Project Kantine: Deel 2

Een simulatie met database

Instituut voor Communicatie, Media & IT

Hanzehogeschool Groningen
University of Applied Sciences

2018 – 2019

Inhoudsopgave

Inleiding

In het tweede deel ga je verder met het uitwerken van de simulatie en komen concepten als overerving en exceptions weer voorbij. De laatste twee weken staan in het teken van het werken met data, in het bijzonder een toepassing van Object Relational Mapping (ORM).

Regels

Voor deel 2 gelden dezelfde regels als voor deel 1:

- De deadline voor het inleveren van de uitwerking is steeds 18:00 op de eerste maandag volgend op de week waar de opgaven bij horen. Je levert dus bijvoorbeeld je uitwerking van week 1 uiterlijk maandag in week 2 in. De docent zal vragen om een korte demonstratie en uitleg over je uitwerking.
- Elke week wordt het opgeleverde werk beoordeeld met een uitmuntend, goed, voldoende of onvoldoende (10,8,6,4). Het uiteindelijke cijfer zal een gemiddelde zijn van deze beoordelingen.
- Indien een week met onvoldoende beoordeeld wordt is er de volgende week een mogelijkheid tot herkansing.
- Het is niet erg als je de opgaven verdeelt, maar je wordt wel geacht alle uitwerkingen te kunnen toelichten. Een excuus als „Dat heb ik niet gemaakt, dus dat kan ik niet uitleggen" wordt dan ook gezien als het niet hebben gemaakt van de opgave.
- Niet aanwezig zijn bij de nabespreking is hetzelfde als het niet hebben gemaakt van de opgave.
- In principe wordt één cijfer aan een groep gegeven, maar de docent zal hier van afwijken als er sprake is van onevenredige werkverdeling of het niet kunnen uitleggen van een uitwerking.

Week 4: Overerving en foutafhandeling

Opgave 1: Betalen doe je zo

In deze opgave ga je de daadwerkelijke betaling van een persoon implementeren. Een betaling kan op twee manieren gebeuren: *contant* of met een *pinpas*. In principe kun je een interface `Betaalwijze` introduceren, maar bij beide betaalwijzen zou je wel het tegoed kunnen opslaan. We gaan ervan uit dat een betaling met een pinpas gebeurt vanaf een rekening waar een kredietlimiet op zit. Kredietlimieten bestaan niet voor een contante betaling, tenminste de bodem van je portemonnee is de kredietlimiet. Kortom, we introduceren een abstracte klasse `Betaalwijze` als volgt:

```
1  public abstract class Betaalwijze {
2
3      protected double saldo;
4
5      /**
6       * Methode om krediet te initialiseren
7       * @param saldo
8       */
9      public void setSaldo(double saldo) {
```

```
10          this.saldo = saldo;
11      }
12
13      /**
14       * Methode om betaling af te handelen
15       *
16       * @param tebetalen
17       * @return Boolean om te kijken of er voldoende saldo is
18       */
19      public abstract boolean betaal(double tebetalen);
20  }
```

Daarnaast zijn er twee concrete subklassen `Contant` en `Pinpas` die deze abstracte superklasse extenden.

```
1  public class Contant extends Betaalwijze {
2      /**
3       * Methode om betaling af te handelen
4       */
5      public boolean betaal(double tebetalen) {
6          // method body omitted
7      }
8  }
```

en

```
1  public class Pinpas extends Betaalwijze {
2
3      private double kredietlimiet;
4
5      /**
6       * Methode om kredietlimiet te zetten
7       * @param kredietlimiet
8       */
9      public void setKredietLimiet(double kredietlimiet) {
10         // method body omitted
11     }
12
13     /**
14      * Methode om betaling af te handelen
```

```
15       */
16      public boolean betaal(double tebetalen) {
17          // method body omitted
18      }
19  }
```

