Übungsblatt 4

Hinweise

- **Abgabe:** Bitte geben Sie Ihre Lösungen zu dieser Aufgabe digital bis zum 28.11.2021, 23:00 Uhr im Ilias ab. Spätere Abgaben werden nicht akzeptiert. Exportieren Sie Ihr Java-Projekt in *eine* zip-Datei. Benennen Sie Ihre zip-Datei **blatt4_nachname_matnr.zip**.
- **Einzelarbeit:** Die Aufgaben zu diesem Übungsblatt sind ausschließlich in Einzelarbeit zu lösen und abzugeben; es werden keine Gruppenarbeiten akzeptiert. Bei gleichen Lösungen werden *beide* Lösungen mit 0 Punkten bewertet.
- Sie benötigen mindestens 10 Punkte, um das Übungsblatt zu bestehen. Wir empfehlen aber dringend, auch Aufgabe 1c) zu bearbeiten; sie hilft ungemein dabei, Fehler im Code für die vorherigen Aufgaben zu finden. Über alle vier Übungsblätter hinweg müssen Sie mindestens 60 Punkte erreichen, d.h. im Schnitt 15 pro Blatt.

Einleitung

Da Sie in diesem Übungsblatt Administratorrechte auf eine Datenbank benötigen, können Sie leider den *weathertop* Server nicht nutzen. Installieren Sie deshalb *PostgreSQL* in Version 13 lokal auf Ihrem Rechner. Legen Sie in PostgreSQL einen Administratornutzer namens 'admin' an, der das Passwort 'admin' hat. Erstellen Sie zudem mit diesem Nutzer eine Datenbank mit dem Namen 'admin' – PostgreSQL verbindet Ihren Nutzer dann standardmäßig mit dieser Datenbank wenn Sie keine andere angeben.

Laden Sie das beigefügte Java-Projekt namens *views-and-triggers* in eine IDE Ihrer Wahl. Installieren Sie die Abhängigkeiten aus der Datei *pom.xml* mit *Maven*[1]. Installieren Sie als nächstes Checkstyle[2] via Maven oder installieren Sie ein Checkstyle Plug-In für Ihre IDE[3]. Nutzen Sie für Checkstyle die im Projekt enthaltene Konfigurationsdatei *google_checks.xml*. Tipp: Sie können in Ihrer IDE u.U. die Checkstyle-Regeln als Einstellungen für den Codestil importieren[4].

In den folgenden Aufgaben vervollständigen Sie das *views-and-triggers* Java-Projekt mit eigenem Code. Sie dürfen dabei keine Pakete nutzen, die nicht bereits in der Datei *pom.xml* angegeben sind. Ihr Code – also alles unter dem Verzeichnis *src* – muss kompilierbar sein. Der Code muss zudem konform zu den gegebenen Checkstyle Regeln sein. Für die Abgabe exportieren Sie ihr Projekt als zip-Datei, die genau dieselbe Ordnerstruktur hat, wie die zip-Datei die sie für dieses Übungsblatt heruntergeladen haben. Fügen Sie außer Java- oder SQL-Quellcode keine Dateien hinzu. D.h. keine Libraries, Jar-Dateien, class-Dateien, Bilddateien, etc.

[1] `https://maven.apache.org/`
[2] `https://checkstyle.sourceforge.io/`
[3] z.B. `https://plugins.jetbrains.com/plugin/1065-checkstyle-idea`
[4] `http://biercoff.com/how-to-import-checkstyle-rules-into-intellij-idea-code-format-rules/`

Aufgabe 1 — 16 + 4 + 7 = 27 Punkte

Entwickeln Sie eine Java API um die Datenbank eines Webshops zu verwalten. Alle Datenbankoperationen die der Shop benötigt sollen als Methodenaufrufe von Klassen gekapselt werden. Dadurch müssen sich Nutzer der API nicht mit Details von JDBC, wie `Connection` oder `PreparedStatement`, auskennen.

Als Grundlage für die API nutzen Sie die abstrakte Klasse `ShopOperations`. Diese Klasse stellt für einen bestimmten Datenbanknutzer eine Verbindung zu einer PostgreSQL Datenbank her. Da sie das `AutoCloseable` Interface implementiert, kann man sie wie eine `java.sql.Connection` in einem `try (...)` Block nutzen. Zudem bietet die Klasse eine Funktion `prepareStatement`, die eine SQL-Query aus einer Datei einlesen und als `PreparedStatement` ausgeben kann. Vollziehen Sie nach, woher die Funktion `prepareStatement` die Dateien liest. Finden Sie zudem heraus, was bei der Funktion `prepareSelfClosingStatement` anders ist.

