Updated Program Specification

Domain: Giveaway application

Brief Description:

- 1-line description: A giveaway raffle application, allowing users to create, participate, complete tasks and win prizes!
- Features include: creating customized raffles with specific tasks as organizers, joining raffles as participants, completing various tasks to win
- So how does it work?
- Organizers create raffles with specific dates and rules, and create tasks that they want a participant to complete (via a weblink).
- Participants search these raffles, enroll in them and then complete tasks by opening links to qualify as valid candidates
- Organizers generate winners by randomly choosing from those participants who have completed all tasks, and automatically notify them of their winnings via email and in app.

***Main Features*:**

- Login/Create account page for all users (participants and organizers), account login/creation process is the same for both types of users except for when creating an account, where the user has to specify whether they want to sign up exclusively as a raffle organizer or participant.
- Main page for users to check the raffles in which they are involved (this main page is the same for both types of users, with a few differences for organizers). From here you can enter a raffle's page (SubPage) where the range of actions of the user are limited by their account's role (organizer or participant). Raffles can be "completed" (by completing tasks as a participant), or "modified" (by changing some of the raffle's characteristics as an organizer).
- Within this Main page, organizers will also have the option to create new raffles and tasks which are the requisites for a participant to complete, while participants should have the option to search for and then join an ongoing raffle.

***Inside a raffle's subpage*:**

- As a participant: you can see the details of the raffle you are enrolled in, and you can choose to "complete" a raffle by completing the tasks displayed on the screen as indicated by the organizer (this involves entering an answer, getting this answer checked at the moment, and getting feedback on whether the task was successfully completed).
- As an organizer: you can see the details of the raffle as set up at the moment of the raffle's creation, from here you can edit certain attributes like adding a raffle rules text file, adding raffle tasks for users to complete, and changing the raffle's end date. From this page, the organizer should also be able to generate the set of random raffle winners, and notify such winners via email.

Program Specification:

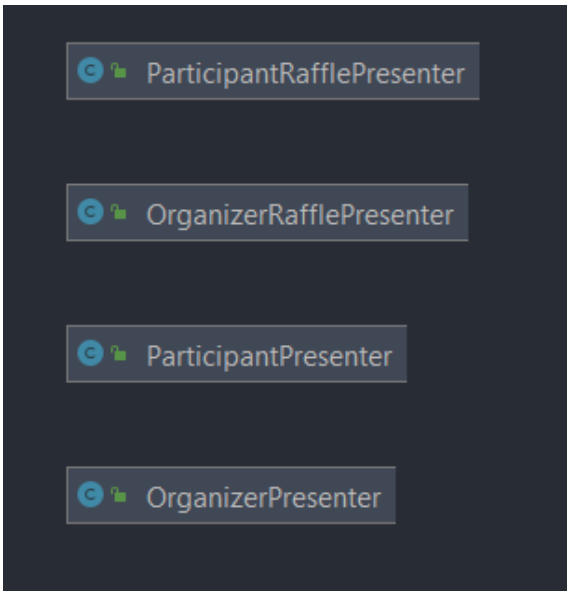
- Entities:
    - Users (Organizers and participants)
    - Raffles (Organizer and participant)
    - Tasks
- UseCases
    - CompleteTask
    - CreateRaffle
    - ExecuteTask
    - Many more accounting for various functional aspects of the program

- Controllers
  - UserController
  - OrgRaffleController
  - PtcRaffleController
  - TaskController
  - UserController
- GUI and Presenters
  - Raffle Pages
  - User Pages
  - Sign in and sign up pages
  - User Presenters
  - Raffle Presenters
- Managers
  - OrganizerSystemManager
  - ParticipantSystemManager
- Database
  - AccessData
  - Getters (RaffleGetter, TaskGetter, UserGetter)
  - Adders (RaffleAdders, TaskAdders, UserAdders)
  - DataTools

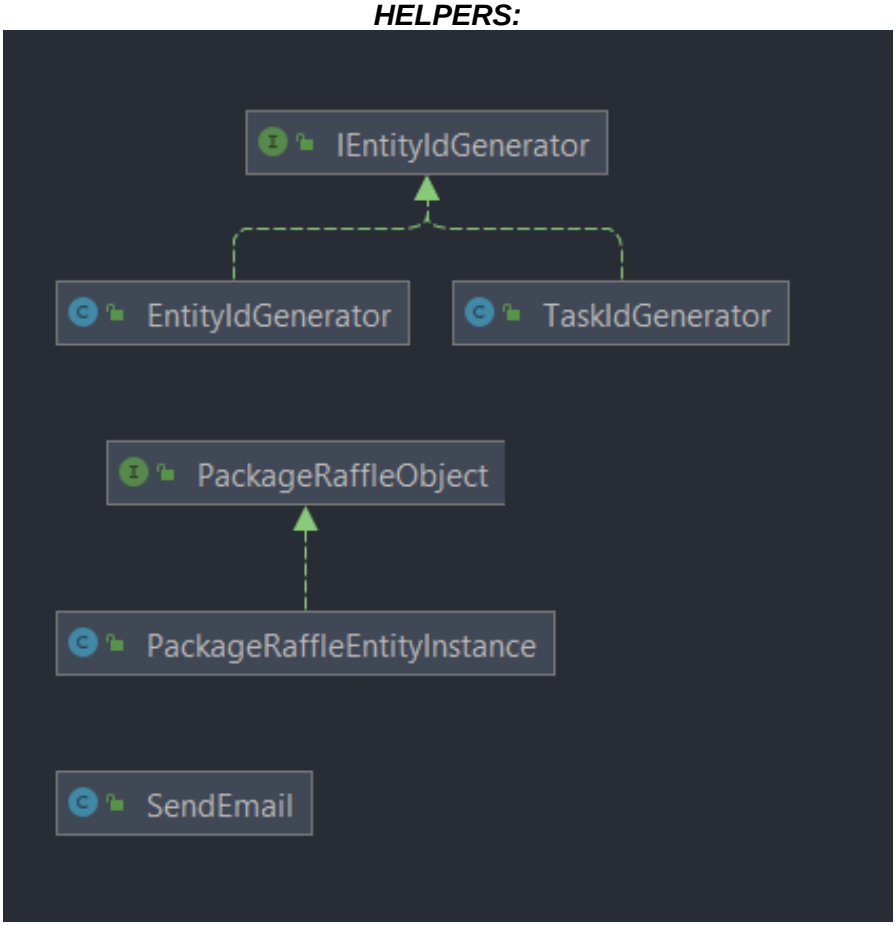
Diagram of Code (UML DIAGRAM):