Merk trouwens op dat een `Persoon` een referentie heeft naar een abstracte klasse `Betaalwijze`. Dit is een „heeft-een" relatie, dus er is *geen* overervingsstructuur tussen `Persoon` en `Betaalwijze`. Je zou inderdaad niet kunnen beweren dat een persoon een betaalwijze is, maar wel dat een persoon een betaalwijze heeft.

a. Implementeer deze drie klassen.

b. Stel dat bij de pinpasbetaling zou worden gecommuniceerd met een `Bank` object. Teken een sequentie-diagram waaruit blijkt hoe je de methode `boolean betaal(double tebetalen)` in `Pinpas` implementeert.[1]

c. Waarom is de instantie variabele saldo protected gemaakt? Waarom is dat handig?

d. Maak een private instantie variabele `betaalwijze` in `Persoon`. Maak een getter en een setter. Aanpassen van de constructor is niet nodig.

e. Pas de code in `Kassa` aan zodat bij de betaling naar de betaalwijze van de persoon wordt gevraagd en de methode `betaal(double tebetalen)` wordt aangeroepen. Indien de betaling faalt laat dan een melding zien op het scherm.

f. Pas de code in de methode `rekenAf` in de klasse `Kassa` aan zodat bij de betaling naar de betaalwijze van de persoon wordt gevraagd en de methode `betaal(double tebetalen)` wordt aangeroepen. Indien de betaling faalt laat dan een melding zien op het scherm. De opbrengst mag natuurlijk ook alleen opgehoogd worden als de betaling gelukt is.

Opgave 2: Kortingskaarthouder

De docenten en kantinemedewerkers zijn in de gelukkige omstandigheid dat ze korting krijgen op de aangeschafte artikelen, voor docenten is dat 25%, met een maximale korting van 1.00 Euro per kantinebezoek en kantinemedewerkers krijgen zelf 35% korting, zonder een maximumbedrag. Dit is te realiseren door de onderstaande interface te definiëren:

```
1  public interface KortingskaartHouder {
2
```

[1] Hint: vergeet niet hoe de klasse `Pinpas` een referentie naar een `Bank` object heeft gekregen.

```
3       /**
4        * Methode om kortingspercentage op te vragen
5        */
6       public double geefKortingsPercentage();
7
8       /**
9        * Methode om op te vragen of er maximum per keer aan de korting
             zit
10       */
11      public boolean heeftMaximum();
12
13      /**
14       * Methode om het maximum kortingsbedrag op te vragen
15       */
16      public double geefMaximum();
17  }
```

Doordat je zou kunnen zeggen dat docenten en kantinemedewerkers kortingskaarthouders zijn is het verstandig om de `Docent` en `KantineMedewerker` klasse deze interface te laten implementeren.

a. Pas de `Docent` en `KantineMedewerker` klasse aan zodanig dat ze de interface `KortingskaartHouder` implementeren.

b. Pas de code in de klasse `Kassa` aan zodat er gecheckt wordt of een `Persoon` ook een kortingskaart heeft (in objecttermen kortingskaarthouder is) en daar rekening mee wordt gehouden bij de betaling.[2]

Opgave 3: Theorie over (abstracte) klassen en interfaces

a. Kun je een instantie maken van een *interface* via **`new`**? Leg uit waarom het logisch is dat het wel of niet kan.

b. Herhaal de vorige vraag met *abstracte* klassen.

c. Kan een klasse meerdere *klassen* overerven?

d. Kan een klasse meerdere *interfaces* implementeren?

e. Kan een klasse tegelijk een klasse overerven en interfaces implementeren?

f. Klopt de stelling dat elke methode in een interface abstract is? Licht je antwoord toe.

g. Moet een klasse abstract zijn als minstens één methode abstract is? Licht je antwoord toe.

[2] Hint: gebruik de **`instanceof`** operator en casting.

h. Leg het begrip *polymorfisme* van klassen uit en geef twee voorbeelden (één met abstracte klassen en één met interfaces).

