Sedative Skyscrapers - Design Document

Updated Program Specification

Domain: Giveaway application

Brief Description:
Online raffle platform allowing for brands/organizers to put up items for prizes and organize raffles to have other users (participants) complete a set of actions (created by the raffle's organizer) to have a chance to enter a random draw in which the actions they completed turn into entries, and from the pool of entries submitted by all participants, an amount of winners (as specified by the raffle organizer) is selected to win the prize.

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). 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, while participants should have the option to 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 (this would be a phase2 extension where the Main page also includes a page dedicated to notifications that a participant gets when taking part in a raffle).

*Note on "actions" completed by users: We plan to start off with simple actions like solving equations, or typing text displayed on the console, and as we progress through the project, we will try to expand the range of actions that can be performed.

Entities: # the main classes of the program
- Participant, Organizer
- Task
- Raffle

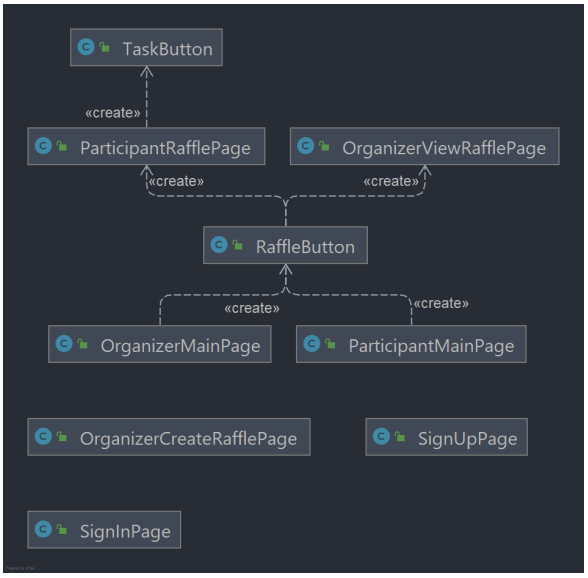
Use Case Classes: # some classes which invoke entities
- LoginUser
- CreateUser
- LoginRaffle (joining a raffle)
- CreateRaffle
- CreateTask
- AddRaffleTask (adding tasks to be completed to a raffle object)
- GetParticipant
- GetOrganizer

Controllers: # calling the use cases
- LoginRaffleController
- CompleteTaskController
- and more to match the use cases we have

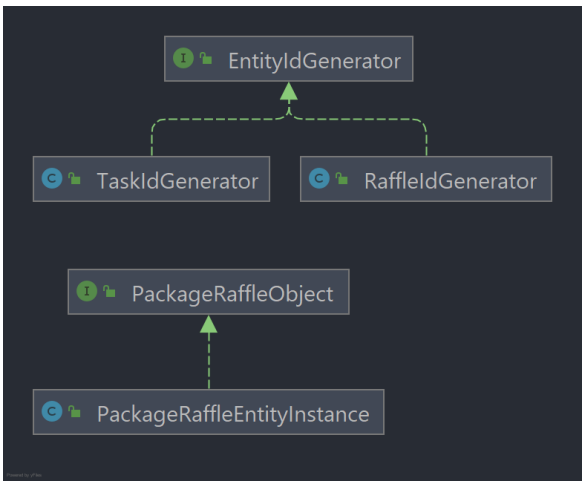
GUI:
-OrganizerMainPage
-SignInPage

Diagram of Code (UML DIAGRAM)

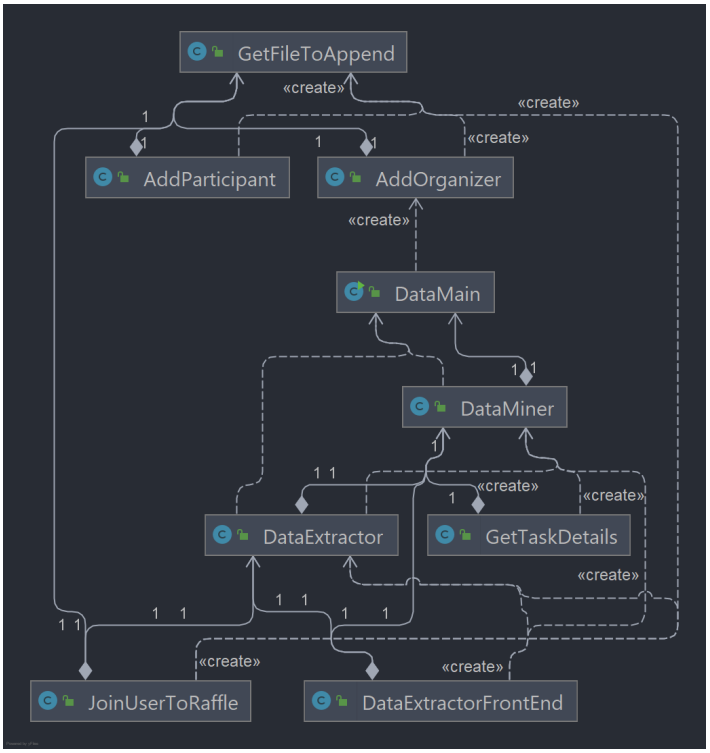
GUI:



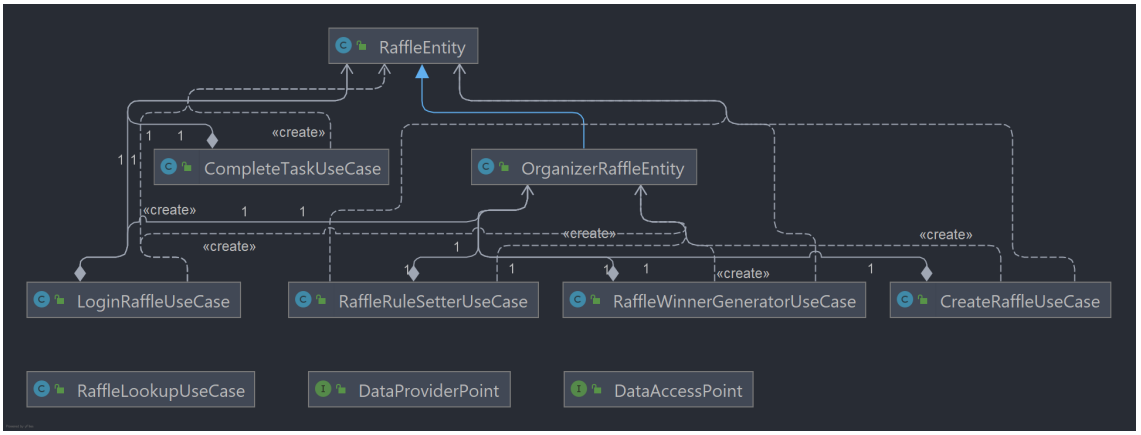
HELPERS:



DATABASE:

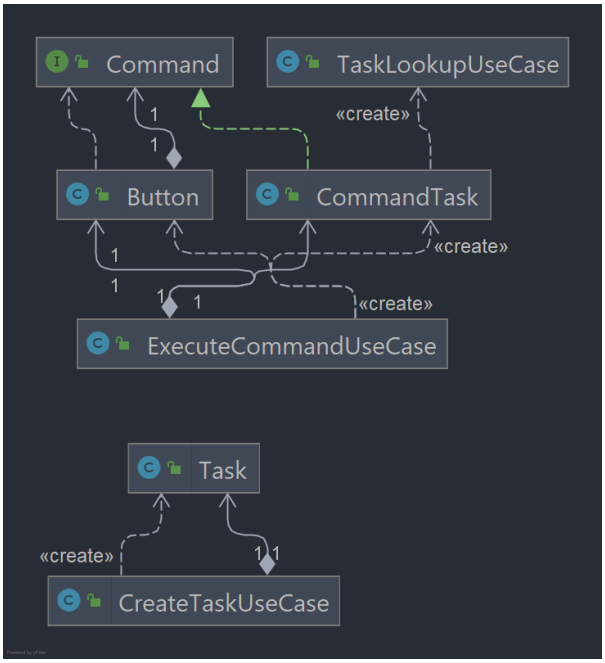


RAFFLE COMPONENT (PACKAGE)

