המכללה האקדמית כנרת- בית הספר להנדסה ע"ש אחי רכוב

המחלקה ל- .B.Sc בהנדסת תוכנה

שם הקורס: מבוא לתכנות מערכות

Final Project

Coupon management system

GENERAL

.The course includes an accompanying project detailed in the document

:The project is divided into three phases

Step 1: Build the system core. At this point the 'brain' of the system will be established. The core of the system will be responsible for receiving data, storing them in a database and managing them. At this point programming skills and the use of basic syntax will be applied to produce a working system and .meaningful logic

Step 2: Rewrite the core system for the latest technologies in the industry. At this stage, there will be a move to advanced and up-to-date platforms in the industry, such as Spring and Hibernate, and an .upgrade of the code

System description

Coupon management system enables companies to generate coupons as part of their advertising and marketing campaigns.
The system also has registered customers (Customers). Customers can purchase coupons. Coupons are .limited in quantity and validity. Customer is limited to one coupon of each type

.The system records the coupons purchased by each customer

The system's revenues are from the purchase of coupons by customers and the creation and updating of .new coupons by the companies

:Access to the system is divided into three types of Clients

.Administrator - Management of the list of companies and the list of customers .1

Company - Management of a list of coupons associated with the company .2

.Customer - Purchase coupons .3

General instructions

- Work on the project can be done independently or in groups of up to three programmers.
- Guidance and submission dates - each stage in the project is submitted by the date of the next stage. The project presentation dates appear in the course program. This schedule is rigid and its change requires the approval of the course coordinator or the professional bodies. In any case of a request for a change in the schedule a lowering of the score should be considered.
- The project should be presented working, with no compilation errors and no crashes.
- Source files should be submitted by submitting files online
- Source files should be submitted by submitting files online.
- Documentation must be used. It is advisable to produce Java Docs.
- A detailed text document regarding installation, usernames and passwords, database login data and user instructions must be submitted at each stage.
- If capabilities beyond the requirements have been added and / or APIs have been used beyond what is learned in the classroom - this is welcome, and the list of these additions must be noted when submitting for them to be addressed in the test.
- Do not submit a project that prints Stack Trace in cases of Exceptions. System-tailored Exceptions should be generated, and clear messages and user calls should be used in any case of error.

Phase 1

Building the core of the System

OOP, Java Beans, DAO, JDBC, Threads

Description

At this stage, the database for storing and retrieving information about customers, companies and coupons will be defined. An isolation layer called DAO (Data Access Objects) will be built over the .database, which will allow easy work from Java to the database

In addition, basic infrastructure services will be established, such as a database of links to the database .(ConnectionPool) and a daily job that maintains the system and cleans it of expired coupons

Three Entry Points will be defined for the system for each of the system's client types - administrator, .company and client, which will be connected by logging in

The following are the execution steps for Step 1

.Part A - Defining the database and building the tables

.Part B - Building Java Beans classes that represent the information in the database

Part C - Building a ConnectionPool that enables the management of the database connection to .the database

Part D - Construction of DAO classes that enable general CRUD operations to be performed on .the database

.Part E - Building the business logic required by the three types of Clients of the system

Part F - Building a class that allows Clients to enter and therefore using the appropriate business .logic

.Part G - Building a daily Job for deleting expired coupons from the system

Part H - Building a Test Class to demonstrate the system's capabilities and operating it from the .main

:Highlights

- It is important to document the code.
- It is important to use meaningful names for variables / functions / classes / packages
- Must write effectively, without copying code (DRY - Do Not Repeat Yourself).
- Custom Exceptions (Example: CouponAlreadyExistsException) should be used for specific system anomalies.
- A general try-catch must be set up in the Test Class capable of catching all the Exceptions.
- It is recommended to work with the MySQL database.
- It is mandatory to divide the classs into appropriate packages, with significant names.

Part A - Defining the database and building the tables

:The following is a list of required tables

:COMPANIES - TABLE OF COMPANIES .1

(ID (int - master key, automatic numbering •

(NAME (string - company name •

(EMAIL (string - company email •

(PASSWORD (string - login password •

:CUSTOMERS - Customer table .2

(ID (int - master key, automatic numbering •

(FIRST_NAME (string - first name •

(LAST_NAME (string - last name •

(EMAIL (string - customer email •

(PASSWORD (string - login password •

:CATEGORIES - Table of coupon categories .3

(ID (int - primary key, automatic numbering •

(.NAME (string - coupon category name - food, appliances, restaurants, vacations, etc •