GUI:

PRESENTERS:

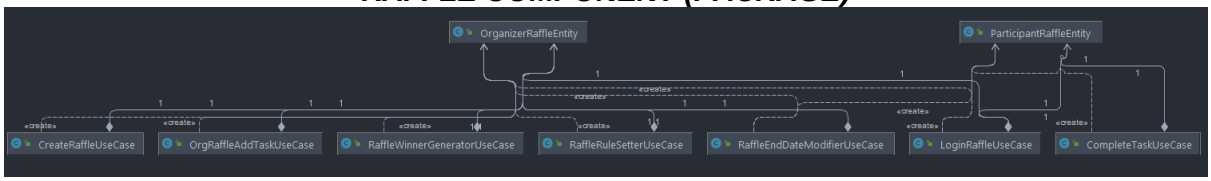


HELPERS:

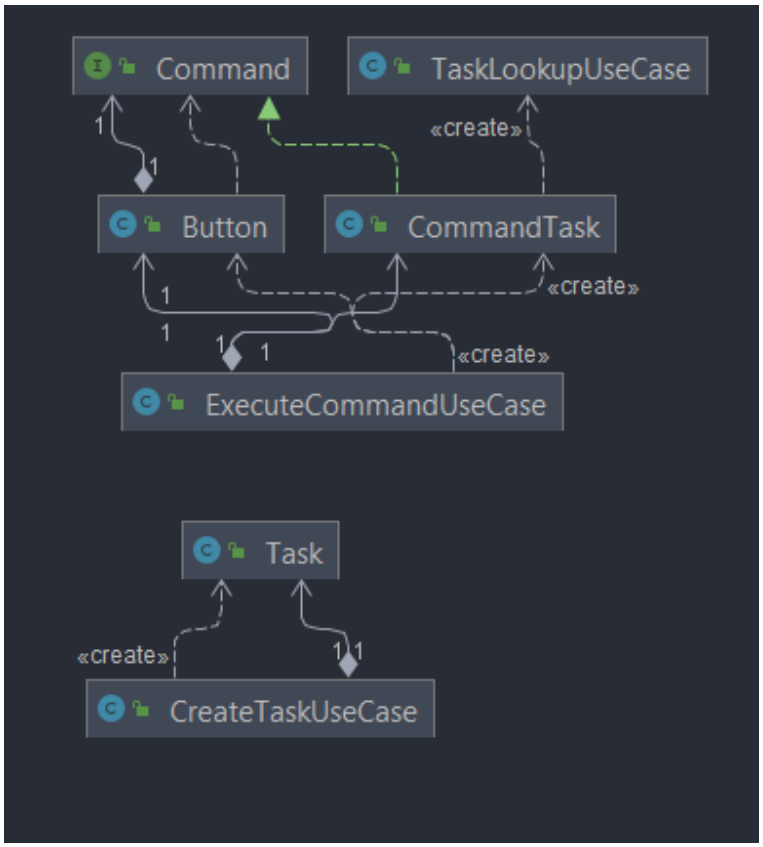


DATABASE_RE:

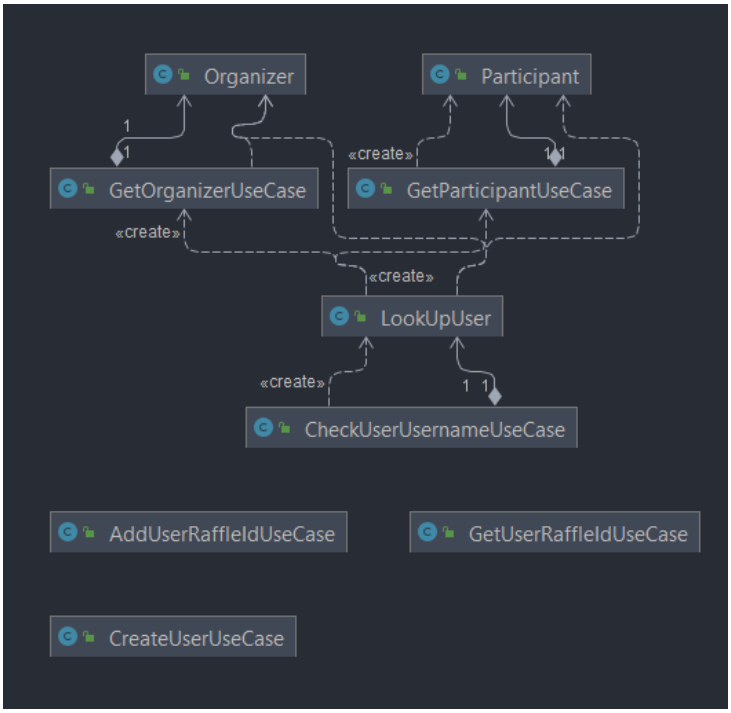
RAFFLE COMPONENT (PACKAGE)



