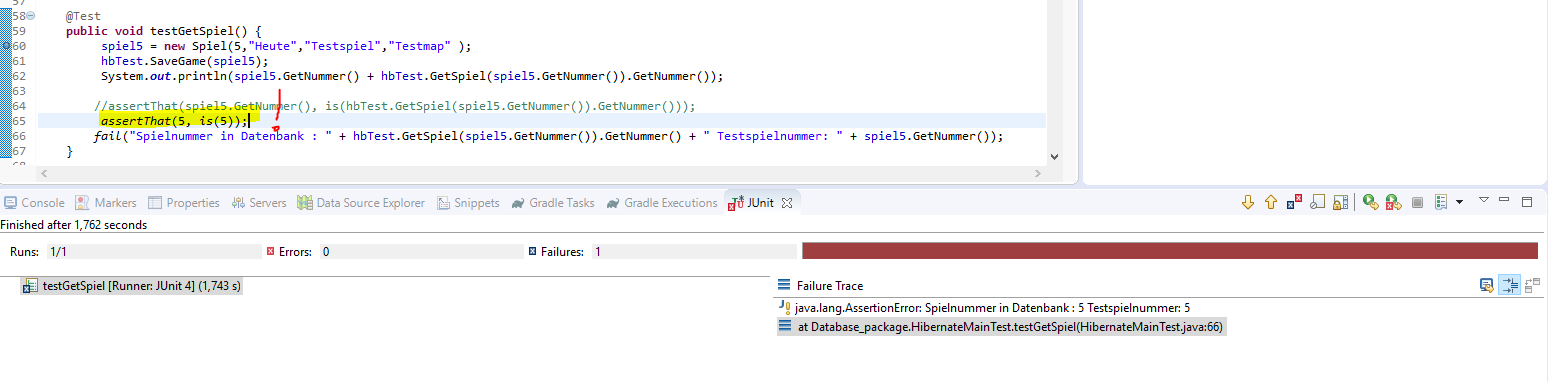
Ebenso wird ein HTTP 409 Fehler ausgegeben falls die ClientIP des ersten registrierten Spielers die gleiche des sich gerade registrierendes Spielers ist.

Sind alle Bedingungen erfüllt wird ein neuer Spieler registriert.

#### Unit Tests

Datenbank

Ich habe eine Reihe von Unit-Tests für meine HibernateMain-Klasse geschrieben, aber nachdem dieser Test versagt:



assertThat(5, is(5)) gibt mir hier einen Fehler aus. Denn 5 ist ja bekanntlich ungleich 5…

Hält man mit der Maus drüber so sieht man, dass beides Integer-Werte sind.

Nichtsdestotrotz die Unit-Tests der Klasse HibernateMain:

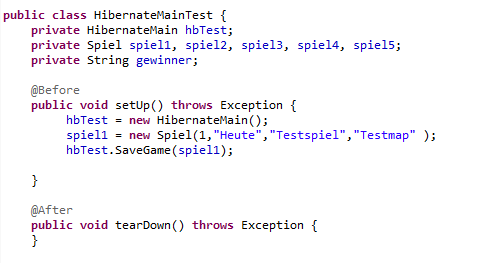


Abbildung 13 Unit-Test vorbereitung

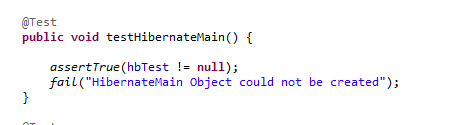


Abbildung 14 1. Unit-Test

Dieser Test prüft ob die Erstellung eines HibernateMain-Objektes erfolgreich war.

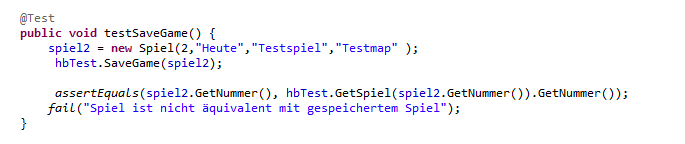


Abbildung 15 2. Unit-Test

Dieser Test prüft ob das gespeicherte Spiel das gleiche ist wie das definierte Testspiel. Da Objekte nicht direkt miteinander verglichen werden konnten (andere ObjectID) werden die Nummern mitenander verglichen, denn jede Nummer kann nur einmal existieren in der Datenbank (Primary-Key constraint).

