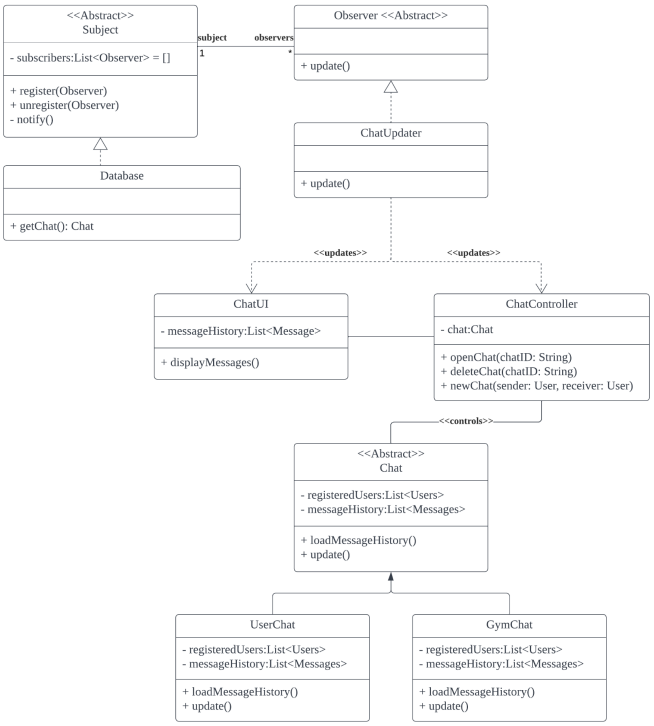
Design Problems/Patterns

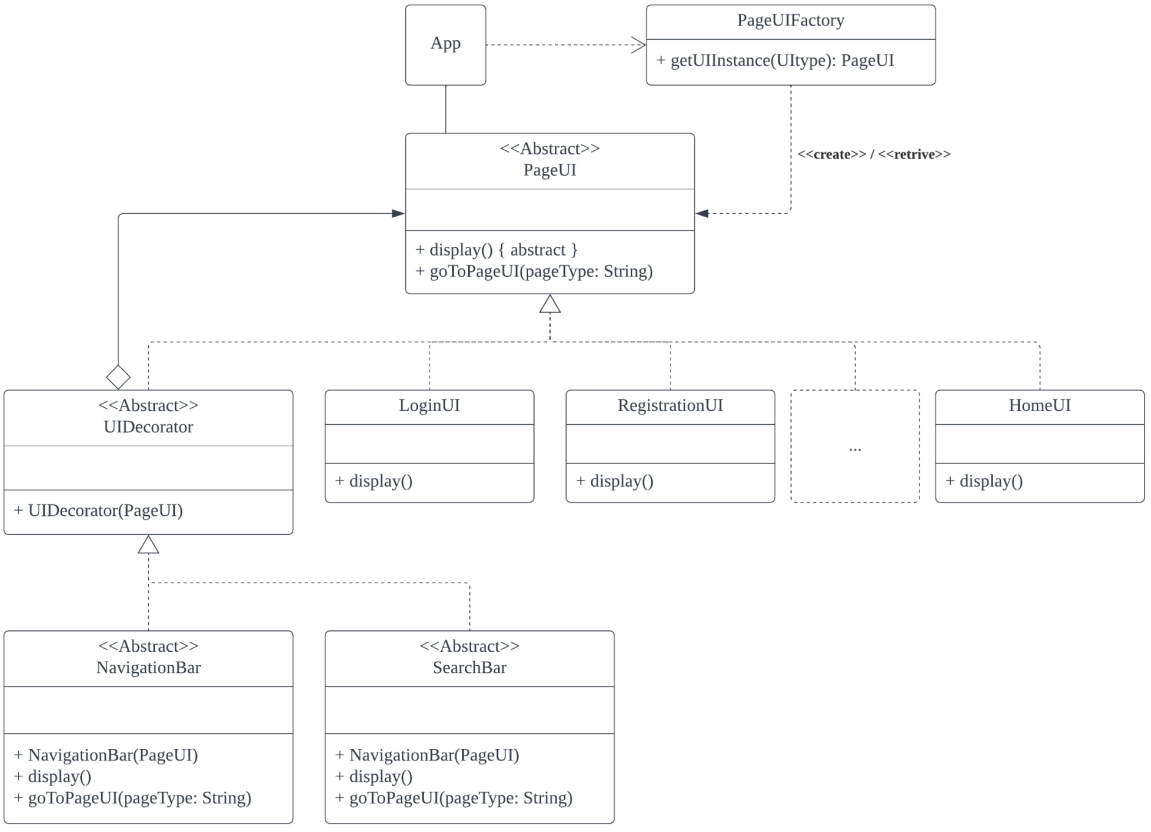
1. Observer + Facade pattern

- Description:
  - “Event driven” software design where observers registers to a subject and receives updates/notification when subject data is updated
  - Facade class includes the classes whose methods are required and runs the methods to complete a function
- Scenarios:
  - User chat page and gym dicussions automatically updates when any member of the chat sends a message
- Solution:
  - ChatUpdater class class is a observer and also a facade
  - Chats and gym discussions databases are subjects
  - ChatUpdater are automatically registered to the database upon creation
  - Gym discussions are registered to the database based on user’s decision to like/follow the gym
  - When gym discussion or chats are updated with new messages, database will trigger push update to ChatUpdater, and ChatUpdater will trigger methods required on ChatController and ChatUI to retrieve latest data
  - User’s UI will reflect the updated message history
- Notes:
  - Other pages (e.g. workouts page, user page, gym page, etc…) updates during user browsing does not fall under observer pattern as user is required to manually trigger update to receive latest information