Opgave 4: Een paar doordenkers

a. Kan een klasse abstract zijn als geen enkele methode abstract is in die klasse? Probeer het eens uit. Leg waarom het logisch is dat dit wel of niet kan.

b. Moet een subklasse van een abstracte klasse altijd alle abstracte methodes implementeren? Leg uit waarom het logisch is dat dit wel of niet kan.

c. Als een klasse niet alle methoden van een interface implementeert kun je iets doen om een (compiler)fout te voorkomen. Wat? Waarom is de oplossing logisch?

d. Leg uit waarom het logisch is dat een instantie variabele niet abstract kan zijn.

e. (Uitdaging) Zoek uit wat een *final* methode is. Leg daarna uit waarom het logisch is dat een methode niet tegelijkertijd abstract en final kan zijn.

Opgave 5: `TeWeinigGeldException` maken

In opgave 2 van vorige week heb je een abstracte klasse met twee concrete subklassen gemaakt. De methode `betaal(double tebetalen)` geeft een **`boolean`** terug als indicatie dat de betaling wel of niet is gelukt.

a. Maak een eigen `TeWeinigGeldException`, met drie constructors:

- `TeWeinigGeldException()`
- `TeWeinigGeldException(Exception e)`
- `TeWeinigGeldException(String message)`

b. Verander het return type van de methode `betaal(double tebetalen)` in **`void`** en voeg een **`throws`** declaratie toe waarbij een `TeWeinigGeldException(String message)` wordt gethrowd als er onvoldoende saldo is. Als de betaalmethode een `TeWeinigGeldException` gooit wordt deze gevangen in `KantineSimulatie`. Zorg dat in het catch-blok de naam van diegene die te weinig geld heeft wordt afgedrukt.

c. Pas de code in `Kassa` aan zodat de `TeWeinigGeldException` wordt gevangen.

Week 5: Data persistentie

De data leeft zolang de simulatie loopt maar het zou mooi zijn als we resultaten kunnen opslaan om ze op een later tijdstip weer op te kunnen halen. In deze week ga je een begin maken met het opzetten van een database voor het opslaan van gegevens en zal je zien hoe je Object Relational Mapping (ORM) kan toepassen op jouw project.

Op Blackboard vind je een voorbeeldproject waar gebruik is gemaakt van de Java Persistence API (JPA) voor het „mappen" van objectdata naar een database (ORM) en hoe relaties tussen objecten onderling kunnen worden opgeslagen.[3] Het doel is dit project te gaan verkennen en bestuderen om in de laatste week de technieken die je tegenkomt zelf te gaan toepassen.

Opgave 1: Opzet en configuratie

a. Download het voorbeeldproject **JPAVoorbeeld** van Blackboard en importeer het in jouw IDE.

b. Zorg er voor dat een database server is opgestart en maak een nieuwe database aan voor dit project, bijvoorbeeld met de naam `jpavoorbeeld`.[4]

De configuratie kan je vinden in het bestand`resources/META-INF/persistence.xml` en voor de connectie zijn de volgende regels van belang:

```
1 <!-- The JDBC driver for your database -->
2 <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.
     Driver" />
3 <!-- The JDBC URL to the database instance -->
4 <property name="javax.persistence.jdbc.url" value="jdbc:mysql://
     localhost:3306/javavoorbeeld" />
5 <!-- The database username -->
6 <property name="javax.persistence.jdbc.user" value="username" />
7 <!-- The database password -->
8 <property name="javax.persistence.jdbc.password" value="password" />
```

c. Pas in de configuratie (waar nodig) de volgende velden aan om de verbinding met jouw database te kunnen maken:

[3] De Java Persistence API is een *specificatie*. Dit wil zeggen dat het alleen maar een een aantal regels opstelt die moeten worden *geïmplementeerd*. In het voorbeeld wordt *Hibernate* gebruikt, een project dat deze regels implementeert. Zie https://www.baeldung.com/jpa-hibernate-difference voor een kort overzicht.