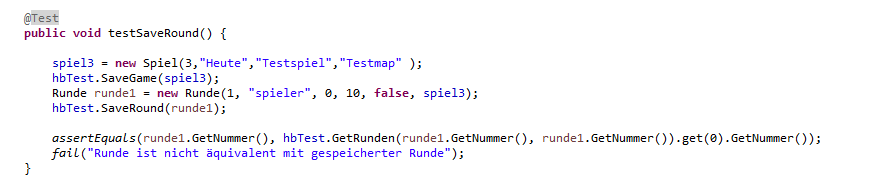


Abbildung 16 3. Unit-Test

Dieser Test prüft ob die gespeicherte Runde die selbe ist wie die eben erstellte. Es werden wieder die Nummern verglichen und ein neues Objekt Spiel zum speichern dieser Runde verwendet.

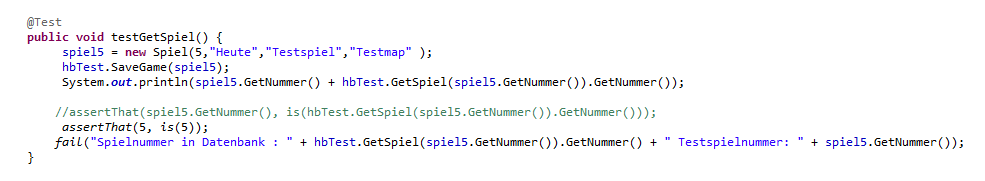


Abbildung 17 4. Unit-Test

Dieser Test prüft ob ein gespeichertes Spiel das gleiche ist wie das Spiel das eben erstellt wurde.

Hier tritt der assertThat(5, is(5)) Fehler auf. Anscheinend ist es egal was hier steht es kommt einfach immer ein Fehler. Ich weiß nicht wieso.

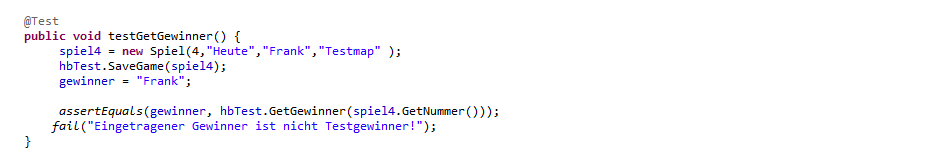


Abbildung 18 5. Unit-Test

Dieser Test prüft ob ein zuvor erstellter Gewinner der gleiche Gewinner ist wie in einem gespeicherten Spiel.

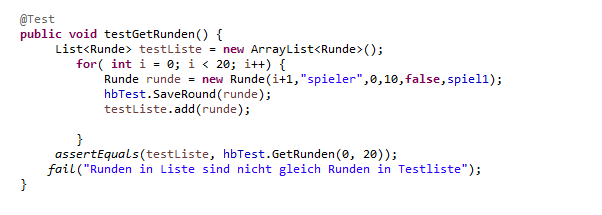


Abbildung 19 6. Unit-Test

Dieser Test prüft ob die in einer Liste eingefügten Runden die gleichen bleiben wie die in der Datenbank eingefügten.

### Fehlende Teile dieser Abgabe:

* Netzwerkkommunikation ist nicht vollständig implementiert
* Unit-Test Anzahl ist nicht vollständig (6 Unit-Tests aufgrund assert Fehler)

Meiner Meinung nach komplette Teile:

* Datenbank und Datenbankzugriff inklusive verlangter Beispielabfragen
* Get Route um Spieler zu registrieren inklusive überprüfung auf Fehler und Einhaltung der mit dieser Route verknüpften Business Rules.