  - ChatUpdater will be deregistered upon logout or deletion of app
- Class diagram:



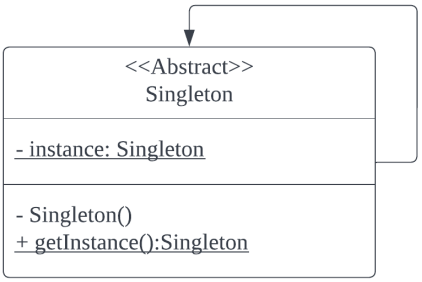
2. Factory pattern + Decorator

- Description:
    - Dynamic creation of objects through a factory object, creating instances of objects implementing a abstract class
    - Attach additional responsibilities to an object dynamically without affecting other objects
- Scenarios:
    - Switching between UI pages
    - Each UI page may not always need a component like navigation bar
- Solution:
    - Each UI page are implementations of a UI abstract class which can be created or called by FactoryUI object
    - Use PageUIDecorator to add the components dynamically when needed.
- Notes:
    - NIL
- Class diagram:



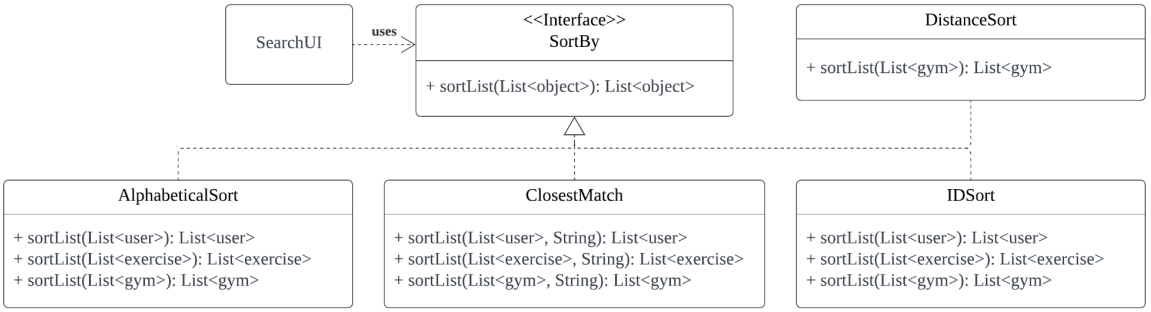
3. Singleton pattern

- Description:
    - Design pattern that restricts the instantiation of a class to a singular instance
- Scenarios:
    - Only one instance of each boundary and control classes required for program
- Solution:
    - Define every boundary and control classes according to singleton structure
- Notes:
    - Entity classes should not be singleton as multiple instances of the object will be created
- Class diagram:



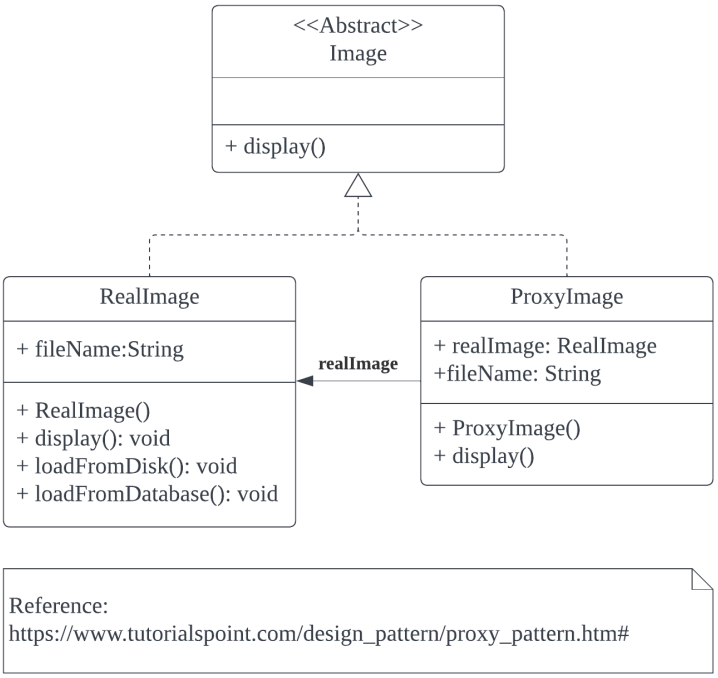
4. Strategy pattern

- Description:
    - Allow for interchangeability of algorithms or objects with similar functions
- Scenarios:
    - Entity lists (users, gyms, and exercises) can be sorted by a variety of manners depending on user preference
- Solution:
    - SortBy interface is referenced to to represent and call for specific sorting algorithms usable by other classes
    - Multiple methods under the same name can be created to handle the different type of lists with different objects
- Notes:
    - DistanceSort only applicable to gyms
- Class diagram:



5. **Proxy pattern**

- Description:
    - Use of a lightweight proxy object to represent an expensive object, instantiating the expensive object only when required
- Scenarios:
    - Images to display during runtime may slow operations if all images loaded upon startup
- Solution:
    - Use of proxy object to indicate image location or access which will load the image during loadImage method call
- Notes:
    - NIL
- Class diagram:



SOLID Principle Compliance

1. **Single Responsibility Principle**

- Description:
    - There should never be more than one reason for a class to change.
    - High cohesion
- Example:
    - Each UI class displays its respective component
    - Each control class is only responsible for the entities under it's control

2. **Open-Closed Principle**

- Description:
    - Module should be open for extension but closed for modification.
    - When new subclass is added, changes to the superclass should not be needed.
- Example:
    - PageUI class holds an abstract method for subclasses to implement.
    - SortBy interface is extendable by adding more sorting methods

3. **Liskov Substitution Principle**

- Description:
    - Subtypes must be substitutable for their base type without disrupting the behaviour of the program
- Example:
    - HomeUI, SearchUI and ChatUI classes are subclasses of PageUI.
    - RegisterUI and LogInUI classes are not subclasses of PageUI class as they do not need separate pages.