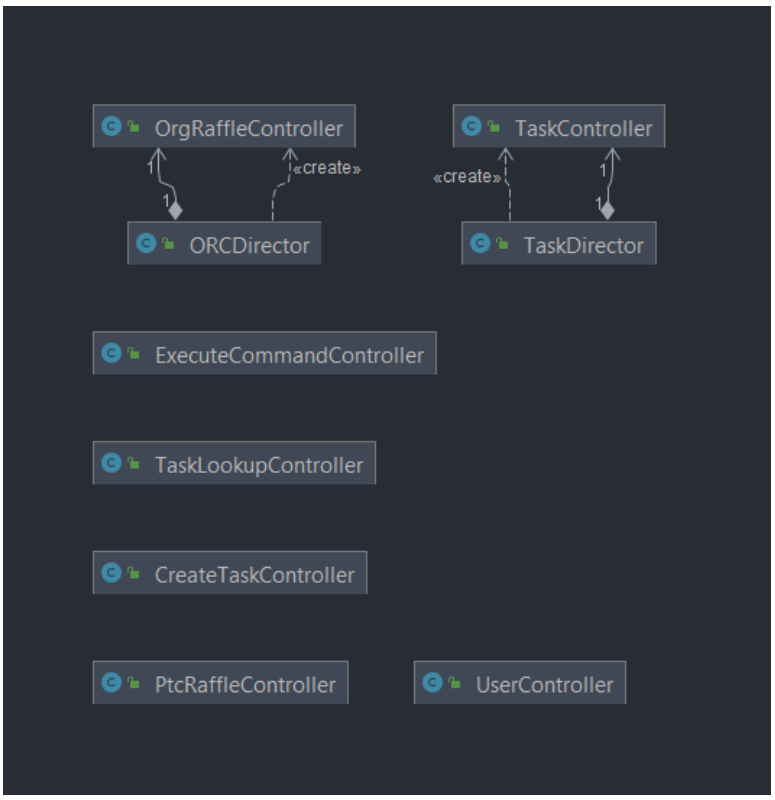
TASK COMPONENT



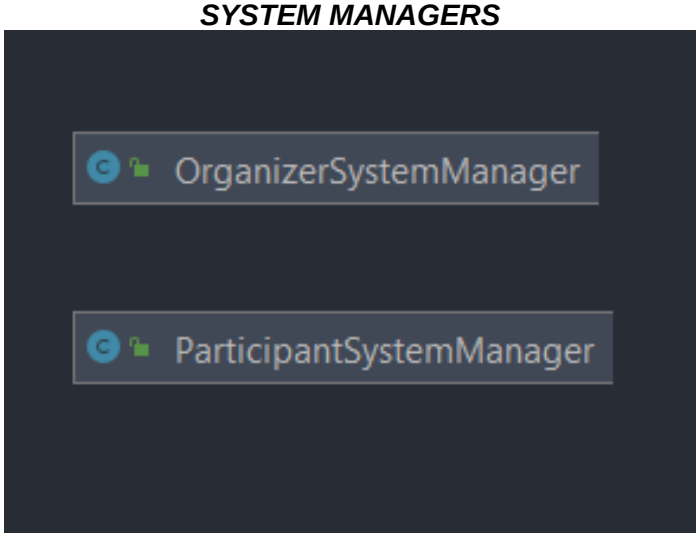
USER COMPONENT


Organizer
Participant
1
1
«create»
1
1
GetOrganizerUseCase
GetParticipantUseCase
«create»
«create»
LookUpUser
«create»
1
1
CheckUserUsernameUseCase
AddUserRaffleIdUseCase
GetUserRaffleIdUseCase
CreateUserUseCase

WEB


OrgRaffleController
TaskController
1
1
«create»
«create»
1
1
ORCDirector
TaskDirector
ExecuteCommandController
TaskLookupController
CreateTaskController
PtcRaffleController
UserController

SYSTEM MANAGERS



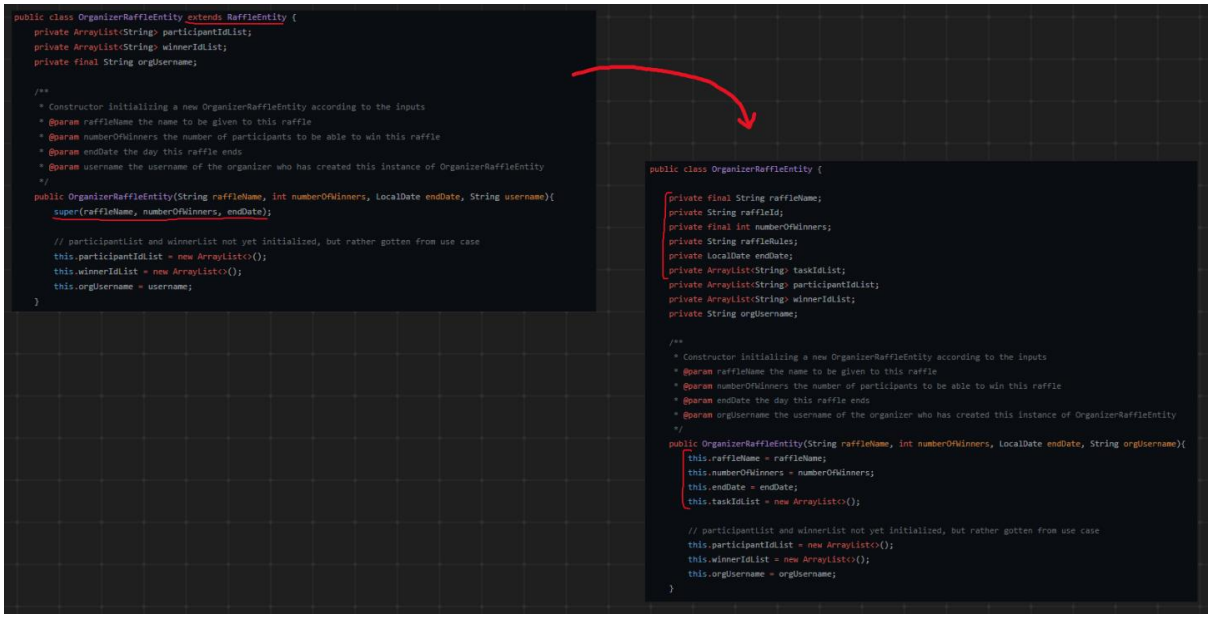
Major Design Decisions

Reason for not implementing Composite Design Pattern for System Manager classes

- The entire program can be represented in a tree hierarchy structure starting from the main sign in page, with subtrees being the previous page and subtree showing all raffle details for the organizer/participant and so on and so forth branching upto the leaf level being individual raffle details. However, because each subtree is much different than the others, there isn't a simple way to over-generalize the common interface with respect to the Composite Design Pattern. Thus we decided to have a linear structure to our code and make appropriate calls to different GUI classes for keeping track of the connections.