:COUPONS - Coupon table .4

(ID (int - master key, automatic numbering •

(COMPANY_ID (int - foreign key to the company code in the membership table •

(CATEGORY_ID (int - foreign key to the category code in the category table •

(TITLE (string - coupon title •

(DESCRIPTION (string - detailed description of the coupon •

(START_DATE (date - coupon creation date •

(END_DATE (date - coupon expiration date •

(AMOUNT (integer - quantity of coupons in stock •

(PRICE (double – price of the coupon •

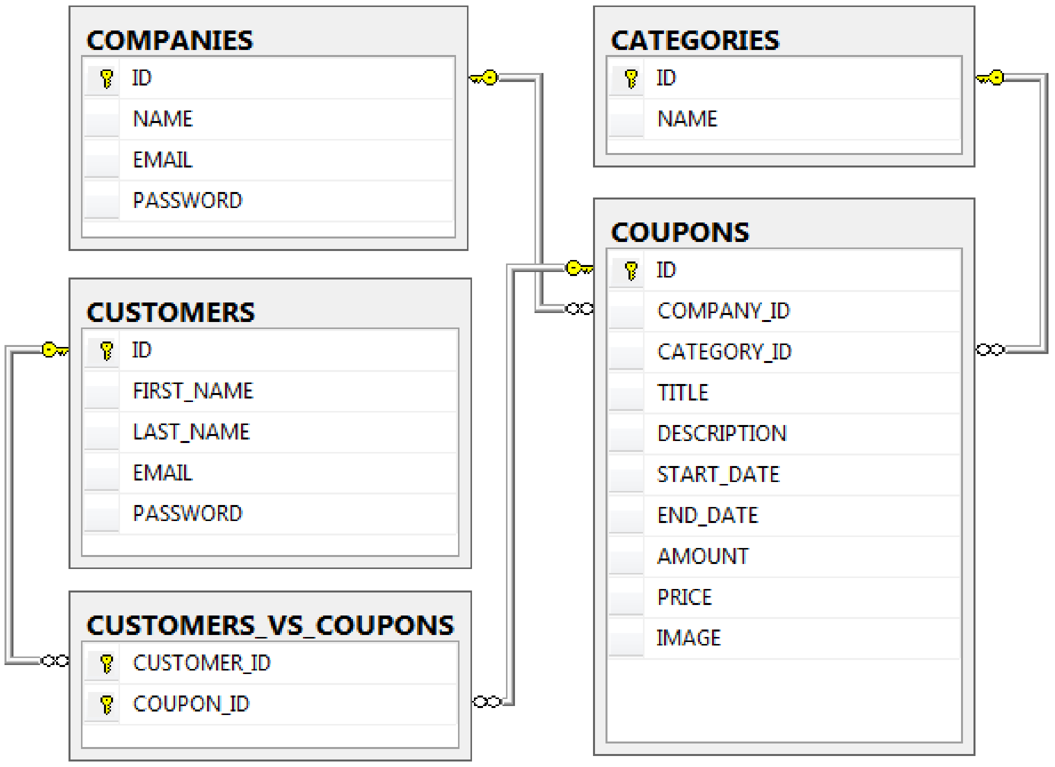
(IMAGE (string – name of the image file •

CUSTOMERS_VS_COUPONS .5

let's you know which coupons each customer purchased and which customers purchased each coupon

- CUSTOMER_ID (int – Primary key in this table and a foreign key to the customer code in the customer table)
- COUPON_ID (int – primary key in this table and a foreign key to the coupon code in the coupon table)