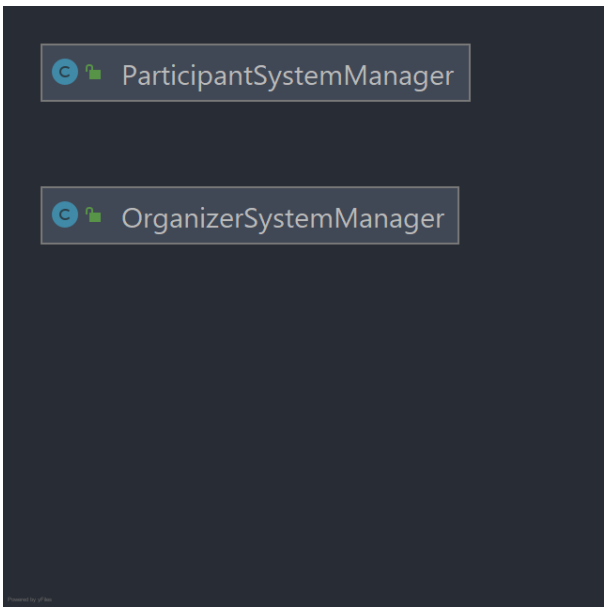


RAFFLE WEB

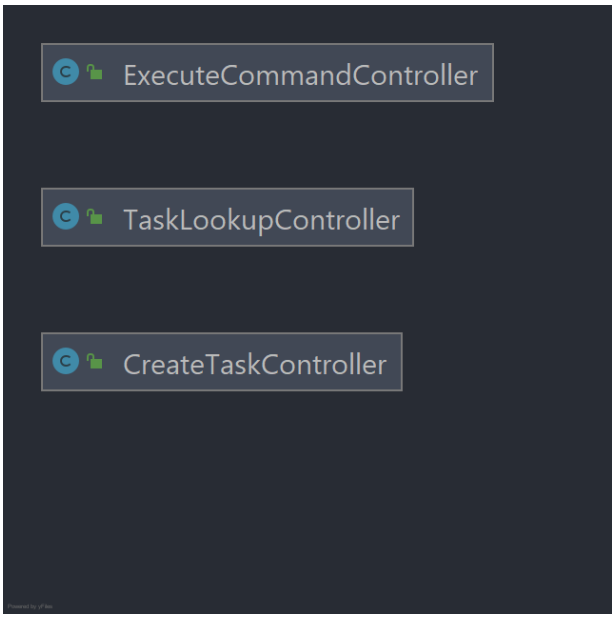
TASK COMPONENT



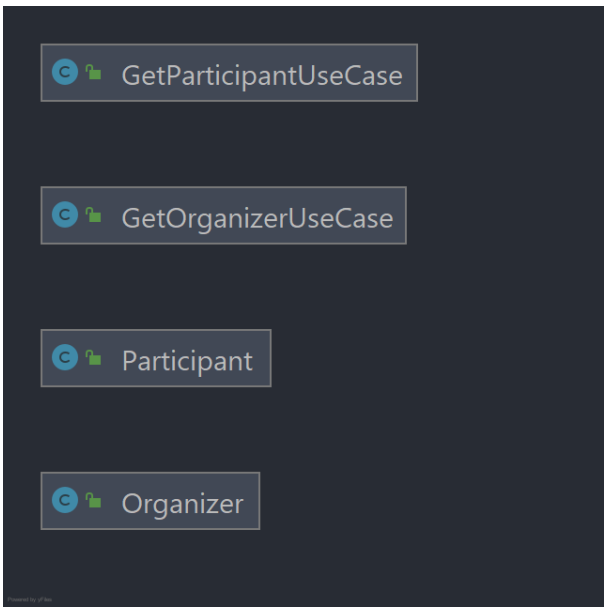
WEB



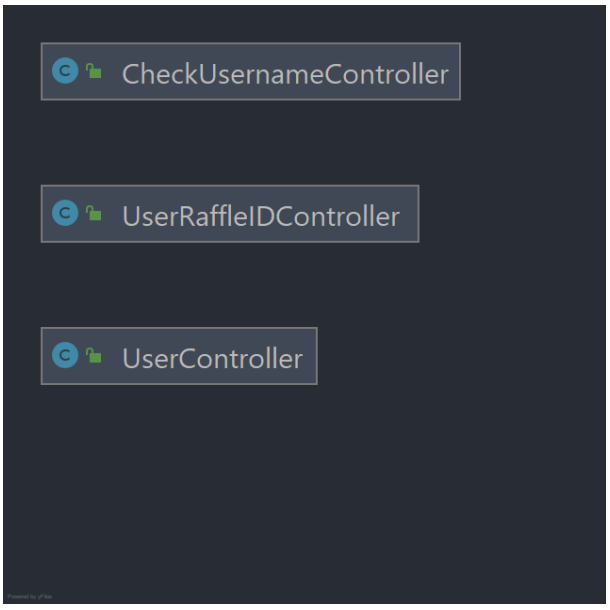
TASK WEB



USER COMPONENT



USER WEB



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 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 other subtrees, 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.