Separation of Organizer and Participant raffle into separate objects:

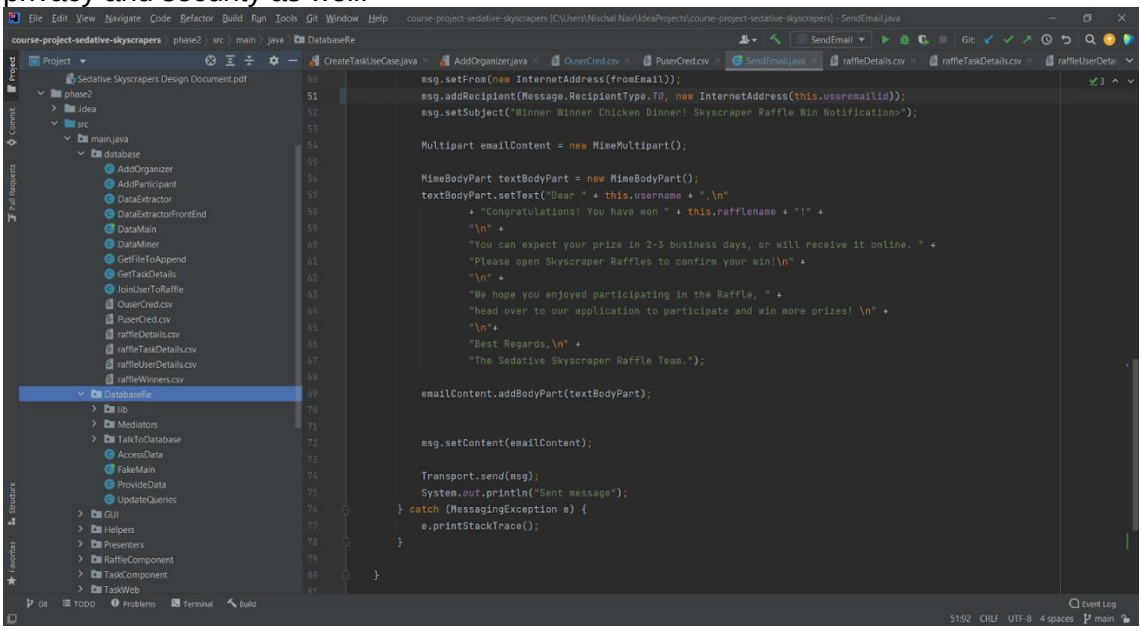
- During the process of making a subscription system between organizer raffles and participant raffles through the observer pattern, things started to look strange. An organizerRaffle would change and then call for the raffleEntity instances it stored through its ptcIdList attribute to change, but these ptcIds referred to raffleEntity instances (at the time, this ended up being changed to just actual participant user ids, but the it helped us realize the change was still relevant), so the organizerRaffleEntity was storing and changing instances of its parent object. At this point we decided to have a discussion and realized that there was no longer an "is a" relationship between orgRaffleEntity and the raffleEntity, which we had lost track of because of the distraction generated by how similar these objects were in terms of the attributes they stored. So it was at this moment, that we decided to refactor this change in (*main(#41), commit a8bc8da9bd47d15c65f7b4d21fe60ad7e60ba224)* and completely separate these two classes.



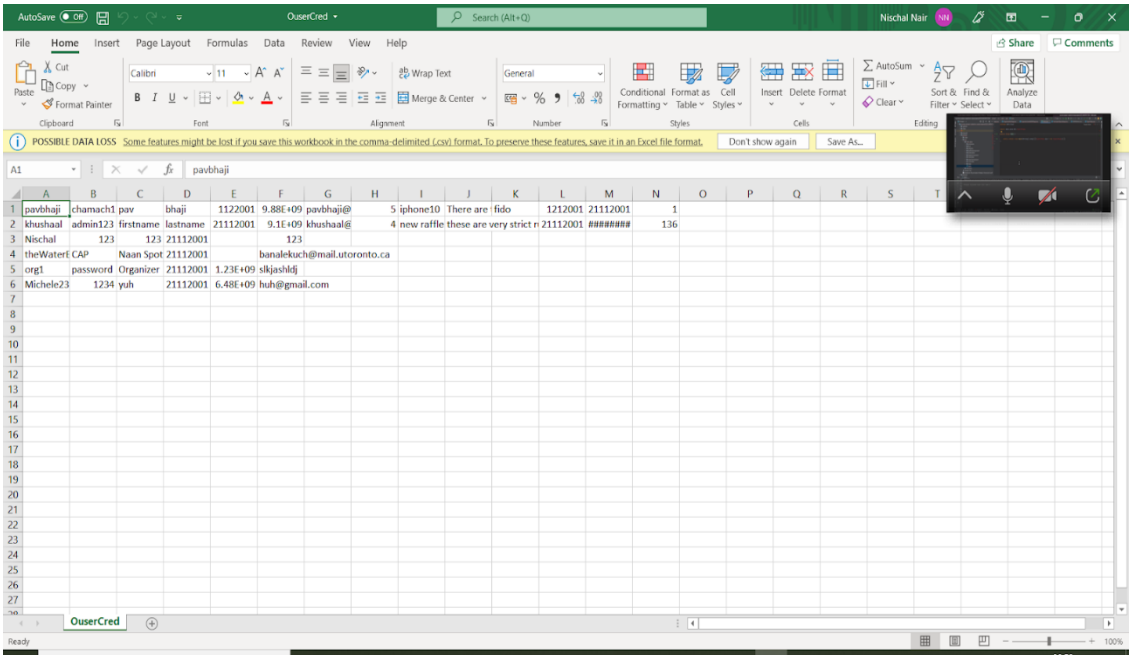
(Superficial before and after code screenshots just showing the base modification and the idea behind the breaking of this parent-child class relationship).
Michele p2 by MicheleMassa802 · Pull Request #41 · CSC207-UofT/course-project-sedative-skyscrapers (github.com).
(More specifically commits 27ce6a5 and a8bc8da)

Restructuring of Database:

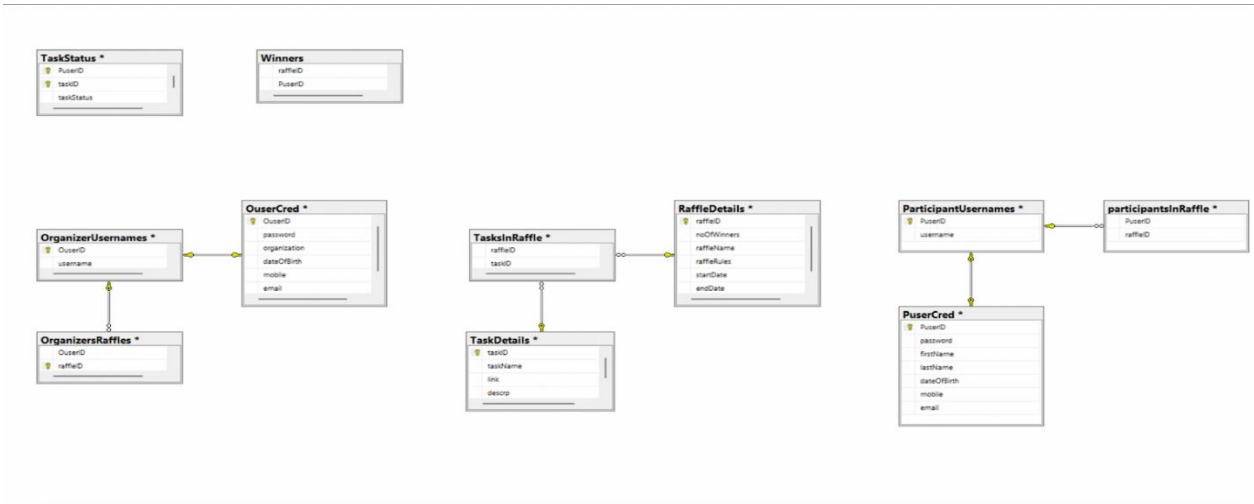
- Originally, our database was implemented using CSV files. We decided to refactor these to SQL, for two major reasons. First, the SQL database would be structured better and allow for vastly improved functionality, allowing the database to do many more things than it did before. Examples include explicitly keeping track of task completion statuses, winners lists. Second, the SQl database is now centralized and online, meaning that multiple users will now be on the same database, unlike the CSV implementation which was stored locally for each user. This vastly improves privacy and security as well.



Old database packaging vs. new (DatabaseRe)



Old CSV database file



New SQL Database Structure

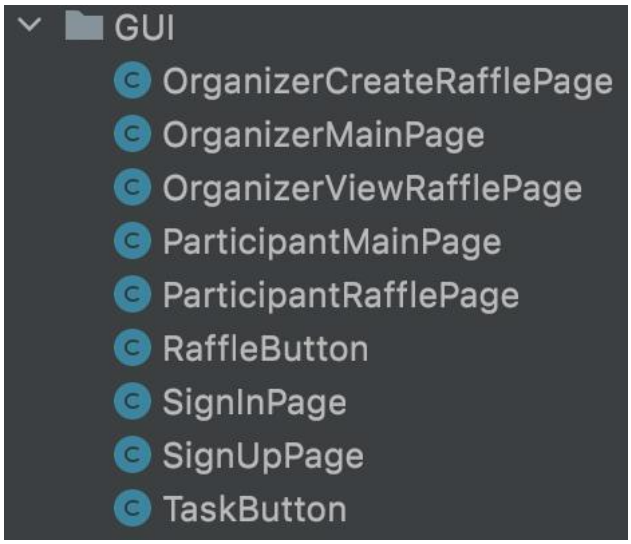
***Restructuring of GUI*:**
After getting feedback from the TA, we refactored the GUI. The link to the pull request can be found here https://github.com/CSC207-UofT/course-project-sedative-skyscrapers/pull/55

- We realised that the GUI was not only doing the work of displaying but also doing the work of presenting. Essentially, the GUI was itself deciding what to output to the user. Hence, we decided to create Presenter classes that would relay display information to GUI.
- There were no back and exit buttons on many pages, hence, limiting user options. To solve this problem we put “Back” and “Exit” buttons on every single page.

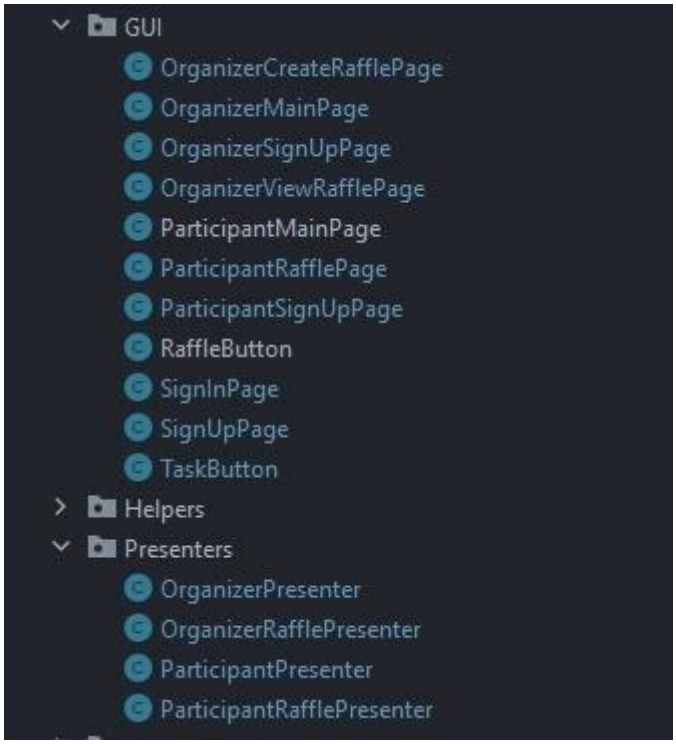
Phase 1

Phase 2

- We also noticed that it was very hard to search for raffles since earlier it was based on Raffle IDs. We upgraded the search feature to make it easier. Now, it searches according to Raffle Name, Raffle ID and Organization, thus, enhancing user experience.
- The SignUpPage was too long. Hence, we have split it into multiple classes.



Phase 1



Phase 2

How our Software Adheres to Clean Architecture

Consistent with clean architecture?

- In our last design document we talked a lot about how well everything adhered to clean architecture up to the point where the phase1 database entered the picture, and how we had use cases directly accessing methods of this database's gateway class (mainly because of time). This issue has now been resolved meaning that now our use case to database gateways connection properly follow clean architecture, which we have done by using the DIP, so we now have these interfaces which the gateway classes implement, while our use cases only really make contact with these interfaces, essentially ridding ourselves from this incorrect dependency.
- The rest of our program and the changes we have made for this phase still follow the core of clean architecture which is the dependency rule, and we take pride in how closely we have looked after each class to make sure the volatility of our code is only found in the outer layers, and how the proper usage of clean architecture has made it relatively simple to expand the functionality of our code. We still however feel like we might have fallen a little short in terms of our usage of our system manager classes, since we have them as a bridge between our GUI inputs and the respective controllers due to the complex nature of the GUI, but at least the expansion onto functioning controllers has relieved a great deal of pressure from these system managers. We would have loved to implement the facade design pattern with these system manager classes, but as controllers started to gain ground, these system managers got a little too small to have an entire facade design pattern just for two relatively small classes.

