CSCB07F17 Assignment [100 Marks]

Logistics

Due Date: Monday November 6th 2017

Due Time: 10:00 am

NO LATE SUBMISSIONS WILL BE ACCEPTED

GROUP SIZE: Individual

BLACKOUT START: Sunday November 5th 2017 10:00 pm

Preface

The purpose of the assignment is to tie together the skills we learned in the exercises, and build our first full Java application. The development done here will directly impact your final group project. Your group project will build upon the code from the assignment. As marking may not be completed at the time of the first phase of your project is completed, it is important to adhere fully to all interface agreements given in the assignment, so that you can swap out the code from your best team member in the project phase.

In the assignment, we will be building a simple inventory and sales tracking application. This will allow for simple behaviours such as adding to inventory, selling items, adding new types of items, checking sales volumes, and adjusting prices of products.

You will be required to store information to a SQLite database. As this course is not about database design, and does not expect you to know SQL, most of this has been handled in the starter code.

Section 1: svn check out and check in starter code [0 marks]

Now that we have gotten fairly comfortable with svn, the following part of the Assignment will be worth very few marks, but is still important to complete:

1. Check out your repository from MarkUs (https://markus.utsc.utoronto.ca/svn/cscb07f17/[UTORID]_repo || /group_NNNN/Assignment)

   OR
2. Update your repository from exercise3 – we will be using the same repositories, just in a new folder
3. Download the starter code from Piazza
4. Un-tar the starter code using the command tar –zxvf assignmentStarterCode.tar.gz
5. Using either the .svnignore file, or by manually adding only the folders and files you want, ensure that your checkin does not include anything except the files contained in the src directory
6. Complete the commit of these files

At this point you should have a working copy of your repository with the starter code.

[Marking: 0 marks for checking out repository, -5 marks for including unnecessary files]

Section 2: UML Use Case Diagram [5 Marks]

In lecture, we discussed Use case diagrams. Please take the time to draw up a simple use case diagram, based on this assignment. Don't forget, Use Case diagrams are about what interactions different users can have with the system (such as an employee, an administrator, and a customer). By doing this diagram, you should grow a better understanding of the inner workings of the system you are about to build. In order to get a better understanding of what the application can do, read through the detailed comments in SalesApplication.java.

The diagram should be submitted as either a .PNG or .JPG file, and please ensure that the final file size is under 5 Mb. There are lots of tools out there that can be used to help draw these diagrams on the computer (such as Dia, Visio, etc). You can also simply draw by hand, and either scan or take a clear picture of it, and upload that.

Section 2: Implementation [80 Marks]

Please read through this section at least once fully before starting. There are several nuances that you may miss if you jump right into coding.

Some other things to note: When implementing, try and place methods in logical packages, for ease of organization. The class snippets below will list only public methods, and will not list any helpers that you can use. Please follow SOLID design when building out the code for this project. The given UML will help guide you on the minimum necessary to be successful, but it is only a guideline. At minimum you must implement all methods given in the UML diagrams. You may create your own Exceptions, as well as any other classes and interfaces that you feel will improve the design of your code!

Basics [10 Marks]

Before we get going on the fun stuff, we must first implement a whole slew of basic classes. Let's first implement classes to represent the key objects in our data model:

<<abstract>> User
- id : int - name : String - age : int - address : String - roleId : int - authenticated : boolean
+ getId() : int + setId(id : int) : void + getName() : String + setName(name : String) : void + getAge() : int + setAge(age : int) : void + getRoleId() : int

+ {authenticate(password : String) : boolean}

*NOTE: Methods inside braces ({ }) are considered final

In the above, we have a method called authenticate that must be implemented. This method takes in a password in plain text, and validates if it matches the password found in the database for the given user. There is a method in com.b07.security.PasswordHelpers called comparePassword that takes in a string of a hashed password from the database, and an unhashed password and returns true if they match. This could be very useful.