[4] Je zal waarschijnlijk nog MySQL geïnstalleerd hebben, maak daar een nieuw schema aan.

- de connectie url
- username en wachtwoord

De configuratie gaat er van uit dat je een MySQL database gebruikt, mocht dit niet het geval zijn dan zal je een andere *driver* moeten instellen voor `javax.persistence.jdbc.driver`.[5]

In het project vind je de klassen `Student`, `StudieInschrijving`, `StudentKaart` en `Telefoon`. `Student` is vergelijkbaar met de klasse `Student` in de kantinesimulatie met het belangrijkste verschil dat de Student in dit geval één of meerdere `Telefoon` en `StudieInschrijving` objecten heeft. Verder kan een Student natuurlijk maar één `Studentkaart` hebben.

Je zal ook zien dat in deze klassen Java *annotaties* worden gebruikt, een techniek om *metadata* aan klassen toe te voegen. Deze annotaties doen op zich niets maar geven extra informatie die kan worden gebruikt in een andere context, bijvoorbeeld door JPA als objecten moeten worden opgeslagen in een database.[6] Zie bijvoorbeeld het volgende fragment van de klasse `Telefoon` voor het gebruik van deze annotaties:

```
1  import javax.persistence.Id;
2  import javax.persistence.Column;
3  import javax.persistence.Entity;
4  import javax.persistence.GeneratedValue;
5  import javax.persistence.GenerationType;
6
7  @Entity
8  public class Telefoon {
9      @Id
10     @GeneratedValue(strategy = GenerationType.IDENTITY)
11     private int id;
12
13     @Column(name = "nummer")
14     private String nummer;
15
16     @Column(name = "type")
17     private String type;
18
19     public Telefoon() {
20         // ...
```

[5] Er zijn veel alternatieve databases waar je misschien al mee hebt geëxperimenteerd en het is waarschijnlijk dat daar ook een JDBC driver voor beschikbaar is. Je zal deze driver als afhankelijkheid dan ook moeten opnemen in de Maven configuratie.

[6] Zie https://docs.oracle.com/javase/tutorial/java/annotations/ voor een korte inleiding in Java annotaties.

```
21        }
22    }
```

d. Lees de klassen `Student`, `StudieInschrijving` en `Telefoon` door. Zou je al iets kunnen vertellen over het doel van de annotaties `@Id`, `@GeneratedValue` en `@Column`?

In de klasse `Main` vind je de methode `runVoorbeeld` waar een aantal objecten worden aangemaakt, opgeslagen en verwijderd. Voer de methode `main` uit (die de methode `runVoorbeeld` aanroept). Controleer vervolgens of gegevens in de database zichtbaar zijn.

e. Een *viertal* tabellen zullen nu in de database zijn aangemaakt, welke zijn deze?

Opgave 2: Relaties

De objecten in dit voorbeeld hebben relaties met elkaar, zo heeft een Student een StudentKaart en één of meerdere StudieInschrijving(en). Je zal onderhand bekend zijn met *One-to-One*, *One-to-Many* etc. relaties in databases en hoe deze relaties in code (met JPA) zijn uit te drukken zal je in deze opgave gaan verkennen.

a. Het attribuut `telefoons` in de klasse `Student` is geannoteerd met `@OneToMany`. Waarom zou hier `@OneToMany` en niet `@OneToOne` zijn gebruikt?

b. Het attribuut `studies` in de klasse `Student` is ook geannoteerd met `@OneToMany`, dit omdat een student natuurlijk bij meerdere studies ingeschreven kan staan. In tegenstelling tot `telefoons` wordt hier geen `@JoinTable` annotatie gebruikt.