Scenario Walkthrough demonstrating clean architecture:

This example: a new user wants to create a raffle.

- When the program is run, the user is presented with the initial page where they can choose to sign up (in their case as an Organizer since they want to create a raffle) and enter the credentials by which the program will recognize them through; at this point, the GUI feeds these details to the system manager, which calls the UserController and inside here, the credentials are fed to the GetOrganizerUseCase where a new Organizer entity is created and then fed to the database (violating clean architecture in this call, which sends the details directly to the databases' updater method). After this the user is thrown to the main page where they would select to create a raffle, this process consists of 3 steps:
- In the first step, the user will be presented with the core information that must be provided in order to create such raffle, at this point the system manager calls the CreateRaffleController to have it call its CreateRaffleUseCase with the information the user has provided, which this use cases processes and sends back to the controller to keep this information until the process is finished.
- In the second step the user must enter the rules for the raffle, which follow the same structure of calling and receiving as the CreateRaffleController and use case, but now through the RaffleRuleSetterController and use case, allowing the system manager to hold onto this information until the process is done.
- In the third step, the user will need to enter the information about the tasks they want the participants of the raffle to complete, this requires calls to the CreateTaskController, which creates a Task instance with the required information through the CreateTaskUseCase, this task data is then, through this use case, sent to the database where the details of this entity are stored (violating clean architecture in this call, which sends the details directly to the databases' updater method). Along with the creation of the Task entities, the raffle that the user has created must now update its taskIdList attribute to reflect the added tasks, which is done by having the system manager call the RaffleTaskController, which feeds this list of task ids and all the previous raffle-related information that has been saved throughout the other two steps to the OrgRaffleTaskEditUseCase, which accordingly creates an OrganizerRaffleEntity with all of this information, which is then packaged and sent to the database where it is to be stored and accessed whenever the details of this raffle are to be displayed (violating clean architecture in this call, which sends the details directly to the databases' updater method).

How our software is consistent with SOLID principles update

Dependency Inversion Principle:

- It was something that came in handy this phase in order to help us solve one of our unwanted dependencies which went against the dependency rule of clean architecture. The issue arising from having use cases call methods from a gateway class directly (going against the desired dependency flow), meant that we needed a way to deal with this wrong-facing dependency, and so what we did was introduce new interfaces consisting of all the methods our gateway classes contained (meaning these gateway classes would then implement the interfaces), and then we made it so that our use cases referenced these interfaces rather than having them directly depend on the gateways, essentially getting our dependencies in order by following the steps of the DIP..

Single Responsibility Principle:

- We can look at the very logic of the program and find evidence of how specific classes change based on solely one actor. Our program only has two real actors

which are participants and organizers, and a couple examples of how this principle works within our program are:

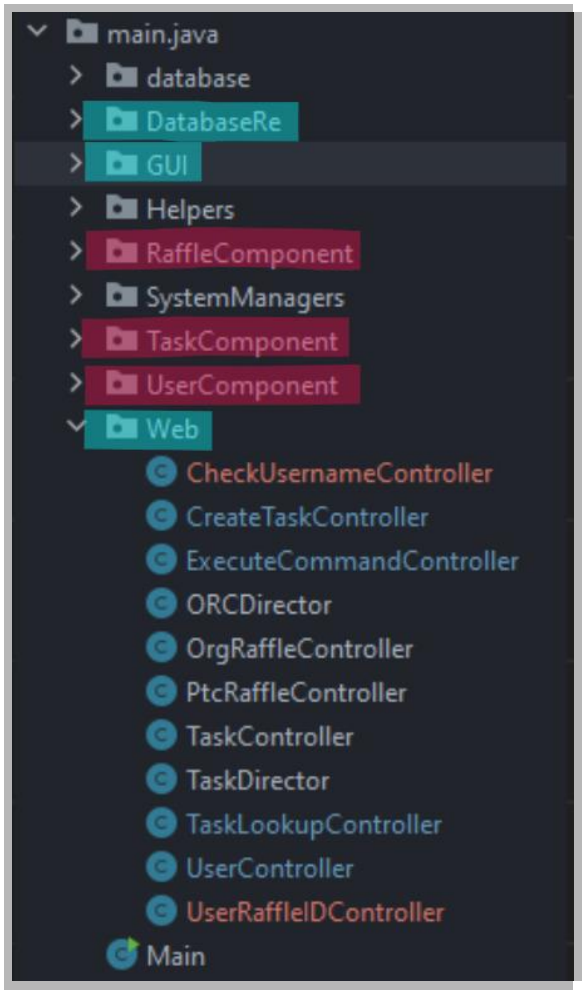
- When a participant decides to complete a raffle's task, we have made sure that the use case CompleteTask and the PtcRaffleController which handles it are completely free from the influence of organizers, since they can only be accessed by the program once a valid participant user ID is provided.
- Secondly, in instances where an organizer wants to change a detail about a raffle and participants are already signed up for it, we have made use of the observer pattern, so that when a change happens, the program is only really responding to the organizer that has staged the change, while the functionality required to answer to the participants so that their raffle information is also updated, resides inside the ParticipantRaffleEntity code, allowing the one use case in charge of handling the organizer's edit to solely answer the organizer.

***What can be improved*:**

- As described in the clean architecture portion of this document, we would have loved to implement a facade design pattern in order to handle what has seemed to be a violation of the SRP in our system manager classes ever since phase 1. Whenever we want to hook up the functionality of the GUI to the system managers (web component), these managers end up calling multiple functions that trigger multiple entities, and so while there are a couple layers between these managers and our entities, these classes technically respond to more than 1 actor (responding to the User, Raffle and Task entities), and despite the one true source for change for these managers would be the user and their respective needs, a change in any of the 3 mentioned entities will end up having a domino effect until it gets to these managers. Getting to control our controllers for this phase 2 has made this violation become a little less impactful, since a lot of responsibility has been passed from the system managers to the controllers, which is what drove us to the decision to not try to solve this issue by using a facade pattern, given the smaller scale of the program.