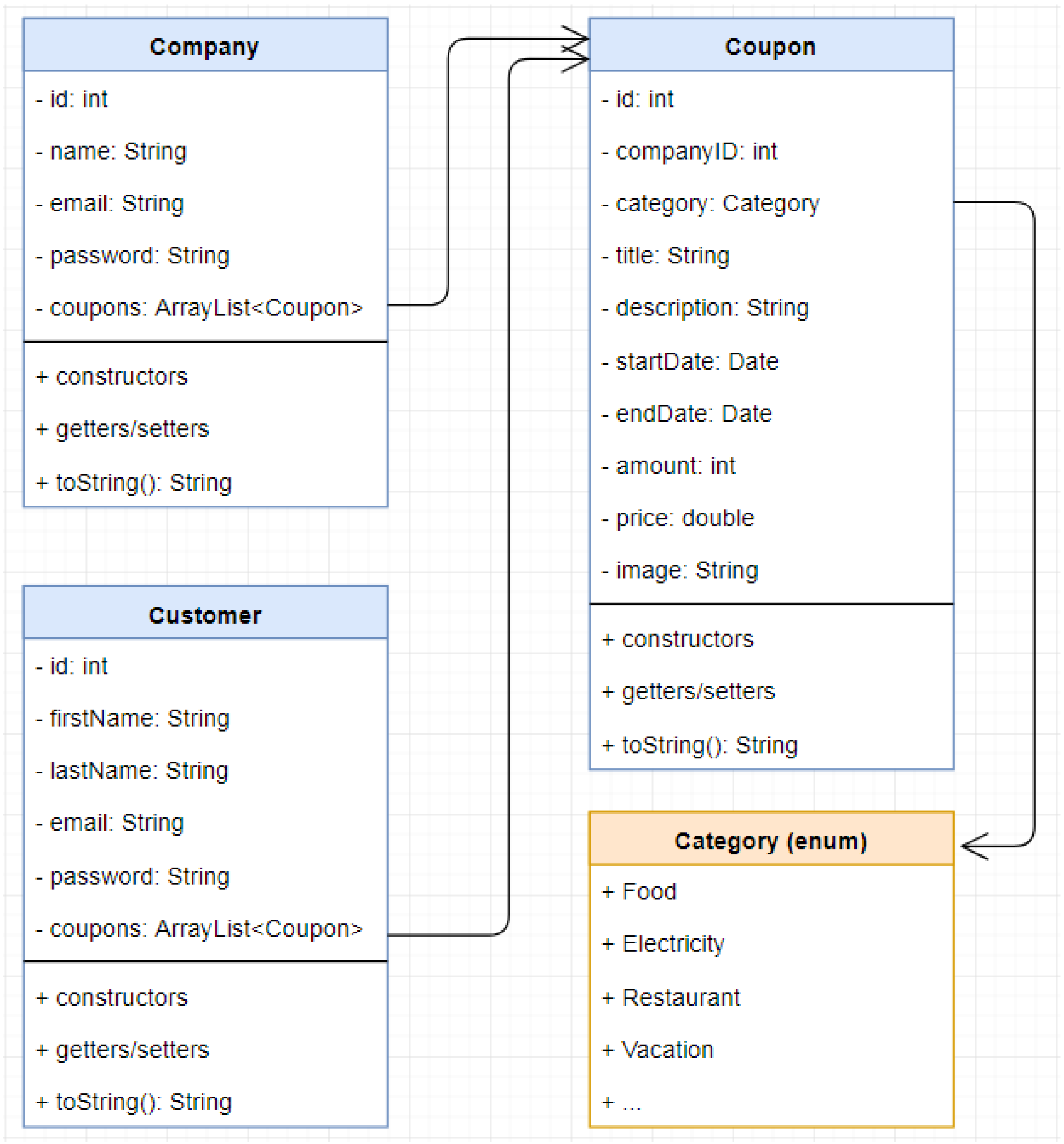
:The following is a summary of the tables and the relationships between them



Part B - Building Java Beans classes that represent the information in the database

Java Beans are pure information classes that represent the information managed by the application. The DAO classes, for example (Data Access Objects), receive Java Beans objects and translate them into SQL queries that are sent to the database. Each main table is represented by .its own class. The relationships between the tables are represented by appropriate Collections

Below is a summary of the Java Beans classes



Company
- id: int
- name: String
- email: String
- password: String
- coupons: ArrayList<Coupon>
+ constructors
+ getters/setters
+ toString(): String
Customer
- id: int
- firstName: String
- lastName: String
- email: String
- password: String
- coupons: ArrayList<Coupon>
+ constructors
+ getters/setters
+ toString(): String
Coupon
- id: int
- companyID: int
- category: Category
- title: String
- description: String
- startDate: Date
- endDate: Date
- amount: int
- price: double
- image: String
+ constructors
+ getters/setters
+ toString(): String
Category (enum)
+ Food
+ Electricity
+ Restaurant
+ Vacation
+ ...

Part C - Building a ConnectionPool that enables the management of the database connection to the database

Connection Pool is a Singleton class (a class from which there is a single object) that allows you to manage a fixed number of Connections to the database. The Connections are stored in a repository (Set type collection) at the class level

:The following are the methods required in the Class

()Connection getConnection

Removes one Connection object from the repository and returns it by return so that it can be .used to perform operations on the database

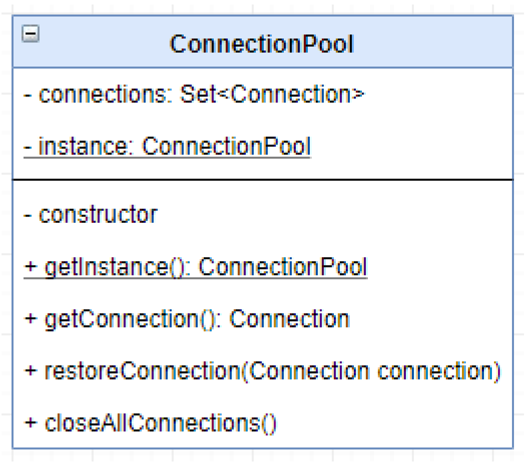
(void restoreConnection(Connection connection

Receives as an argument one vacant connection object and returns it to the repository. Because it has now been added to another Connection repository - performs notify to notify any thread that is waiting (i.e., performed a wait) that one Connection has now been freed and so you can .try to get it