> Nutzen Sie die Methoden `prepareStatement` und `prepareSelfClosingStatement` für alle SQL Queries die Sie für den folgenden Code benötigen! Für eine gute Wartbarkeit sollen in Ihrem Code keine Queries als Strings auftauchen. Stattdessen legen Sie alle Queries als Textdateien ab und greifen mit den beiden Methoden darauf zu. Die einzige Ausnahme sind Queries, bei denen Sie Nutzernamen einfügen müssen – das funktioniert nicht mit `PreparedStatement`.

> Tipp: Bearbeiten Sie die Aufgaben a), b) und c) parallel, indem Sie sich im Unit-Test `ShopOperationsTest` stückweise vorarbeiten!

a) API für Administratoren — 16 Punkte

Schreiben Sie eine Klasse, mit der Administratoren den Webshop verwalten können. Die Klasse soll `AdminShopOperationsImpl` heißen und von der abstrakten Klasse `AdminShopOperations` erben. Implementieren Sie die Klasse wie folgt:

- `createShopDatabase()`: Legt eine neue Datenbank mit Namen *shop* an. Falls eine Datenbank mit diesem Namen bereits vorhanden ist wird diese zuerst gelöscht. *(1 Punkt)*
- `createTables()`: Legt die folgenden Relationen in der momentan verbundenen Datenbank an:
  - CUSTOMER(NAME, BALANCE) mit geeigneten Typen für die Daten aus *customer.data*
  - ARTICLE(ARTICLE, PRICE) mit geeigneten Typen für die Daten aus *article.data*
  - PURCHASE(ID, CUSTOMER, DATE, ARTICLE, QUANTITY) mit geeigneten Typen für die Daten aus *purchase.data* und mit zwei Fremdschlüsselbeziehungen

    CUSTOMER → CUSTOMER(NAME),
    ARTICLE → ARTICLES(ARTICLE).

  (1 Punkt)
- `populateTables()`: Befüllt die drei Relationen CUSTOMER, ARTICLE und PURCHASE mit den Daten *customer.data*, *article.data* und *purchase.data* aus dem Verzeichnis *src/main/resources*. Nutzt dafür die statische Klasse `ShopResource` und die Klasse `org.postgresql.copy.CopyManager`. *(1 Punkt)*

- `createUsers()`: Legt für jeden Kundennamen aus der Tabelle CUSTOMER einen gleichnamigen Account in der Datenbank an. Setzt als initiales Passwort ebenfalls den Kundennamen. Stellt sicher, dass die Kunden keine Rechte haben, um auf die Relationen CUSTOMER, ARTICLE und PURCHASE zuzugreifen. Falls bereits Nutzer mit gleichen Namen in der Datenbank vorhanden sind, werden diese vorher gelöscht. *(2 Punkte)*

- `createViewHistory()`: Erstellt ein View HISTORY, das für den anfragenden Kunden angibt, wann er welche Artikel in welcher Menge, zu welchem Gesamtpreis und unter welcher Vorgangsnummer gekauft hat. Die View ist nach der Vorgangsnummer absteigend sortiert. Erteilt jedem Kunden Leserechte, aber keine weiteren Rechte, auf das neue View.

  Beispiel: *emilie* hat in ihrer Bestellung am 09.02.2021 sechs *Toner_159* zum Einzelpreis von 47 Euro bestellt. Wenn *emilie* sich an der Datenbank anmeldet und die Anfrage `SELECT * FROM history` ausführt, erhält sie als Ergebnis:

id	date	article	quantity	price
11	2021-09-02	Toner_159	6	282

  (1,5 Punkte)

- `createFunctionNewPurchase()`: Erstellt eine Funktion `new_purchase(text, INTEGER)RETURNS boolean` in der Datenbank – d.h., *nicht* in Java! –, die für einen Kunden *true* zurückliefert, falls sein Guthaben ausreicht, um eine bestimmte Ware bei gegebener Stückzahl zu bezahlen. In diesem Fall soll die Funktion die Bestellung für den Kunden, das heutige Datum, sowie eine neue, noch nicht vergebene ID in die Relation PURCHASE eintragen. Außerdem soll die Funktion das Guthaben des Kunden in der Relation CUSTOMER um den entsprechenden Betrag verringern. Der Kunde ist der Datenbanknutzer, der die Methode aufruft. *(4 Punkte)*