Packaging Strategies

- Our phase2 code makes use of the packaging by component strategy, which gathers together all the business logic of related parts of the program into a single component (labeled with “component” at the end of the package name), while leaving the things like controllers , and details such as the user interface and the database, separate on what we have labelled the web component and the and the GUI component.
- This packaging strategy is perfect when it comes to the organization of the code in our case, as it separates everything into logical sub-pieces which make finding code and understanding what is done in a component very easy. We mentioned in our phase 1 design document how important it was to properly encapsulate packages by making good use of the java class access modifiers, which we have done in certain places (but mostly in methods rather than classes), meaning we have some sort of partial encapsulation which is fine but doesn’t really let us take full advantage of the benefits of proper encapsulation such as independently working on specific packages and more.



Design Patterns

Observer Design Pattern:

- Due to the publisher-to-subscriber nature of how raffles work by having participants sign up to raffles created by organizers, and because of how these raffles have some attributes being subject to change, we needed a way to let the raffle objects of participants get notified (and updated) whenever an organizer changed something. Our observer pattern has the OrganizerRaffleEntity acting as the publisher and the ParticipantRaffleEntity objects acting as subscribers. Inside an instance of organizerRaffleEntity, we store the userId of the participants enrolled to that raffle, and when the endDate of a raffle is changed/pushed back (which is the only change that an organizer can perform post-raffle-creation), the subscribers objects are built through their references according to the database's information, and have their update methods (called updateEndDate inside the ptcRaffleEntity) called, essentially executing the necessary changes on the observer's side. Note that as mentioned above in this document, when talking about SOLID, this pattern creates and highlights what might look like a violation of the SRP, since you have a single method in this OrgRaffleEditTaskUseCase that seems to be answering/performing changes for :organizers and participants, however, the delegation of the update to the ptcRaffleEntity class gets rid of this duality, which only arises because of the establishing of a publisher-to-subscriber relationship.

Builder Design Pattern:

- Originally, Organizer and Participant classes both implemented the interface User and the factory method was implemented to create Organizer and Participant objects. However, as we added more extensions, one of the additional functionalities is that we want to provide flexibility for the information that is required for the user to input when creating an account. Therefore, the builder pattern is implemented to achieve this flexibility. And instead of the two classes both implementing the user interface and a UserFactory creating the user, the two classes are now independent from each other and have their own builder class. However, because instead of setting the information as final, we want a participant to be able to change their information in the future so instead we omitted the builder implementation for participants.

Command Design Pattern:

- In our project, tasks are completion requirements to be a valid participant in a given raffle and can be of various types. Some examples of tasks could include: subscribing to a YouTube page, completing a survey or following a social media profile. The command design pattern could allow a user to complete various tasks in one click or touch, regardless of what the task specifies. This feature aids the user tremendously, as it shortens the time taken to complete tasks. This implementation would involve using a CompleteTask command object, which is invoked by a user pressing a button on their screen through a Button invoker object, and is received by the task object itself which then opens the relevant links provided by the organizer and updates the status of the task to complete, which is then in turn reflected by the participant being fully eligible to receive prizes within a raffle.

Progress report

Questions and Extensions

- Most of the questions we had relating to clean architecture decisions had been solved after Phase 1. We have thought of certain potential extensions that could be useful if this was to be used in the real world.
- Addition of two-step verification and other measures for security purposes for both participants and organizers. On the participant side, this is to reduce the flooding of the program by bots which reduce the odds of winning raffles, keeping the raffles fair. On the organizer side, this is to prevent fraud and ensure organizers have the resources to host these raffles, and are not defrauding users by pretending to be from certain organizations.
- Adding a payment gateway to allow people to access more premium raffles by paying a certain amount.
- When there are too many raffles to choose more, the users can let the program randomly (using a spinning wheel in the GUI) select a raffle for the user from the top raffles.

What has worked well so far

- Design geared around improving the user experience.
- New database structure has improved functionality tremendously.
- Email notifications have now been added so that winners are notified in real time.
- Extra search methodologies (by raffleID, by organization) allow for widespread searches.
- Extensively tested Raffle, User and Task Use Cases using multiple tests.

Project Accessibility Report

- Equitable Use - Our giveaway application protects privacy as Organisers cannot see Participants details in the program. It provides the same means of use for all users. It does not take any extra personal information (religion etc).
- Flexibility in Use - Users can work at their own pace (adaptability of use). They are given a variety of choices throughout the code (to make / delete multiple tasks).
- Simple and Intuitive Use - Well formatted GUI with big and fonts. There is a good contrast between colours and use of simple English (easy to comprehend).
- Perceptible Information - We could implement this as an extension in the future by adding a dark mode / high contrast mode. This would promote clear and easy perceptibility in the program.
- Tolerance for Error - Minimises errors by asking the user if they are sure through pop-ups before important functions are carried out. Specific errors are also displayed when they occur (so that they can be reversed).
- Low Physical Effort - Minimum movement is required to use our program. Cannot be applied to our program mainly because we just need to open a link in order to stand a chance to win a raffle.
- Size and Space for Approach and Use - All buttons of GUI remain within reach and easily accessible. Our program can be operated from any computer while the user is in a comfortable position.

Target Demographic

Our program would be marketed to all adults, mostly the ones who are interested in gambling. If we begin taking money from users, then by law, children will not be permitted to use our program. Generally, the working population will have money to pay to join a raffle.

Demographics that are less likely to use our program

Unfortunately, our program can only be used by adults as if users pay to enter raffles, this program can be considered a gambling application. Children are not allowed to gamble by law. Other people who have low disposable income may not want to waste their limited funds.

Individual Summaries

- Michele:
    - **Phase1:**
    - Refactored the phase0 code revolving around the Raffle entity into structured clean architecture.
    - Worked on the use cases and controllers needed to allow organizer raffles to generate winners from within an organizer raffle object.
    - Added the ability to both add and remove tasks from an organizer raffle while updating the corresponding participant raffles (this ended up getting removed).
    - Added the ability to have multiple participants from a raffle complete tasks.