How our Software Adheres to Clean Architecture

- Consistent with clean architecture?
One of the things we take pride in about this program is the ways we worked our concept around in order to follow the dependency rule which is at the core of clean architecture, we wanted every possible place anyone could look at to reflect the dependency rule which is the core of clean architecture, and if the viewer goes back to versions of our code (on github) prior to the integration of the database, everything will be consistent with clean architecture. However, when the time came to integrate the database with the rest of the program, we ended up falling short on time, and decided to make our use cases / controllers just call the database extractor and updater functions without making use of an interface to connect these (you will see the interfaces for this connection in previous versions, but our issue was related to how the database itself was implemented which didn't allow for the database extraction and uploading methods to properly implement the interface), so this is the only place where a violation of clean architecture should be found.

Scenario Walkthrough demonstrating clean architecture and our shortcomings:
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

- Liskov Substitution Principle: raffles in our program can take the form of "standard" raffles of type "RaffleEntity", and "organizer" raffles, of the type "OrganizerRaffleEntity". If we were to define a program in terms of objects of type "RaffleEntity", the behaviour of such program would remain unchanged if we were to replace all of these "RaffleEntity" objects with "OrganizerRaffleEntity" objects, meaning that type "OrganizerRaffleEntity" is a subtype of "RaffleEntity" following the Liskov Substitution Principle. In simpler terms, the subclass "OrganizerRaffleEntity" in this case only adds to the behaviour of the masterclass "RaffleEntity" so that the extra functionality that raffle organizers need is present, rather than replacing or overriding anything about it.

- What can be improved: When it comes to solid, while we believe we have fulfilled it at every possible part of the program, but there is a part where it seems impossible not to violate the SRP, since 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. Furthermore, we want to take a closer look at how we divide our code into components, since for phase1 we have made an attempt at dividing by components, but it really was just for organization purposes, rather than proper encapsulation, since most classes are still marked as public.

Packaging Strategies

Our phase1 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 web components (labeled with “web” at the end of the package name). 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. One of our key improvements for phase2 is to properly modify the access modifiers of classes in their respective packages to implement proper encapsulation, since we know packaging is way more than just a way to organize files.

Design Patterns

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.

Progress report

- ### Questions

  At the moment our group is not struggling with anything related to the code of the project itself, we had to make compromises which will become obvious when carefully looking through the code and in fact, we have pointed these things out in our design document, but these compromises arise from struggles with timing rather than understanding of the concepts we have tried to make use of for this phase. However, already thinking about phase2, we do have a question, and it's related to encapsulation. We have made use of packaging by component during this phase, and so far it has made wonders when it comes to the organization and file structure, however we are struggling to understand how to actually make this packaging style useful, by this we mean, how are we to determine how to determine the access modifier of certain classes in order to make use of encapsulation?

- ### What has worked well so far

  Up to now despite having last minute issues with our database and github, we have had some very positive changes in terms of the structure of the main entities. Some examples of this includes getting rid of the Participable and Organizable interfaces, which just added unnecessary complexity to otherwise very simple and basic classes; and the split going from having a Raffle parent class and both an OrganizerRaffle and ParticipantRaffle subclasses to just having a RaffleEntity class which replaces the ParticipantRaffle from phase0, which acts as a parent to child OrganizerRaffleEntity, which replaces the OrganizerRaffle class from phase0, since this change has made the structure of the code related to this entity less convoluted, while greatly simplifying the amount of work to be done.

- ### Individual Summaries

- Michele: 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; and added the ability to have multiple participants from a raffle complete tasks (some parts of my implementation ended up

getting commented out because of limitations on other parts of the program, but the work can still be seen).

- Varun: Wrote the Graphic User Interface for the software.
- Aakash: Implemented System Manager classes and methods for use from GUI. Made internal connections between the system manager and the different controller classes that store data into the database.
- Shih-Hua: 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.
- Garv: Implemented the Task creation controllers and use cases in order to adhere to clean architecture and refactored the Task entity to reflect the changes in how tasks are now implemented. Also segregated files
  into their respective packages namely, web and component packages for Task.
- Nischal: Implemented the command design pattern for the execution of tasks, multiple task use cases and controllers, and major design and refactoring updates to the task class and its attributes/integration into the rest of the program. Specifically redesigned task to be better implemented and error free(from errors seen in phase0), designed and implemented the classes for use in command design pattern for taskCompletion and integration into web browser links.
- Khushaal: 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.