()void closeAllConnections

.Closes all Connections in terms of database

:Below is the schema of the class ConnectionPool

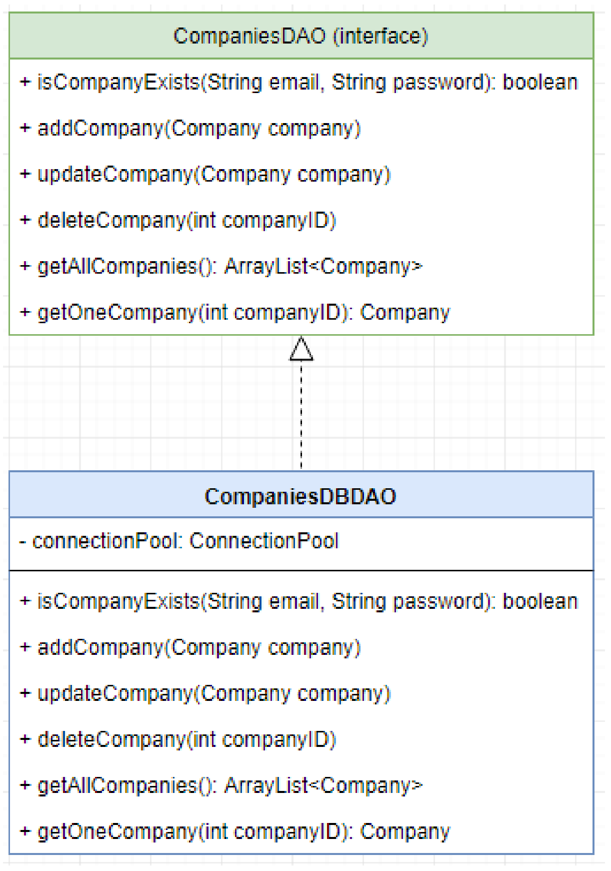

ConnectionPool
- connections: Set<Connection>
- instance: ConnectionPool
- constructor
+ getInstance(): ConnectionPool
+ getConnection(): Connection
+ restoreConnection(Connection connection)
+ closeAllConnections()

Part D - Construction of DAO classes that enable general CRUD operations to be performed on the database

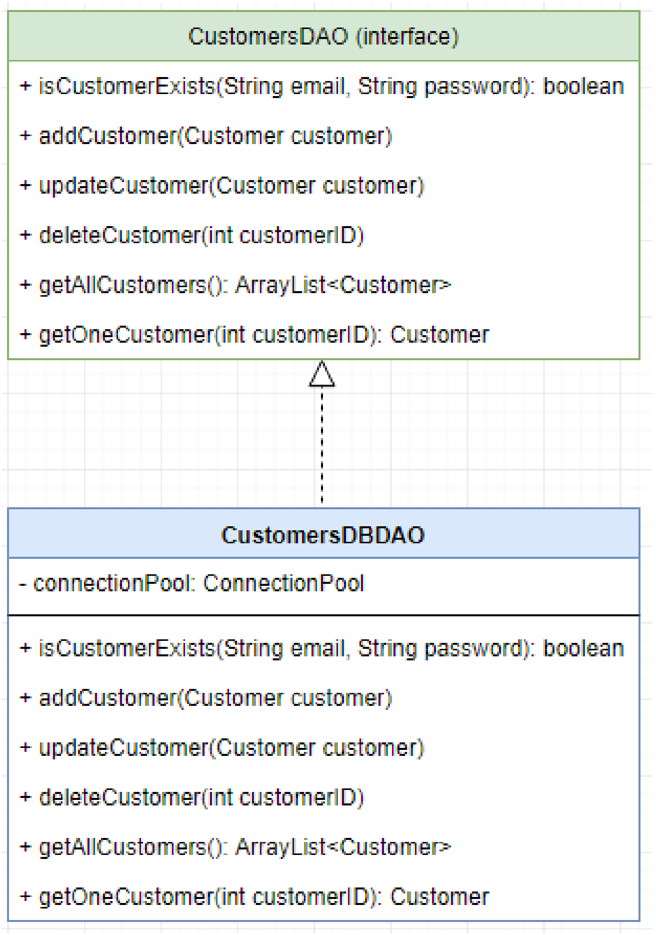
DAO (Data Access Objects) classes are classes that allow you to perform general CRUD (Create / Read / Update / Delete) operations on the tables in the database. These classs do not implement the logic associated with the application but only general CRUD operations. These classes accept as arguments Java Beans objects or simple primitive values (int, string, etc.), generate SQL queries from them and execute the queries in the database. They can also return individual Java Beans objects, Java Beans Collections, or simple primitive values. These classes .use the ConnectionPool to connect to the database to perform the various operations

Due to the separation of this layer, if a database other than the existing one is required in the future, it will be possible to replace the database without any contact with the rest of the system but only by replacing the DAO layer. We will define interfaces that contain the required .functionality for the DAO layer and implement this functionality in the various DAO classes