<<interface>>
Item
+ getId() : int
+ setId(id : int) : void
+ getName() : String
+ setName(name : String) : void
+ getPrice() : BigDecimal
+ setPrice(price : BigDecimal) : void

<<interface>>
Sale
+ getId() : int
+ setId(id : int) : void
+ getUser() : User
+ setUser(user : User) : void
+ getTotalPrice() : BigDecimal
+ setTotalPrice(price : BigDecimal) : void
+ getItemMap() : HashMap<Item, Integer>
+ setItemMap(itemMap : HashMap<Item, Integer>) : void

<< interface >>
Inventory
+ getItemMap() : HashMap<Item, Integer>
+ setItemMap(itemMap: HashMap<Item, Integer>) : void
+ updateMap(item : Item, value : Integer) : void
+ getTotalItems() : int
+ setTotalItems(total : int) : void

<<enumeration>>
Roles
ADMIN
EMPLOYEE
CUSTOMER

*NOTE: Enumerators are discussed in the Appendix of the assignment

At this point, we have created an object that represents each table in the database except for three: the UserRole table, the UserPassword table, and the ItemeizedSales table. We will not be representing these directly with their own objects, as the UserRole is just associating a specific role to a specific user. The other two missing tables are UserPassword and ItemizedSales. These are special lookup tables that stores the passwords separate from the user itself, and the details on a sale. This is represented by the authenticate method on User and the itemMap inside of Sale. We separate data this way to create clean data tables.

Database Time [30 Marks]

In this section, we will be building the objects that help us to interact with the database in the way we'd like to. We will also be building the classes that extend the abstract base class for User. We will also be building a special object that works as a Ledger / SalesLog for our application.

When working with a database, it is good to add a few layers between user's interactions, and the raw access to the database. For this assignment, I have implemented all the SQL code for you, and have given you some starter code for helper methods that hide away the connection details from the user. We will need to make some modifications to the three helper classes: DatabaseInsertHelper, DatabaseSelectHelper, and DatabaseUpdateHelper.

All the base methods are there, what we need to do is validate that all values being passed in towards our database do not violate the expected values in any way. To help us understand this, let's refer to this chart:

Table	**Column**	**Accepted Values**
INVENTORY	ITEMID	A valid ID from the ITEMS table
	QUANTITY	The amount of this item currently available in the store. Must be 0 or greater.
ITEMIZEDSALES	SALEID	A valid ID from the SALES table
	ITEMID	A valid ID from the ITEMS table
	QUANTITY	The number of item ITEMID sold in sale SALEID. This quantity must be greater than zero
		SPECIAL NOTE: The SALEID + ITEMID together must be a unique combination.
ITEMS	Id	Integer value – autoincrementing
	NAME	The name of the item. Cannot be null, must be less than 64 characters.
	PRICE	The price to charge a customer for this item. Must be greater than zero. Stored as TEXT should have two decimal points of precision.

ROLES	ID	Integer value – autoincrementing
	NAME	The name of the role. This should come from your enumerator. You should prevent any value that is not part of your enumerator.
SALES	ID	Integer value – autoincrementing
	USERID	A valid ID from the USERS table
	TOTALPRICE	TEXT value representing a BigDecimal. This value must equate to: ITEMID*QUANTITY*PRICE for each ITEMID and QUANTITY in the ITEMIZEDSALES table with the SALESID equal to the ID.
USERS	Id	integer value – autoincrementing
	name	The name of the user for this account “Bob Smith” for example
	Age	The age of the user – this is an integer value
	Address	The address of the user – with a 100 character limit!
USERPW	NA	DO NOT EDIT THIS TABLE DIRECTLY
USERROLE	USERID	A valid ID from the USERS table
	ROLEID	A valid ID from the ROLES table

When implementing the given methods, be sure to read the starter code over, and add anything necessary to be successful. Be sure to note, when returning User type objects, you might be best to make private methods that construct the correct type of user based on their Id.

Once finished, implement the following classes which all extend User:

Admin
- id : int - name : String - age : int - address : String - roleId : int - authenticated : boolean

+ Admin(id : int, name : String, age : int, address : String)
+ Admin(id : int, name : String, age : int, address : String, boolean : authenticated)
+ promoteEmployee(employee : Employee) : boolean

The promote Employee method should take a user who is currently an employee and promote them to an administrator. This promotion should change their role in the database, and destroy their current object, loading a new object that represents an administrator. The return is true if the operation was successful, and false otherwise.