   - Hoe wordt het verschil zichtbaar in de database? Let hier op de tabellen die zijn aangemaakt en de velden in de tabellen voor een Student en StudieInschrijving.
   - Zou je een nadeel kunnen bedenken waarom voor de relatie `Student` en `StudieInschrijving` via `studies` *geen* `@JoinTable` is gebruikt?

c. In de klasse `StudieInschrijving` is het attribuut `student` met `@ManyToOne` geannoteerd. Beschrijf hoe deze relatie in de database zichtbaar wordt.

d. De klasse `StudentKaart` is geannoteerd met `@Embeddable` en het attribuut `kaart` in de klasse `Student` met `@Embedded`.

   - Waar vind je een StudentKaart terug in de database?
   - Wat zou je hieruit kunnen afleiden met betrekking tot het gebruik en de functie van `@Embeddable` en `@Embedded`?

Opgave 3: Transacties

De meeste databases kennen het concept van een *transactie* waar meerder queries worden gegroepeerd en als één operatie worden uitgevoerd. Mocht ergens een fout optreden dan kunnen alle queries weer ongedaan worden gemaakt.

Zoek in de klasse `Main` een voorbeeld van het gebruik van een transactie. Je zal zien dat gebruik wordt gemaakt van een *try-catch* blok om een transactie bij een fout af te breken en terug te draaien (*rollback*).

a. Een transactie wordt ook wel een *unit of atomicity* genoemd, waarbij atomiciteit in de context van databases „alles of niets" betekent. Vooruitkijkend op week 6, kan je voor de kantinesimulatie een situatie bedenken waar je dit patroon zou kunnen toepassen?[7]

Opgave 4: Voorbereiding

Nu je hebt gezien hoe je ORM kan gebruiken wordt het tijd om dit aan de simulatie toe te voegen. De opgaven van volgende week gaan over de *implementatie*, nu ga je daar alvast voorbereidingen voor treffen door een configuratie aan te maken en een *entity manager* aan de klasse `KantineSimulatie` toe te voegen.

De volgende stappen zullen het gemakkelijker maken om direct aan de slag te kunnen in een IDE als Eclipse, IntelliJ of zelfs Visual Studio Code.

a. Maak een nieuwe database aan voor de simulatie, bijvoorbeeld met de naam *jpatest*.

b. Kopiëer `pom.xml` van het voorbeeldproject naar jouw project. Dit bestand is een configuratie voor *Maven*, een hulpmiddel voor Java-projecten voor het beheren van afhankelijkheden (bijvoorbeeld noodzakelijke JPA gerelateerde modules) en hoe het project gecompileerd moet worden [8].

c. Maak het bestand `resources/META-INF/persistence.xml` aan en configureer de verbinding met de database.

d. Geef de in de configuratie '`<persistence-unit/>` in plaats van *JPAVoorbeeld* een andere (meer relevante) naam, bijvoorbeeld *KantineSimulatie*.

Als laatste ga je een entity manager aan de simulatie toevoegen. In het voorbeeldproject zie je in `Main` het volgende:

[7] Denk aan waar je exceptions hebt toegepast.

[8] Zie ook Maven in 5 Minutes

```java
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;

public class Main {
    private static final EntityManagerFactory ENTITY_MANAGER_FACTORY =
        Persistence.createEntityManagerFactory("JPAVoorbeeld");
    private EntityManager manager;

    public void runVoorbeeld() {
        manager = ENTITY_MANAGER_FACTORY.createEntityManager();

        // transactions omitted

        manager.close();
        ENTITY_MANAGER_FACTORY.close();
    }
}
```

e. Voeg zoals in het bovenstaande fragment een entity manager toe aan de klasse `KantineSimulatie` en vervang *JPAVoorbeeld* door de naam die je eerder hebt gekozen voor de configuratie.