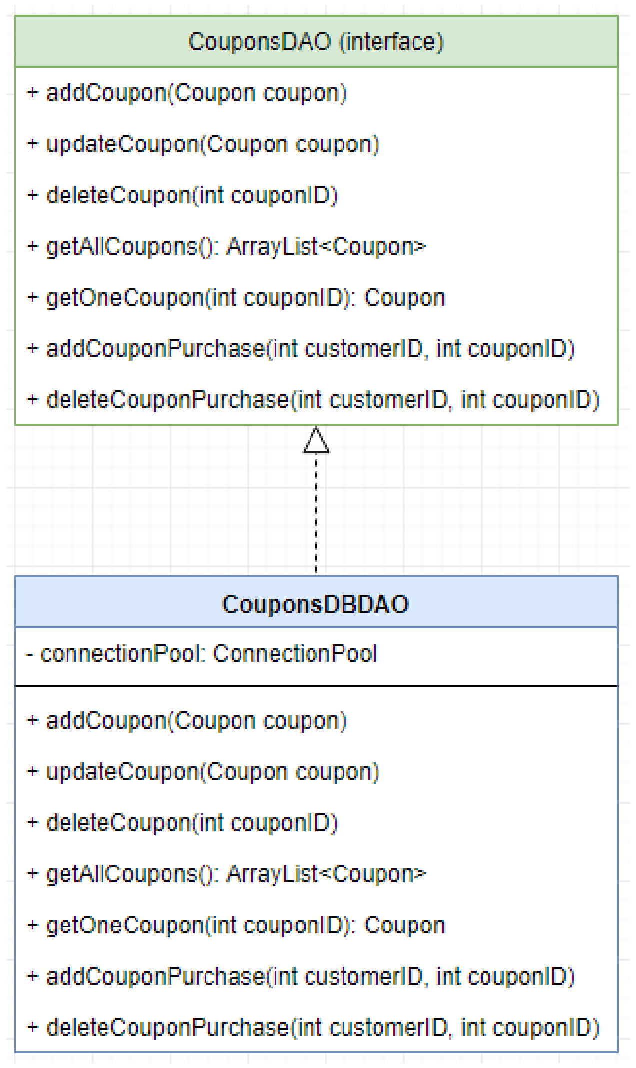
:Below is the schema of the class Company


CompaniesDAO (interface)
+ isCompanyExists(String email, String password): boolean
+ addCompany(Company company)
+ updateCompany(Company company)
+ deleteCompany(int companyID)
+ getAllCompanies(): ArrayList<Company>
+ getOneCompany(int companyID): Company
CompaniesDBDAO
- connectionPool: ConnectionPool
+ isCompanyExists(String email, String password): boolean
+ addCompany(Company company)
+ updateCompany(Company company)
+ deleteCompany(int companyID)
+ getAllCompanies(): ArrayList<Company>
+ getOneCompany(int companyID): Company

: Below is the schema of the class customer


CustomersDAO (interface)
+ isCustomerExists(String email, String password): boolean
+ addCustomer(Customer customer)
+ updateCustomer(Customer customer)
+ deleteCustomer(int customerID)
+ getAllCustomers(): ArrayList<Customer>
+ getOneCustomer(int customerID): Customer
CustomersDBDAO
- connectionPool: ConnectionPool
+ isCustomerExists(String email, String password): boolean
+ addCustomer(Customer customer)
+ updateCustomer(Customer customer)
+ deleteCustomer(int customerID)
+ getAllCustomers(): ArrayList<Customer>
+ getOneCustomer(int customerID): Customer

: Below is the schema of the class Coupon


CouponsDAO (interface)
+ addCoupon(Coupon coupon)
+ updateCoupon(Coupon coupon)
+ deleteCoupon(int couponID)
+ getAllCoupons(): ArrayList<Coupon>
+ getOneCoupon(int couponID): Coupon
+ addCouponPurchase(int customerID, int couponID)
+ deleteCouponPurchase(int customerID, int couponID)
CouponsDBDAO
- connectionPool: ConnectionPool
+ addCoupon(Coupon coupon)
+ updateCoupon(Coupon coupon)
+ deleteCoupon(int couponID)
+ getAllCoupons(): ArrayList<Coupon>
+ getOneCoupon(int couponID): Coupon
+ addCouponPurchase(int customerID, int couponID)
+ deleteCouponPurchase(int customerID, int couponID)

Additional general CRUD functions can be added to the above interfaces and classes if you •
.think it is necessary

In case of incorrect information sent to these classes, it is important to throw in Custom •
.Exceptions that clearly describe the errors

Part E - Building the business logic required by the three types of .Clients of the system

Client is considered anyone who can use the system. The three types of Clients of the system :are

.Administrator - the main administrator of the entire system .1

.Company - each of the companies in the system .2

.Customer - each of the customers in the system .3

Every such Client has the logical business operations that need to be performed by him. For example, an Administrator can add a new company, but a Company or Customer cannot (and should not) add a new company. For example, a customer can purchase a coupon, but an administrator or company cannot (and should not) purchase a coupon. Therefore, all the actions required to be performed by each Client will be in a dedicated class that contains all the .business logic required for that Client

Each such logical business activity will be performed by a dedicated function and will use the DAO classes to perform its logical activity. Such a function can, for the purpose of performing its .activity, perform a series of operations with the help of the DAO classes

For example, in order to purchase a coupon by a customer (one logical action that should be in the Customers' business logic class) we must perform the following series of actions using the :DAO classes

.A. Make sure the customer has not previously purchased such a coupon

.(B. Make sure the required coupon is still in stock (its quantity is greater than 0