    - **Phase2:**

- Separation of OrganizerRaffleEntity and ParticipantRaffleEntity (no parent-child relationship). Pull request : Michele p2 by MicheleMassa802 · Pull Request #41 · CSC207-UofT/course-project-sedative-skyscrapers (github.com). (More specifically commits 27ce6a5 and a8bc8da). This was significant because it changed the structure of how the program had been imagined up to this point, refactoring to respond to this change involved revisiting numerous classes, going, from the raffle entities and use cases up to the system managers.
- Creation of the combined Raffle Controllers, we also touched on this in the major design decisions section. Pull Request: Michele p2 by MicheleMassa802 · Pull Request #41 · CSC207-UofT/course-project-sedative-skyscrapers (github.com). (more specifically commit 9322d1c)
- Update of use case access to the new database, which also included a fair bit of refactoring since now we could go back in to edit certain things inside our database, which changed how the use cases surrounding the creation of new raffles worked.
- New use case and its inclusion into the program to allow organizers to modify the endDate of an ongoing raffle, which involved notifying and updating their respective participants.

- Varun:
  - Phase 1: Wrote the Graphic User Interface for the software.
  - Phase 2: Wrote the presenters and worked on other feedback given by the TA in Phase 1 including adding back and exit buttons, decreasing code smells, etc.

  - Pull request 1: https://github.com/CSC207-UofT/course-project-sedative-skyscrapers/pull/29
    Committed all the GUI files for phase 1 in this commit. This included all the user pages, raffle pages, sign in and sign up pages.

  - Pull request 2: https://github.com/CSC207-UofT/course-project-sedative-skyscrapers/pull/55
    Made the changes that were suggested by the TA in Phase 2. This included adding back and exit buttons in every page, upgrading the search function to search by raffle name, id and organization name. Also split big classes into two or more classes to decrease code smells. For example, SignUpPage was split into SignUpPage, ParticipantSignUpPage and OrganizerSignUpPage.

- Aakash:
  - **Phase 1:**
    → Implemented System Manager classes (Organizer and Participant) and methods for use from GUI.
    Phase 2:
    → Refactored System Manager classes (Organizer and Participant) and methods for use from GUI. Made internal connections between the system manager and the different controller classes that store data into the database. Added additional connections for sending emails to the winners of the raffle.

- **Pull request:** (https://github.com/CSC207-UofT/course-project-sedative-skyscrapers/pull/54)
  –> Refactored code from phase 1 feedback to reflect the change of having only 1 controller class per entity. Added additional methods as per the GUI requirements and implemented code changes to reflect phase 2 extension i.e to send the chosen raffle winners email via the help of helper class.

- Shih-Hua:
  - **Phase 1:**
  - Refactored the User entity into structured Clean Architecture and implemented the Builder design pattern.
  - Implemented Use Cases and Controllers revolving around writing data into the database and retrieving data from the database based on the request of the System Manager classes.
  - **Phase 2:**
  - Created the combined UserController.
  - Implemented use cases to access data and provide data from the new database using dependency injection.
  - Implemented additional methods in the new UserController to fit the need of the SystemManager classes.
  - Refactored Organizer and Participant to align with the need of the new database.
  - Pull Request: https://github.com/CSC207-UofT/course-project-sedative-skyscrapers/pull/47

- Garv:
  - **Phase 1 -**
  - Implemented the Task creation controllers and use cases in order to adhere to clean architecture.
  - Refactored the Task entity to reflect the changes in how tasks are now implemented.
  - **Phase 2 -**
  - Consolidated the Task Controllers into one TaskController class which would call specific controllers depending on variables initialized.
  - Created a TaskDirector to help in building the objects and initialize variables.
  - Added Javadocs to all Task related classes
  - Made test cases for Task.
  - **Pull Request 1** - Garvfinal by garvmsood · Pull Request #46 · CSC207-UofT/course-project-sedative-skyscrapers (github.com)
    It was important as the controllers needed to be consolidated into one TaskController which would call the specific controller depending on variables initialized.
  - **Pull Request 2** - Changes made to Task and Task Use Case and Controller added by garvmsood · Pull Request #15 · CSC207-UofT/course-project-sedative-skyscrapers (github.com)
    This was used to implement the final parts of the command design pattern that is used in our program to execute commands.

- Nischal:
  - **Phase 1:**
  - Implemented Task entities and their respective use cases and controllers

- Designed and implemented the Command Design pattern for the completion of tasks by participants.
- Implemented the new method of completing tasks, changing it from a simple question answer textbox to now opening any type of URL via the java.web libraries
- Worked on integrating original database structure with the rest of the program
- **Phase 2:**
- Creating the combined task controller from the multiple task controllers in phase 1, due to the change in design decision to adhere better to clean architecture.
- Integrating various task use cases and components to interact with the redeveloped database and its corresponding interface.
- Creating the SendEmail helper that notifies winners in real-time via email, informing them of prize wins and raffle victories
- Refreshing the design document to reflect phase2, creating new presentation
- Pull Request 1 : Addition of the Email Notification Sender with additional libraries, important because it's the main method by which winners are notified of winnings and is the final step in the program. https://github.com/CSC207-UofT/course-project-sedative-skyscrapers/pull/42

- Khushaal:
  - **Phase 1:** designed and created the database. The data is stored in 4 tables in the form of CSV (later may shift to SQL) in a way to minimize the data stored by preventing repeated storing of data. The classes in database can be segregated into three types:
  - 1) Opens the file (E.g. GetFileToAppend, DataMiner)
  - 2) Reads the data and send it to evaluator (DataExtractor,)
  - 3) Evaluates the information as requested for a specific function and returns (GetTaskDetails, JoinUserToRaffle)
  - The reverse order is followed when we want to save the data to the database. I still have to make different directories for each segregation.
  - **Phase 2:** completely redesigned the database. Shifted the database to MySQL and the database now runs on AWS server. Linked the database with the program using Java MySQL connector. The database is divided into two *packages* now. Each package has separate classes depending on entity and whether the data is being accessed or uploaded.
    - *TalkToDatabase*: These classes set the connection to the database. And, contains Configuration constants of the database
    - *Modifiers*: These classes modify by getting, updating or accessing the data to/from the database. These methods get the Queries from Query classes in the main dir.
    - *Main Dir:* The main directory contains the AccessData.java and ProvideData.java which implements the DataAccessPoint and DataProvidePoint Interfaces, to make the Dependency Injection possible.
  - **Two Important Pull Request Links**
  - https://github.com/CSC207-UofT/course-project-sedative-skyscrapers/pull/38
  - https://github.com/CSC207-UofT/course-project-sedative-skyscrapers/pull/37