Opgave 5: Bijzondere prijzen

Voor we verder kunnen is nog één bijzondere situatie waar rekening mee moet worden gehouden want bij de kassa hangt namelijk de volgende waarschuwing:

> **Op dagaanbiedingen is geen medewerkerskorting van toepassing.**

De kantine heeft blijkbaar dagaanbiedingen waarbij één of meerdere artikelen in prijs zijn verlaagd. We moeten dus gaan letten op welke artikelen wél en op welke géén korting van toepassing is bij afrekenen naast de algemene korting op basis van een kortingkaart!

a. Pas de klasse `Artikel` aan met een `korting` veld en maak de bijbehorende *getter* en *setter*. `korting` is een concreet bedrag en *niet* een percentage.

b. Maak een constructor aan die naast naam en prijs ook korting als parameter heeft. Houdt er verder rekening mee dat korting een standaard waarde van 0 euro heeft indien geen korting wordt doorgegeven als constructor parameter.

c. Zorg in de simulatie bij het aanmaken van de artikelen dat dagelijks tenminste één artikel in de aanbieding is – je zal hier dus weer een randomisatie moeten toepassen. De korting op een dagaanbieding is 20% van de prijs van het geselecteerde artikel.

d. De verantwoordelijkheid voor het berekenen van het totaalbedrag heb je eerder al verplaatst naar de klasse `Kassa` in de methode `rekenAf`. Pas deze methode aan zodat:

   - de korting op dagaanbiedingen wordt toegepast
   - de korting voor kortingkaarthouders wordt toegepast, uitgezonderd op dagaanbiedingen

Week 6: Bonnetjes en optellingen

In deze laatste week ga je de simulatie afronden en ga je gebruik maken van een database om data in de simulatie op te slaan. Je zal hier gebruik maken van JPA/Hibernate waar je in de vorige week al kennis mee hebt gemaakt.

Opgave 1: De klasse `Factuur`

In de simulatie heb je tot nu toe gewerkt met totalen, bijvoorbeeld de omzet die in een week is gegenereerd. Om een begin te maken met het bijhouden van individuele aankopen gaan we facturen bijhouden, oftewel het aanmaken van een *kassabon.

Op een factuur komt natuurlijk ook een datum en we zouden onze `Datum` klasse hier voor kunnen gebruiken, maar zullen dan tegen een probleem aanlopen als we deze waarde in de database willen opslaan. We hebben namelijk nergens aangegeven met welk database type ons objecttype `Datum` correspondeert en Hibernate zal het daarom opslaan als een `bytea` (byte array) type waar we verder niet veel mee kunnen doen (bijvoorbeeld queries op toepassen). We gaan om deze reden de klasse `Datum` vaarwel zeggen en het standaardtype `java.time.LocalDate` gebruiken dat Hibernate wél kent en kan *serialiseren* naar het database type `DATE`.

Voor het eenvoudig aanmaken van een `LocalDate` object kan je de volgende constructie gebruiken:[9]

```
1 LocalDate datum = LocalDate.to(2019, 5, 16);
```

De logica voor de afhandeling van een bestelling vindt plaats in de klasse `Kassa` in de methode `rekenAf`. Deze methode gaat nu een Factuur aanmaken voor elke klant. De volgende begincode is gegeven voor de nieuwe klasse `Factuur`:

[9] Mocht je meer precisie in jouw simulatie nodig hebben dan kan je ook het type `java.time.LocalDateTime` gebruiken om ook uur, minuten en seconden vast te leggen. `LocalDateTime` zal corresponderen met het `DATETIME` database type.

```java
import java.time.LocalDate;
import java.io.Serializable;

public class Factuur implements Serializable {

    private Long id;

    private LocalDate datum;

    private double korting;

    private double totaal;

    public Factuur() {
        totaal = 0;
        korting = 0;
    }

    public Factuur(Dienblad klant, LocalDate datum) {
        this();
        this.datum = datum;

        verwerkBestelling(klant);
    }

    /**
     * Verwerk artikelen en pas kortingen toe.
     *
     * Zet het totaal te betalen bedrag en het
     * totaal aan ontvangen kortingen.
     *
     * @param klant
     */
    private void verwerkBestelling(Dienblad klant) {
        // method body omitted
    }

    /*
     * @return het totaalbedrag
     */
    public double getTotaal() {
```

```
42          return totaal;
43      }
44
45      /**
46       * @return de toegepaste korting
47       */
48      public double getKorting() {
49          return korting;
50      }
51
52      /**
53       * @return een printbaar bonnetje
54       */
55      public String toString() {
56          // method body omitted
57      }
58  }
```

