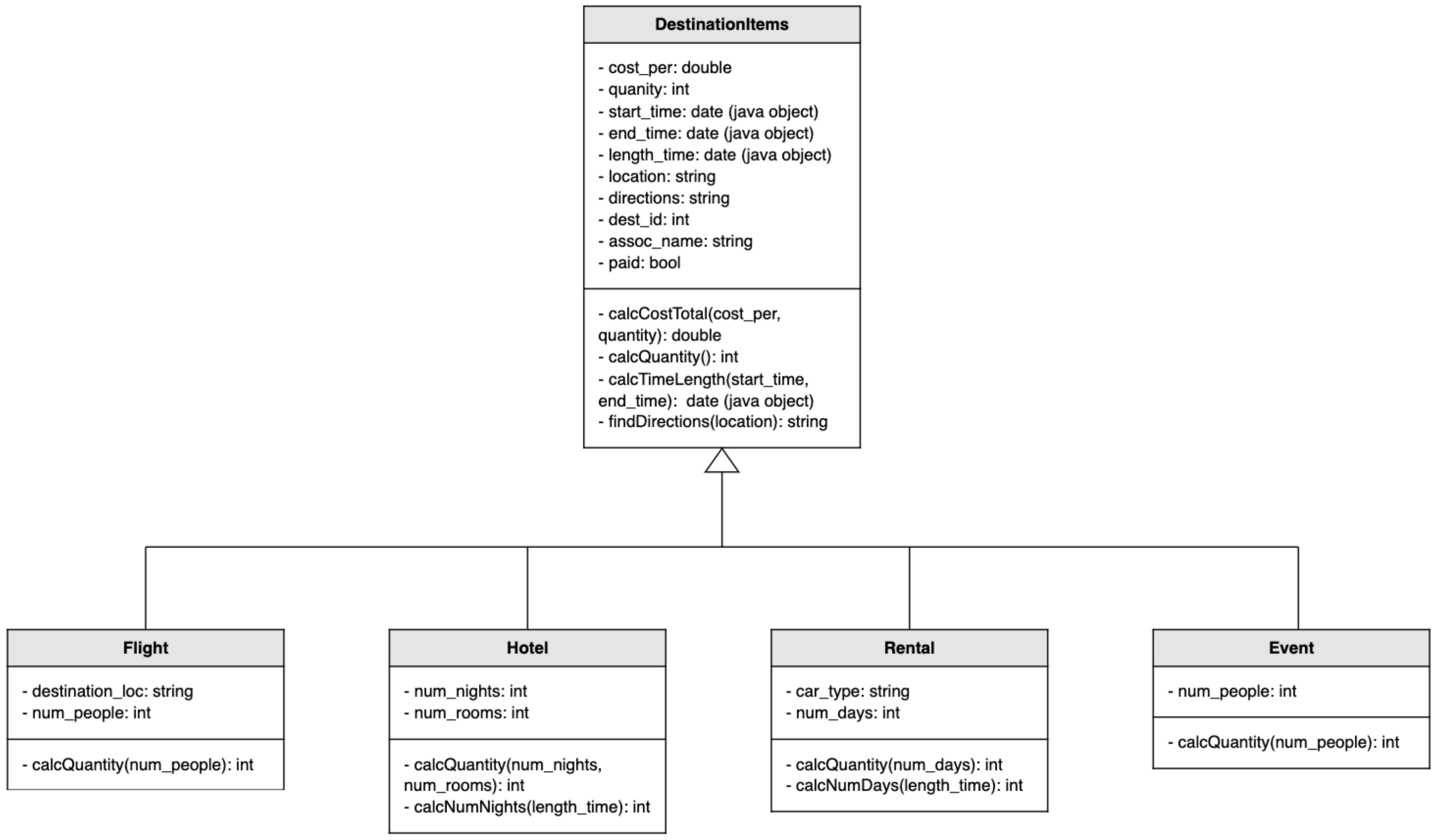
TravelGavel Design Pattern

Template



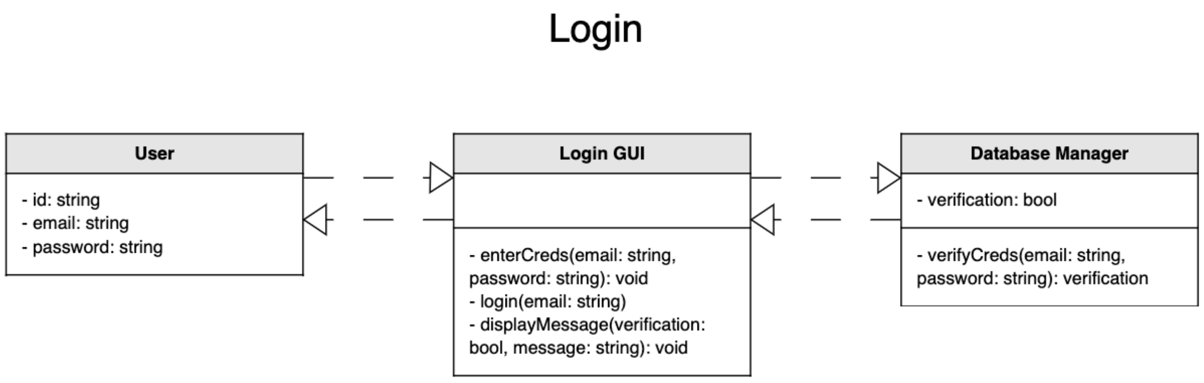
DestinationItems
- cost_per: double
- quanity: int
- start_time: date (java object)
- end_time: date (java object)
- length_time: date (java object)
- location: string
- directions: string
- dest_id: int
- assoc_name: string
- paid: bool
- calcCostTotal(cost_per, quantity): double
- calcQuantity(): int
- calcTimeLength(start_time, end_time): date (java object)
- findDirections(location): string
Flight
- destination_loc: string
- num_people: int
- calcQuantity(num_people): int
Hotel
- num_nights: int
- num_rooms: int
- calcQuantity(num_nights, num_rooms): int
- calcNumNights(length_time): int
Rental
- car_type: string
- num_days: int
- calcQuantity(num_days): int
- calcNumDays(length_time): int
Event
- num_people: int
- calcQuantity(num_people): int

Description and Justification

The template design pattern is used when multiple classes have similar functionalities and repetition of code. It allows common parts of an implementation to be shared while allowing subclasses to refine other parts. The template design pattern defines common parts in the superclass, which makes calls to abstract methods that deliver the other parts. An abstract method will define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. The subclasses can redefine certain steps of an algorithm without changing the algorithm's structure. Invariant steps (standard) are implemented in an abstract base class, while variant steps (customizable) are either given default implementations or not implementations.

The items associated with a destination (flights, hotels, car rentals, and events) share many of the same attributes and functionality. All destination items will have a cost per unit (which could be zero), quantity, start time, end time, length of time, location, directions, destination id (to link item to destination), name associated with the item, and whether it has been paid for. Something that does vary between the item types is what cost per unit represents and how quantity is calculated. For a flight, the cost per unit would be the cost per ticket and quantity would be the number of tickets. For a hotel, the cost per unit would be the cost per night, per room and quantity would be the number of nights times the number of rooms. For a rental car, cost would be the cost per day and quantity would be the number of days. For an event, the cost per unit would be cost per person and quantity would be number of people. This changes the implementation of how quantity is calculated and requires some dependencies for some of the item types' attributes and methods, specifically number of people needs to be tracked for a flight or an event, number of days need to be calculated and tracked for a car rental, and number of rooms needs to be tracked and number of nights needs to be calculated and tracked for a hotel. Additionally, a flight will have a destination location and a rental car will have a car type. The repetition in attributes and methods makes this section of the code well suited for the template design pattern, as the common code can be abstracted to a destination item superclass.