.third. Make sure the coupon expiration date has not yet arrived

.D. Make the purchase of the coupon by the customer

.God. Decrease the quantity in stock of the coupon by 1

Such a design, which allows to externalize simple logical operations, which behind the scenes use a series of more basic operations, is called Facade because then we will only need to use these Facade classes to perform the necessary business logic in the system, without any further .use of the basic DAO classes More

:The three business logic classes are

.**AdminFacade** - Contains the business logic of the Administrator .1

.**CompanyFacade** - Contains the business logic of the Company .2

.**CustomerFacade** - Contains the Customer Business Logic .3

Because these three classes need to use DAO components, it would be appropriate to define a base class (abstract of course) that contains the DAO components, or at least only their References, and the different Facade classes will inherit this base class and create the

.appropriate DAO components for them

The following is the logic required to execute by an Administrator client to be built in the :AdminFacade class

- Login.
  - In this case (only for Administrator) it is not necessary to check the email and password in front of the database, but to check them as Hard-Coded.
  - The email will always be admin@admin.com and the password will always be admin.
- Adding a new company.
  - It is not possible to add a company with the same name as an existing company.
  - It is not possible to add a company with the same email as an existing company.
- Update an existing company.
  - The company code cannot be updated.
  - The company name cannot be updated.
- Deletion of an existing company.
  - The coupons created by the company must also be deleted.
  - In addition, the history of the purchase of the company's coupons by customers must also be deleted.
- Return of all memberships.
- Return of a specific company according to a company code.
  - Adding a new customer.
  - It is not possible to add a customer with the same email to an existing customer.
- Update an existing customer.
  - The customer code cannot be updated.

- Delete an existing client.
    - The customer's purchase coupon history must also be deleted.
        - Return of all customers.
            - Returning a specific customer according to a customer code.

The following is the logic required to execute by a Company client to be built in the :CompanyFacade class

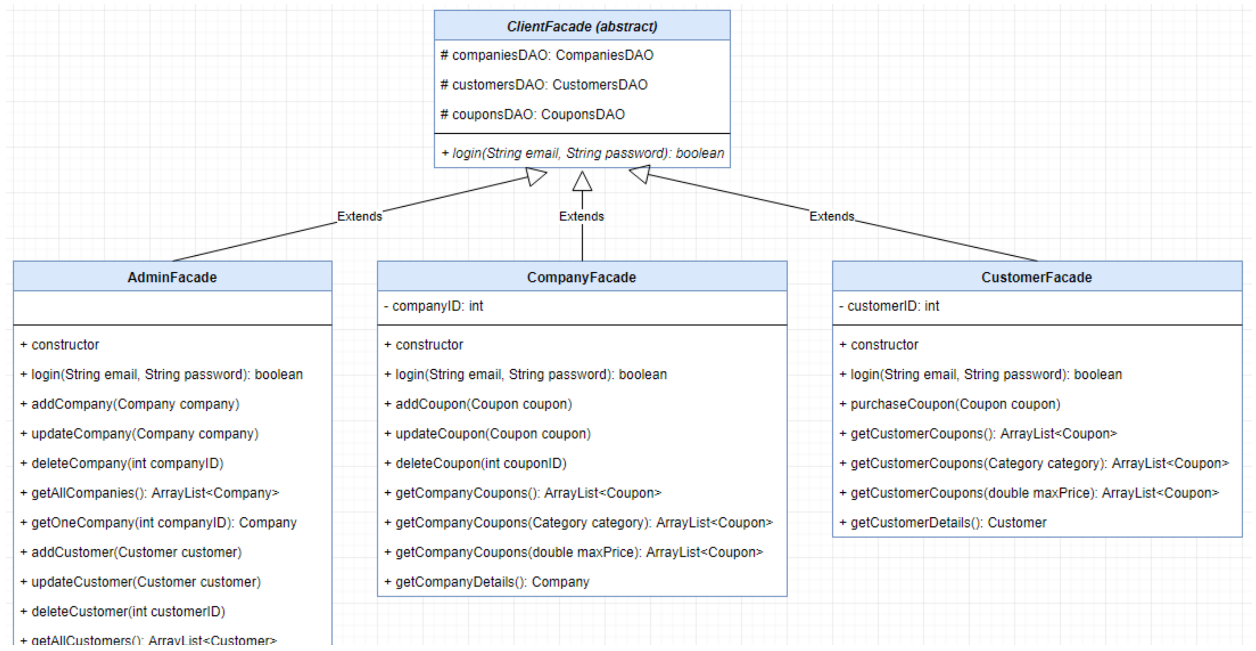
- Login.
    - Check the login details (email and password) against the database.
- Add a new coupon.
    - Do not add a coupon with the same title to an existing coupon of the same company. It is allowed to add a coupon with the same title as a coupon from another company.
- Update an existing coupon.
    - The coupon code cannot be updated.
    - The company code cannot be updated.
- Delete an existing coupon.
    - The purchase history of the coupon by customers must also be deleted.
- Return of all company coupons.
    - That is, all the coupons of the company that made the Login must be returned.
    - Return all coupons from a specific category of the company.
    - That is, only coupons from a specific category of the company that made it
    - should be returned

The following is the logic required to execute by a Company client to be built in the CompanyFacade class:

- Login.
    - Check the login details (email and password) against the database.
- Add a new coupon.
    - Do not add a coupon with the same title to an existing coupon of the same company. It is allowed to add a coupon with the same title as another coupon from another company.
- Update an existing coupon.
    - The coupon code cannot be updated.