a. Verplaats de berekening van het totaal en de toegepaste korting in `rekenAf` naar de methode `verwerkBestelling` in de klasse `Factuur`. Let op: betalingen blijven door de Kassa uitgevoerd worden, dit is niet de verantwoordelijkheid van een Factuur.

b. Maak voor elke afrekening nu een Factuur object aan en gebruik de getters van `Factuur` om het totaalbedrag en de toegepaste korting van de Kassa te verhogen.

De klasse `Factuur` is nog een reguliere klasse en JPA/Hybernate weet niet dat het een entiteit is die zij moet gaan beheren:

c. *Annoteer* de klasse `Factuur` zodat deze door een entitymanager beheerd kan worden. In *JPA-Voorbeeld* kan je geschikte voorbeelden vinden, bedenk dat je nog *niet* met complexe (many-to-one, etc.) relaties rekening hoeft te houden.

d. Implementeer `toString` zodat je in de simulatie ook een bonnetje kan printen.

Opgave 2: Facturen opslaan

Het voorbereidend werk nu is gedaan en ondanks de toevoeging van een Factuur bij de afrekening zal de simulatie nog steeds werken. De refactoring was nodig want in deze opgave gaan we de factuur daadwerkelijk opslaan in een database.

Je hebt ter voorbereiding in de klasse `KantineSimulatie` een `EntityManager` object aangemaakt. Zoals je in *JPAVoorbeeld* hebt kunnen zien wordt deze gebuikt voor beheren en uitvoeren van data-

basetransacties. De klasse `Kassa` heeft hier nog geen weet van en we moeten dit manager object nu gaan doorgeven.

a. Voeg aan zowel de klasse `Kantine` als de klasse `Kassa` een veld `manager` toe van het type `javax.persistence.EntityManager`

b. Pas de constructors van `Kantine` en `Kassa` aan zodat het een EntityManager object als parameter accepteert en het veld `manager` in beide klassen wordt geïnitialiseerd met deze waarde.

c. Zorg bij het aanmaken van een Kassa in de klasse `Kantine` dat het `manager` object wordt doorgegeven.

De klasse `Kassa` heeft nu een EntityManager en deze kan worden gebruikt voor het opzetten van een databasetransactie om een factuur op te slaan. Natuurlijk willen we een factuur pas opslaan als de betaling succesvol is verlopen, indien dit niet slaagt zal de transactie ongedaan moeten worden gemaakt.

d. Gebruik de EntityManager om een factuur op te slaan door middel van een transactie in de methode `rekenAf` in `Kassa`. Breek de transactie af als de betaling niet is geslaagd (een *rollback*). Maak hier gebruik van de `TeWeinigGeldException` om te bepalen of een betaling wel of niet is geslaagd.

Opgave 3: Totalen en gemiddelden

Nu data in de database aanwezig is kunnen ook queries worden opgesteld en uitgevoerd in `KantineSimulatie` zodra de simulatie heeft plaatsgevonden. Wederom vind je in het *JPAVoorbeeld* project hier voorbeelden van.

a. Maak een query om de totale omzet en toegepaste korting op te vragen, voer deze uit en print het resultaat.

b. Maak een query om de gemiddelde omzet en toegepaste korting per factuur op te vragen, voer deze uit en print het resultaat.

c. Maak een query om de top 3 van facturen met de hoogste omzet op te vragen, voer deze uit en print het resultaat.