TravelGavel DCD

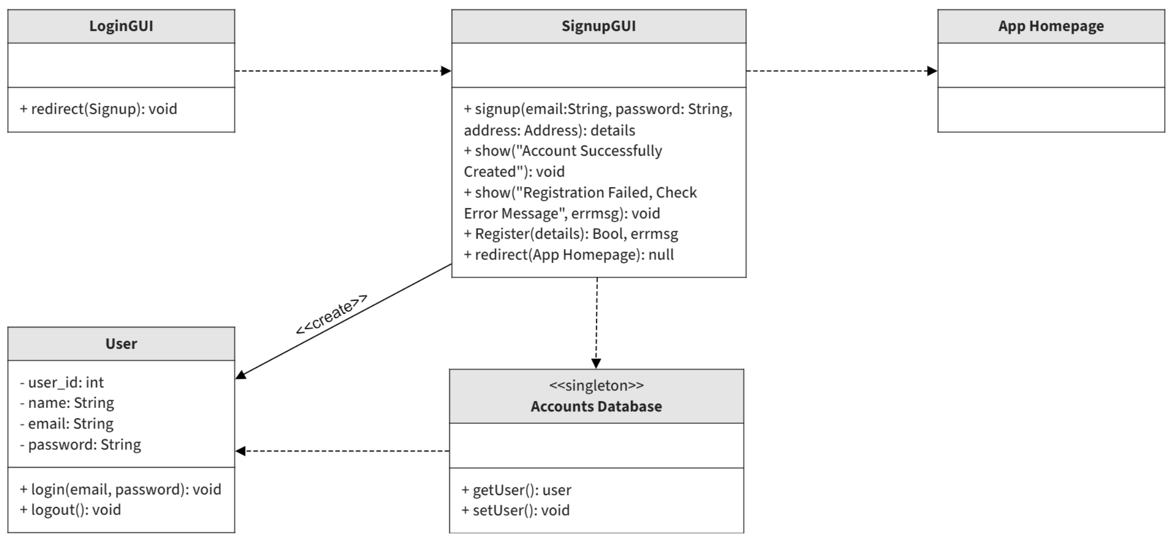

Login
User
- id: string
- email: string
- password: string
Login GUI
- enterCreds(email: string, password: string): void
- login(email: string)
- displayMessage(verification: bool, message: string): void
Database Manager
- verification: bool
- verifyCreds(email: string, password: string): verification

Description and Justification

Login:

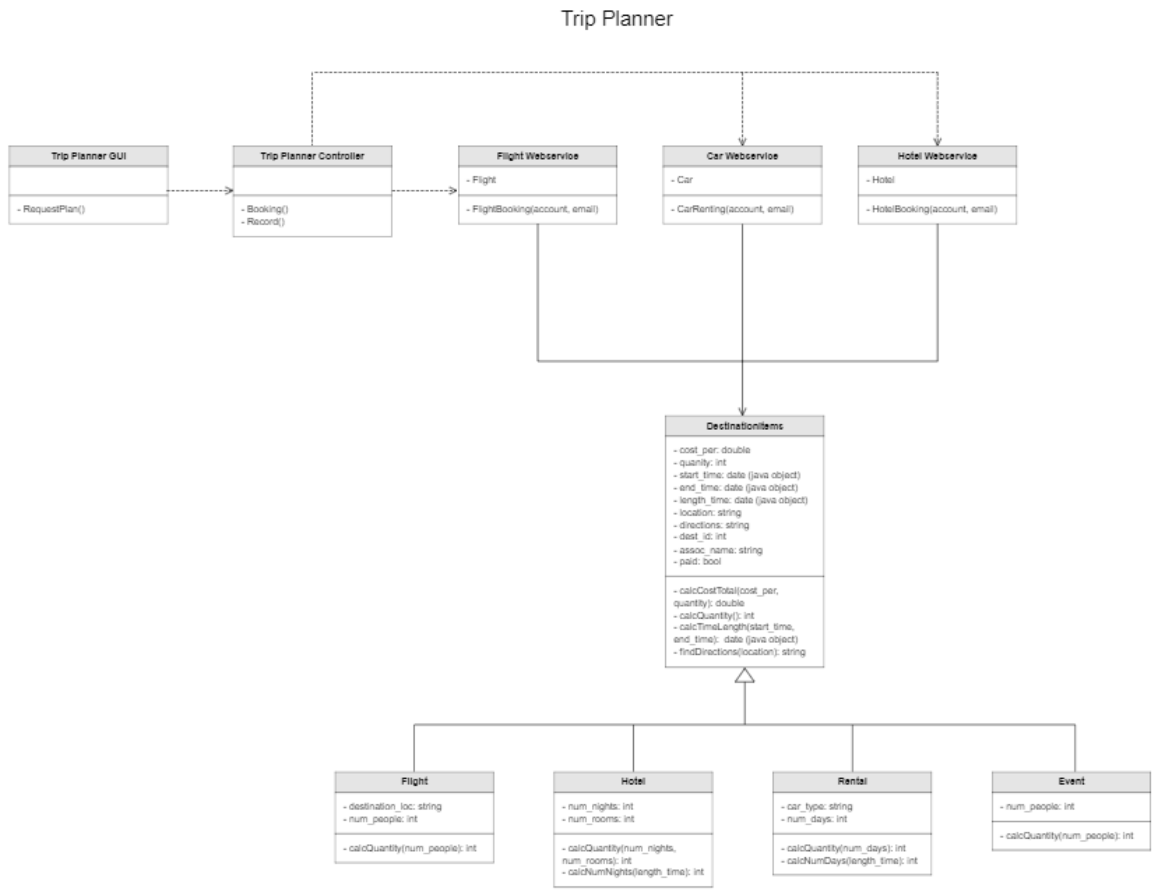
The classes needed to log in a user are: User, Login GUI, and Database Manager. In order to split account management into a separate DCD, this does not include signing in, which can be assumed to follow the account set up process. The User must be able to send information to the Login GUI, so that a user can enter an email and password to verify their credentials. The Login GUI is connected to the Database Manager, so that it can request the user's stored credentials. The Database Manager is connected to the Login GUI, so that it can be sent the user's stored credentials. The Login GUI checks the entered information from the User against the user's stored credentials from the Database Manager. Depending on the result, the user will either be logged in and receive a success message or they will receive a failure message.