Employee
- id : int - name : String - age : int - address : String - roleId : int - authenticated : boolean
+ Employee (id : int, name : String, age : int, address : String) + Employee (id : int, name : String, age : int, address : String, boolean : authenticated)

Customer
- id : int - name : String - age : int - address : String - roleId : int - authenticated : boolean

+ Customer(id : int, name : String, age : int, address : String)
+ Customer(id : int, name : String, age : int, address : String, boolean : authenticated)

We also want to limit the number of items for sale. For our current application let's start by making only the items in the following enumerator available:

<<enumeration>> ItemTypes
FISHING_ROD HOCKEY_STICK SKATES RUNNING_SHOES PROTEIN_BAR

*NOTE: Enumerators are discussed in the Appendix of the assignment

Further developments [20 Marks]

In this section, we will be making the objects that the SalesApplication will use. These include an interface for Employees, and a ShoppingCart for customers to use.

Implement the following classes:

EmployeeInterface
- currentEmployee : Employee - inventory : Inventory
+ EmployeeInterface (employee : Employee, inventory : Inventory)

+ EmployeeInterface (inventory : Inventory) + setCurrentEmployee(employee : Employee) : void + hasCurrentEmployee() : boolean + restockInventory(item : Item, quantity : Quantity) : boolean + createCustomer (name : String, age : int, address : String, password : String) : int + createEmployee(name : String, age : int, address : String, password : String) : int

Method	Description
EmployeeInterface	If using the constructor with the employee, they must be authenticated in order to be set.
setCurrentEmployee	Set employee to the current employee. Employee must be authenticated to be set
hasCurrentEmployee	true if there is a current employee, false o/w
restockInventory	Update the Inventory with the quantity of item given. Return true if the operation is successful, and false otherwise.
createCustomer	Create a new customer with the information provided. If it is successful return the new userId, otherwise, raise an exception.
createEmployee	Create a new customer with the information provided. If it is successful return the new userId, otherwise, raise an exception.

ShoppingCart
- items : HashMap<Item, Integer> - customer : Customer - total : BigDecimal - {TAXRATE} : BigDecimal
+ ShoppingCart (customer : Customer) + addItem (item : Item, quantity : int) : void + removeItem (item : Item, quantity : int) : void + getItems() : List<Item> + getCustomer() : Customer + getTotal() : BigDecimal + getTaxRate() : BigDecimal + checkOut (shoppingCart : ShoppingCart) : boolean + clearCart() : void

NOTE: { } represents final

Method	Description
ShoppingCart	Customer must be logged in, if they are not, raise an appropriate exception.
addItem	add the quantity of item to the items list. Update the total accordingly

removeItem	Remove the quantity given of the item from items. If the number becomes zero, remove it entirely from the items list. Update the total accordingly
getItems	Return the current contents of the shopping cart
getCustomer	return the customer who is currently using this shopping cart
getTotal	return the total
getTaxRate	return the tax rate
clearCart	Remove all items from the cart.
TAXRATE	Constant value of 1.13.
checkOutCustomer	Take in a shopping cart, validate it has an associated customer. If it does, calculate the total after tax, and submit the purchase to the database. If there are enough of each requested item in inventory, update the required tables, and clear the shopping cart out, returning true. Return false if the operation fails, or if there are insufficient items available.

Putting it all together [20 Marks]

In this final section we will make a small application that will take in various user's inputs, and interact with our program. Finish your implementation of SalesApplication.java following the explanation in the file's TODO comments. You should use BufferedReader like we did on e1 to complete this!

Once you've finished this, you can start playing with your new application!

Section 3: Check in [0 Marks]

Please check in a file called revisionHistory.txt with the following format. Replace all items in [ ] with the corresponding values for your repository:

```
[STUDENT NUMBER] [LAST NAME], [FIRST NAME]
Starter code check in is r[Revision number]
I would like this revision marked: r[Revision number]
```

By adding the above, you earned yourself 3 marks!

Section 4: Style and Comments [15 Marks]

This is a freebie! If you follow the google styles guide and you put meaningful comments on your commits, you will get 15 marks! Yay!