Opgave 4: `Factuur` specificatie

Totalen zijn interessant, maar we weten nog niet heel veel van individuele aankopen, op termijn zou het wellicht interessant zijn om te zien welke producten een klant vaak samen koopt, of als een pro-

duct niet aanwezig is welk product als alternatief wordt gekozen. We zouden met andere worden kunnen kijken wat complementaire- en/of substitutieproducten zijn.[10]

In deze opgave ga je artikelen aan de factuur toevoegen en elk artikel zal worden bewaard in een `FactuurRegel` instantie. De code voor deze klasse wordt gegeven:

```
1  public class FactuurRegel implements Serializable {
2
3      private Long id;
4
5      private Factuur factuur;
6
7      private Artikel artikel;
8
9      public FactuurRegel() {}
10
11     public FactuurRegel(Factuur factuur, Artikel artikel) {
12         this.factuur = factuur;
13         this.artikel = artikel;
14     }
15
16     /**
17      * @return een printbare factuurregel
18      */
19     public String toString() {
20         // method body omitted
21     }
22 }
```

a. *Annoteer* de klasse `FactuurRegel` zodat deze door een entitymanager beheerd kan worden. Het veld `factuur` verwijst naar de factuur waar een FactuurRegel toe behoort – wat voor een relatie is dit en welke annotatie zal je hiervoor moeten gebruiken?

b. Voeg aan de klasse `Factuur` een veld `regels` toe met type `ArrayList<FactuurRegel>`. Dit veld moet ook geannoteerd worden – wat voor een relatie is dit en welke annotatie kan je hiervoor gebruiken?

c. Pas `verwerkBestelling` in `Factuur` aan zodat per Artikel een FactuurRegel instantie aan `regels` factuur wordt toegevoegd.

[10] In bijvoorbeeld e-commerce is dit een belangrijk onderscheid en vormt een basis voor *recommender systems*, het op maat kunnen genereren van aanbevelingen voor klanten op basis van historische data.

d. Pas `toString` in `Factuur` aan zodat ook regels met artikelen op het bonnetje worden geprint. Gebruik hier `toString` in `FactuurRegel` voor.

e. Welke redenen kan je bedenken om Artikelen in een aparte klasse `FactuurRegel` op te slaan en niet direct aan `Factuur` toe te voegen?

Opgave 5: Populaire artikelen

Nu verkochte artikelen ook worden opgeslagen kunnen we additionele queries uitvoeren op op artikelniveau queries op te stellen en uit te voeren, wederom in `KantineSimulatie` aan het einde van de simulatie.

a. Maak een query om de totalen en toegepaste korting per artikel op te vragen, voer deze uit en print het resultaat.

b. Maak een query om de totalen en toegepaste korting per artikel, *per dag* op te vragen. Voer deze uit en print het resultaat.

c. Maak een query om de top 3 van meest populaire artikelen op te vragen, voer deze uit en print het resultaat.

d. Maak een query om de top 3 van artikelen met de hoogste omzet op te vragen, voer deze uit en print het resultaat.

Opgave 6: Bonus

De simulatie is nu ten einde, maar nu je kennis hebt gemaakt met data persistentie in de vorm van ORM zijn er wellicht extra problemen waar je misschien nog aan wilt werken, bijvoorbeeld:

- Artikelen creëer je nu in de simulatie, inclusief voorraadbeheer. Hoe zou je artikelen en voorraad per artikel in een database kunnen vormgeven?
- Bewaar personen in de database en gebruik het BSN nummer als primaire sleutel. Maak een klasse BonusKaart en koppel deze aan een Persoon en bedenk hoe je deze kan gebruiken voor extra kortingen.
- Een Kassa heeft contant geld in de lade. Breid contante betalingen uit met wisselgeld en los situaties op waar een tekort aan wisselgeld kan ontstaan.
- Bedenk zelf een uitbreiding.