PM3: High Level, Low Level, & User Interface Design

By Sachit Tripathi, Nathan Roberts, Ilya Mruz, Tikki Cui

High Level Design:

In order to make our Mentor Mentee matching bot ByteBuddies, we will implement an event-based architectural pattern. This architecture pattern promotes the production, detection, consumption of, and reaction to events, which will be very beneficial for our bot.

The first event would be the event emitter. In this case, the main event emitter will be when a mentee is looking for a mentor. When this is done, an event will be created and passed to the event dispatcher. The event dispatcher is a central component that manages the distribution of all events upstream to event consumers downstream. Event consumers will be different mentors who match the criteria laid forward by the mentee when they made the original event.

A big reason this architecture pattern is beneficial for this bot is that the components are decoupled. A matching algorithm can be implemented which can help match mentors and mentees in the event dispatcher. The algorithm doesn’t need to know about the mentors or mentees directly, it just needs the relevant events.

Additionally, this method is very good in terms of scalability. If the matching algorithm needs to be updated, it can be modified without affecting the mentors and mentees. Additionally, if more matching criteria is to be added, it can be done easily.

Low Level Design:

There are multiple features within the ByteBuddies system where different low level design pattern features can be applied to enhance the interactions between different modules depending on the roles of Mentor, Mentee and Admin roles or the event based system for the API calls used for a web based application.

Firstly, the creational design pattern family can be used in the ByteBuddies mentor matching system which encompasses the Builder patterns. The Builder Pattern is utilized during user registration, offering a flexible and step-by-step approach to constructing user accounts. This is particularly beneficial as user accounts require varying details based on roles, and the Builder Pattern facilitates the creation of instances with specific attributes, increasing code reuse and maintainability.

Secondly, the structural design pattern family involves the Decorator and Proxy patterns which can be applied for responsibilities of the different role classes. The Decorator Pattern dynamically adds responsibilities to users based on their roles, offering a scalable and flexible solution for managing user functionalities. This pattern is essential in the role system of ByteBuddies, allowing for the seamless addition of role-specific features without modifying existing code. The Proxy Pattern is also used for controlling access to resources, ensuring users have the necessary permissions and enhancing security. This is crucial for maintaining a reliable and secure system architecture.

Finally, the behavioral design patterns, including the Command, Observer, and Strategy patterns, can play a vital role in the ByteBuddies system. The Command Pattern encapsulates and parameterized processes such as mentor/mentee requests, allowing for easy addition of new request types. This enhances the system's flexibility and maintainability. The Observer Pattern facilitates user subscriptions and updates about events, promoting a loosely coupled system. This ensures that users stay informed about relevant events without tight dependencies. Lastly, the Strategy Pattern is employed for matching mentors and mentees based on different algorithms. This supports the system's adaptability to different matching strategies, providing a robust solution for diverse user preferences and requirements. Overall, the integration of these design pattern families contributes to the system's usability, reliability, and performance.

Used Pseudo Python Code to define module interactions for design patterns

Pseudocode:

Creational Design Patterns:
```
class UserAccount:
    def set_username(self, username):
        pass

    def set_email(self, email):
        pass

    def set_password(self, password):
        pass

    def build(self):
        pass

class UserAccountBuilder:
    def create_user_account(self):
        pass
```

Structural Design Patterns:
```
class User:
    def __init__(self, username, email, password):
        self.username = username
        self.email = email
        self.password = password

    def get_user_details(self):
        pass

class Mentor(User):
    def __init__(self, base_user, skills):
        self.base_user = base_user
        self.skills = skills

    def mentor_specific_functionality(self):
        pass

    def get_user_details(self):
        pass

class Mentee(User):
    def __init__(self, base_user, academic_goals):
        self.base_user = base_user
        self.academic_goals = academic_goals

    def mentee_specific_functionality(self):
        pass

    def get_user_details(self):
        pass

class Admin(User):
    def __init__(self, base_user, admin_privileges):
        self.base_user = base_user
        self.admin_privileges = admin_privileges

    def admin_specific_functionality(self):
        pass

    def get_user_details(self):
        pass
```