[Marking: 5 marks for following style guide; 5 marks for appropriate comments in java program; 5 marks for appropriate svn comments]

Submission

Your submission will be the revision of the code that was checked in closest to the due date, that is not after the final due time.

This means you do not need to do anything more to submit!

Appendix

A note on BigDecimal

BigDecimal

For all money calculations in this project, we will be using BigDecimal. As some of you noticed well testing e3, it is very hard to compare values such as 0.1, as they are not stored very well in floats and doubles (more precisely – not stored very well in Binary). If you'd like to know more on why, you can read this article which does a good job explaining the concept. To combat this, Java has an immutable class called "BigDecimal" that is in the java.math package. The BigDecimal class should be ***only*** used with its String constructor, as you will already have lost your precession if you use the double-based constructor. When working with BigDecimals, always use String outputs as well, especially when writing to the database.

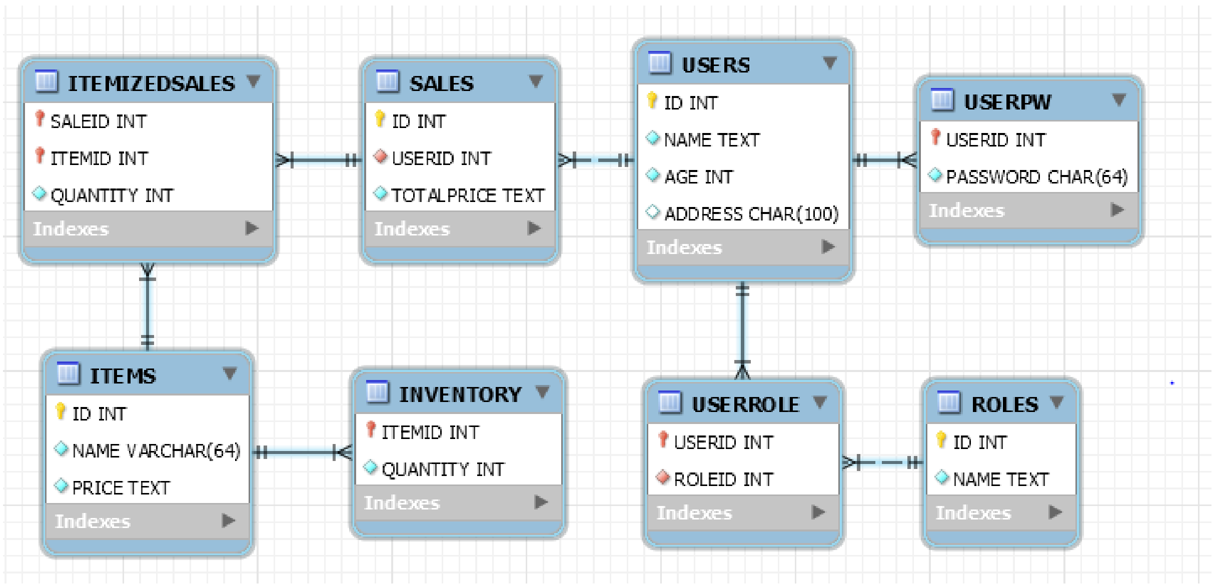
A note on Enum

Enum

For this assignment, there will be two Enum's used (short for enumerator). These are special objects that simply list off a fixed set of strings. In the google style guide, it is recommended that all values within an Enum are kept as all CAPITAL LETTERS. We use enum when we have a short list of things that will almost never change. In the assignment, we use these for the types of accounts, and the types of Roles in the system.

A note on the database and its structure

The database as is implemented is fully unchecked. This means that in theory, users could place anything they want in that meets the requirements of the field type. It is your job to ensure that anything you make that calls upon classes in com.b07.database package is fully checked, and that all values passing through are valid. The database model is as follows (you don't need to understand this, but in case you were curious and didn't want to read the SQL):



You may also find it useful to use a tool such as the SQLite Browser (https://github.com/sqlitebrowser/sqlitebrowser/releases/tag/v3.10.1) in order to take a look at what is currently stored in the database. The database can be found in your application folder in a file named inventorymgmt.db.