- `createRuleDeleteHistory()`: Erteilt jedem Nutzer das Recht, aus der View HISTORY zu löschen. Erstellt eine *Rule* in der Datenbank, die bei Löschoperationen durch Nutzer auf der View HISTORY die Einträge in den Relationen PURCHASE und CUSTOMER anpasst. D.h. es müssen die zugehörigen Bestellungen aus PURCHASE entfernt werden und der Preis der Bestellungen dem Guthaben des Nutzers wieder gutgeschrieben werden. Da die Ware zügig versendet wird, dürfen nur Einträge entfernt werden, die am gleichen Tag erstellt wurden, an dem die Rule ausgeführt wird. *(3 Punkte)*

- `int getBalance(String ofUser)`: Gibt das Guthaben des Nutzers `ofUser` aus. *(1 Punkt)*

- `ResultSet selectCustomerName()`: Liefert alle Namen von Kunden des Webshops aus der Relation CUSTOMER. *(0,5 Punkte)*

- `ResultSet selectArticleName()`: Liefert alle Namen von Artikeln des Webshops aus der Relation ARTICLE. *(0,5 Punkte)*

- `ResultSet selectPurchaseId()`: Liefert alle Id's von Einkäufen im Webshop aus der Relation PURCHASE. *(0,5 Punkte)*

b) API für Kunden — 4 Punkte

Schreiben Sie zudem eine Klasse, mit der Kunden den Webshop nutzen können. Die Klasse soll `UserShopOperationsImpl` heißen und und von der abstrakten Klasse `UserShopOperations` erben. Implementieren Sie `UserShopOperationsImpl` wie folgt:

- `boolean newPurchase(String article, int quantity)`: Kauft für den Kunden den Artikel `article` in der Stückzahl `quantity`. Ruft dafür die von den Administratoren erstellte Datenbankmethode `new_purchase` auf und liefert deren Rückgabewert, siehe oben. *(2 Punkte)*
- `cancelPurchase(String article)`: Storniert für den Kunden alle Käufe des Artikels `article` des heutigen Tages. Führt dafür ein `DELETE` Statement auf der View HISTORY durch, siehe oben. *(1 Punkt)*
- `ResultSet selectHistory()`: Liefert alle Käufe des Kunden aus der View HISTORY. *(0,5 Punkte)*
- `ResultSet selectHistoryToday()`: Liefert alle heutigen Käufe des Kunden aus der View HISTORY. *(0,5 Punkte)*

c) Unit Test **7 Punkte**

Testen Sie Ihre API mit dem Unit Test `ShopOperationsTest.testScenario()`. Dieser Test erstellt mit der Administrator API schrittweise den Webshop. Nach jedem Schritt soll getestet werden, ob die hinzugefügte Funktionalität korrekt durch die Kunden API nutzbar ist. Nutzen Sie für Ihre Tests die Methoden der Klasse `org.junit.jupiter.api.Assertions`. Implementieren Sie damit die folgenden Methoden:

- `assertPaulHasNoAccess()`: Testet ob wirklich nur Administratoren die Berechtigungen für die Relationen CUSTOMER, ARTICLE und PURCHASE besitzen. Verbindet sich dafür als Nutzer 'paul' mit der Administrator API und versucht darüber die Relationen zu lesen. *(1,5 Punkte)*
- `assertEmilieSeesHistory()`: Testet ob Kunden das View HISTORY abrufen können. Verbindet sich dafür als Kundin 'emilie' über die Kunden API mit dem Webshop und ruft ihre Kaufhistorie ab. Prüft, ob die Historie korrekt ist. *(1,5 Punkte)*
- `assertEmilieCanPurchaseToner()`: Testet ob Nutzer einkaufen können. Verbindet sich dafür als Kundin 'emilie' über die Kunden API mit dem Webshop und versucht zuerst 10 Stück 'Toner_216' und danach 2 Stück 'Toner_159' zu kaufen. Prüft ob die Rückgabewerte der Methode korrekt sind und ob die gekauften Artikel in der heutigen Kaufhistorie der Kundin auftauchen. *(2 Punkte)*
- `assertEmilieCanCancelPurchase(AdminShopOperations adminOp)`: Testet ob Nutzer Einkäufe stornieren können. Verbindet sich dafür als Kundin 'emilie' über die Kunden API mit dem Webshop und versucht alle Einkäufe von 'Toner_216' zu stornieren. Prüft ob der Rückgabewert der Methode korrekt ist und ob die Änderungen korrekt in die Tabelle PURCHASE und CUSTOMER übertragen werden. Nutzt für letzteres die Administrator API `adminOp`. *(2 Punkte)*

Sie können die Soll-Ergebnisse ihrer einzelnen Tests händisch aus den Daten in *src/main/resources* ermitteln und dann im Code verwenden.