    - The company code cannot be updated.
- Delete an existing coupon.
    - The purchase history of the coupon by customers must also be deleted.
- Return of all company coupons.
    - That is, all the coupons of the company that made it must be returned
- Return all coupons from a specific category of the company.
    - That means only coupons from a specific category of the company that performed Login should be returned.
- Return all coupons up to the company's maximum price.
    - That is, only coupons should be returned up to the maximum price of the company that performed Login.
- Return of company details.
    - That is, the details of the company that performed the login must be returned.
- The following is the logic required to execute by a Customer type Client, which must be built in the CustomerFacade class:
- Login.
    - Check the login details (email and password) against the database.
- Purchase a coupon.
    - You cannot purchase the same coupon more than once.
    - The coupon cannot be purchased if its quantity is 0.
    - The coupon cannot be purchased if its expiration date has already arrived.
    - After the purchase, the quantity in stock of the coupon must be reduced by 1.
- Return of all coupons purchased by the customer.
    - That is, all the coupons purchased by the customer who made the login must be returned.
- Return all coupons from a specific category purchased by the customer.
    - That means only coupons from a specific category of the customer who performed Login should be returned.
- Return all coupons up to the maximum price the customer purchased.
    - That means only coupons should be returned up to the maximum price of the customer who performed Login.
- Return of customer details.
    - That is, the details of the customer who performed the Login must be returned.

The following is a summary of the Facade :departments

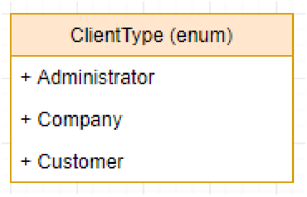

ClientFacade (abstract)
# companiesDAO: CompaniesDAO
# customersDAO: CustomersDAO
# couponsDAO: CouponsDAO
+ login(String email, String password): boolean
Extends
Extends
Extends
AdminFacade
+ constructor
+ login(String email, String password): boolean
+ addCompany(Company company)
+ updateCompany(Company company)
+ deleteCompany(int companyID)
+ getAllCompanies(): ArrayList<Company>
+ getOneCompany(int companyID): Company
+ addCustomer(Customer customer)
+ updateCustomer(Customer customer)
+ deleteCustomer(int customerID)
+ getAllCustomers(): ArrayList<Customer>
CompanyFacade
- companyID: int
+ constructor
+ login(String email, String password): boolean
+ addCoupon(Coupon coupon)
+ updateCoupon(Coupon coupon)
+ deleteCoupon(int couponID)
+ getCompanyCoupons(): ArrayList<Coupon>
+ getCompanyCoupons(Category category): ArrayList<Coupon>
+ getCompanyCoupons(double maxPrice): ArrayList<Coupon>
+ getCompanyDetails(): Company
CustomerFacade
- customerID: int
+ constructor
+ login(String email, String password): boolean
+ purchaseCoupon(Coupon coupon)
+ getCustomerCoupons(): ArrayList<Coupon>
+ getCustomerCoupons(Category category): ArrayList<Coupon>
+ getCustomerCoupons(double maxPrice): ArrayList<Coupon>
+ getCustomerDetails(): Customer

- The companyID variable in the CompanyFacade class should contain the company code that performed Login.
- The customerID variable in the CustomerFacade class should contain the customer code that performed Login.
- Additional auxiliary functions can be added to the above departments if you think it is necessary.
- In case of incorrect information sent to these departments, it is important to throw in Custom Exceptions that clearly describe the errors.

Part F - Building a department that allows clients to enter and therefore returning the appropriate Facade department

This class is a Singleton class called LoginManager that contains a Login function that allows any .of the three types of Clients to connect to the system

:First, build an Enum called ClientType that describes each of the Client types



ClientType (enum)
+ Administrator
+ Company
+ Customer

Second, a Login function must be built in the LoginManager department that will receive a client-type email, password and variable. The function should check whether the login information is correct according to ClientType. If the login information is incorrect - the function will return null. If the login information is correct - the function will return the appropriate :Facade class