Account Creation


LoginGUI
+ redirect(Signup): void
SignupGUI
+ signup(email:String, password: String, address: Address): details
+ show("Account Successfully Created"): void
+ show("Registration Failed, Check Error Message", errmsg): void
+ Register(details): Bool, errmsg
+ redirect(App Homepage): null
App Homepage
<<create>>
User
- user_id: int
- name: String
- email: String
- password: String
+ login(email, password): void
+ logout(): void
<<singleton>>
Accounts Database
+ getUser(): user
+ setUser(): void

Description and Justification (Account Creation):

The classes needed to create a new User account are User, LoginGUI, SignupGUI, Accounts Database, and the App Homepage user interface. This process is executed through the SignupGUI class' dependent relationship with the LoginGUI class, as SignupGUI would be inaccessible without this relationship. The SignupGUI class, once provided with name, email, and password information from the currently-anonymous user, establishes a dependent relation with the Accounts Database, which can furthermore create a new User instantiation only after having been passed the inputted information. Therefore, a special relationship directly between the SignupGUI class and the User class is established to signify that SignupGUI initializes the creation of a new User account, utilizing its dependency on Accounts Database in the process. Finally, a relationship exists where the App Homepage user interface is dependent on the SignupGUI class, as no transition between Account Creation and the App Homepage would occur without the redirect() method from SignupGUI. Trivial relationships not displayed within this diagram only include messages displayed to the user dependent on the state of the Boolean variable returned from the SignupGUI's Register() method.


Trip Planner
Trip Planner GUI
- RequestPlan()
Trip Planner Controller
- Booking()
- Record()
Flight Webservice
- Flight
- FlightBooking(account, email)
Car Webservice
- Car
- CarRenting(account, email)
Hotel Webservice
- Hotel
- HotelBooking(account, email)
DestinationItems
- cost_per: double
- quanity: int
- start_time: date (java object)
- end_time: date (java object)
- length_time: date (java object)
- location: string
- directions: string
- dest_id: int
- assoc_name: string
- paid: bool
- calcCostTotal(cost_per, quantity): double
- calcQuantity(): int
- calcTimeLength(start_time, end_time): date (java object)
- findDirections(location): string
Flight
- destination_loc: string
- num_people: int
- calcQuantity(num_people): int
Hotel
- num_nights: int
- num_rooms: int
- calcQuantity(num_nights, num_rooms): int
- calcNumNights(length_time): int
Rental
- car_type: string
- num_days: int
- calcQuantity(num_days): int
- calcNumDays(length_time): int
Event
- num_people: int
- calcQuantity(num_people): int

Description and Justification (Trip Planner):

To plan a trip, the additional classes needed are: Trip Planner GUI, Trip Planner Controller, Flight Webservice, Car Webservice, and Hotel Webservice. For this process the User requests to plan a trip through the Trip Planner GUI. The Trip Planner GUI is connected to the Trip Planner Controller, which connects to the Flight Webservice, Car Webservice, and Hotel Webservice to book flights, cars, and hotels per the User's request and record those bookings. The webservice classes connect to the Destination class, which in turn connects to the appropriate subclass, depending on the webservice class that connected to it, and retrieves the appropriate information and attributes for the intended object. This information is then sent back along the proper pathways to be recorded in the trip. Depending on the nature of the request, the user will be able to book and record a hotel, flight, or rental for their vacation.