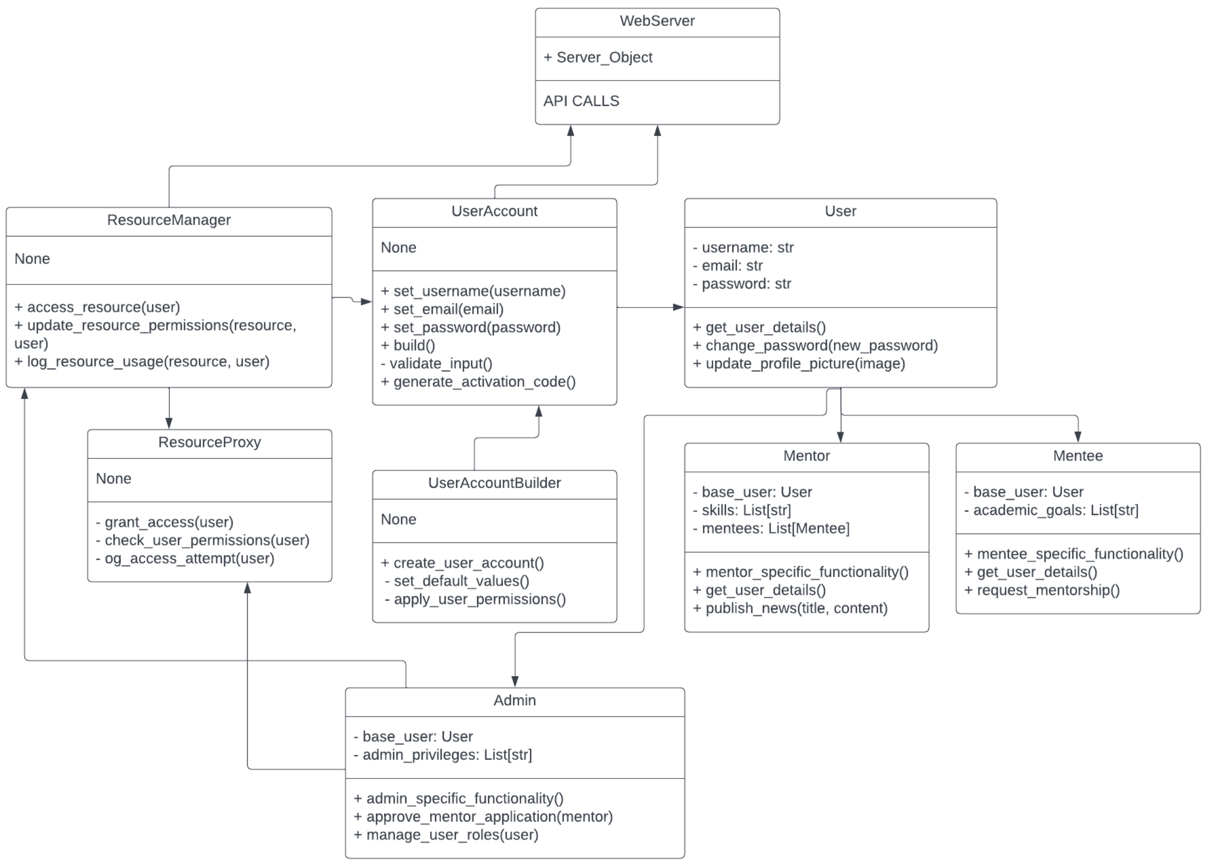
Behavioral Design Patterns:

```
class ResourceProxy:
    def grant_access(self, user):
        return ResourceManager.access_resource(user)

class ResourceManager:
    def access_resource(self, user):
        pass
class Event:
   def __init__(self):
      self.observers = []
   def attach_observers(self, observer):
      self.observers.append(observer)
```

```
  def detach_observers(self):
    self.observers.remove(observer)
  def notify_observer(self):
    for observer in observers:
    observer.update(self)
class User:
   def event_update:
      pass
class MatchingStrategy:
   def match_users(self,  user, pool_of_users):
      pass
class MatchingAlgorithm1(MatchingStrategy):
  def match_users(self,  user, pool_of_users):
      pass
…
```

Class Diagram:



PM3: High Level, Low Level, & User Interface Design

By Sachit Tripathi, Nathan Roberts, Ilya Mruz, Tikki Cui

Design Sketch:

PDF attached

Our primary focus was to create a user friendly and very intuitive user interface. Registering, whether you are a mentor or mentee, is very simple and easily seen via the labels. A mentor is able to add numerous mentees with a simple table. Both the mentor and mentee have a calendar where they can see any future meetings. The mentee has handpicked recommendations based on their profile and preferences. Additionally, the mentee is able to access our interactive learning courses and our forum.