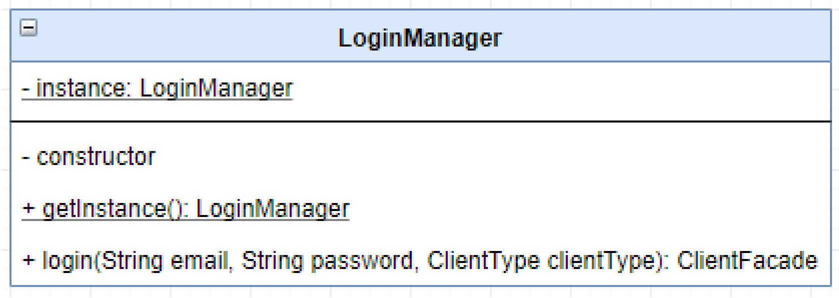
.For Administrator who logged in correctly - the AdminFacade object will be returned •

.For Company that entered correctly - CompanyFacade object will be returned •

.For a Customer who entered correctly - the CustomerFacade object will be returned •

.For each incorrect entry - null will be returned •

:The following is the schema of the LoginManager department



LoginManager
- instance: LoginManager
- constructor
+ getInstance(): LoginManager
+ login(String email, String password, ClientType clientType): ClientFacade

Part H - Building a daily Job for deleting expired coupons from the system

Job is a process that runs in the background regularly and performs some action. A job can perform its action non-stop, or it can perform it at specific times, for example once an hour or once a day or once a week, etc. - depending on the activity it is supposed to perform. For the purpose of performing an action at specific times, it is possible to check every second / minute / hour, etc., whether the required time has already arrived, and if so, to perform the required .action

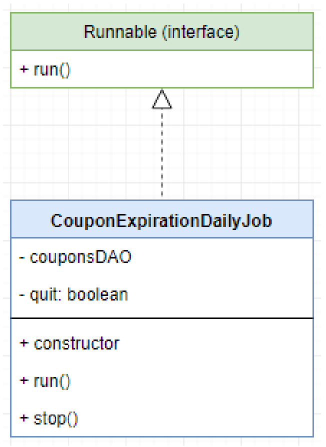
Job realization must be by a separate thread, as it should run non-stop in parallel with the rest .of the system activities

This section requires the construction of a daily Job, which means that it will be done once every day, and that expired coupons will be deleted. In order to delete an expired coupon, the coupon must be deleted from the coupon table and the purchase history of the coupon must .also be deleted

In the Job class you have to build a function that starts the Job and a function that causes the .Job to end

The job should start working at the beginning of the program and end at the end of the .program

:The following is the Job Schedule for deleting expired coupons from the system



- The Job must be started when the system is booted (program start).
- The Job must be stopped when the system closes (end of the program).

Part I - Building a Test Department to demonstrate the system's capabilities and operating it from the main

Is called Test and contains one static function called testAll whose function is to test the whole system by calling each of the business logic functions you have built. There is no need to get anything from the user. All tests should be Hard-Coded. Later - the information will be .transmitted to the system by the website

:The test All function must perform the following actions

.A. Running the Daily Job

B. Login by LoginManager as Administrator receive an AdminFacade object and call each of its .business logic functions

C. Login by LoginManager as a Company receive a CompanyFacade object and call each of its .business logic functions

D. Login by the LoginManager as a Customer receive a CustomerFacade object and call each of .its business logic functions

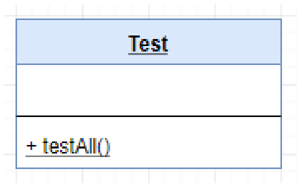
.God. Cessation of the daily job

.(and. Closing All Connections (Calling the CloseAllConnections function of the ConnectionPool

All of these actions must be set within one try-catch and the error message displayed in case of .an exception

Do not display Stack Trace messages on the screen, only clear error messages that come from .Custom Exceptions

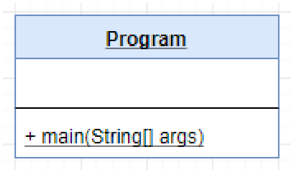
:The following is the schema of the Test Department



.The main class of the system is the Program class, which contains the main function

.The main function must make one call to the testAll function of the Test class

:The following is the schema of the Program Department



Phase 2

Rewrite the system using Spring technology

Spring Framework, Spring Boot, JPA, Hibernate

Description

At this point the system will actually be rebuilt based on Pure Spring and Hibernate.

You can of course transfer code from Phase 1 of the project and "adopt" it to the Spring environment.

Of course, on this occasion it is also advisable to rewrite realizations in need of improvement.

The project must be based on Spring Boot with the following Extensions:

- Spring JPA
- MySQL Driver
- Spring WEB

The following are the execution steps for phase 2:

- The DAO components should be implemented using Entity Beans equivalent to Java Beans from the first stage and using Spring Repositories.
- Custom Queries should be added to Repositories built using Spring according to the requirements of the first step.
- Facades departments should be transformed from the first step to @ Services.
- The CouponExpirationDailyJob class should be implemented as a Singleton.
- The LoginManager class should be implemented as a Singleton.
- A general test of the entire system should be performed by creating a class called Test which is also defined as Singleton with a method that performs a "purpose view" (display and test of the entire system). The method that executes the "purpose view" from SpringBootApplication.main () must be run so that the project run will automatically run the code in the Test class.

!!Thank you

Daoud & Dror