**Design Problems/Patterns**

Observer pattern

* Description:
  + “Event driven” software design where observers registers to a subject and receives updates/notification when subject data is updated
* Scenarios:
  + User receives notifications from chats and gym page threads, based on membership or subscription respectively.
  + User chat page automatically updates when any member of the chat sends a message
* Solution:
  + Gym page and chats are subjects and users are observers
  + Users are automatically registered to the chat if user is a member of the chat
  + Users are registered to a gym page based on user’s decision to subscribe
  + When gym page/chats send notification to user, user will trigger push notification based on user preferences
* 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
* Implementation progress:
  + Completed

Factory pattern + Strategy pattern

* Description:
  + Dynamic creation of objects through a factory object, creating instances of objects implementing a abstract class
* Scenarios:
  + Switching between UI pages
* Solution:
  + Each UI page are implementations of a UI abstract class which can be created or called by FactoryUI object
* Notes:
* Implementation progress:
  + In progress

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
* Implementation progress:
  + Completed

Facade pattern

* Description:
  + Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system
* Scenarios:
  + App startup requires the creation of relevant control classes for app function along with other initialization
* Solution:
  + StartUp facade object can be used to run necessary startup initializations
* Notes:
* Implementation progress:
  + Completed

Decorator pattern

* Description:
  + Attach additional responsibilities to an object dynamically without affecting other objects
* Scenarios:
  + Each UI page may not always need a component like navigation bar
* Solution:
  + Use NavigationBarDecorator to add the components dynamically when needed.
* Notes:
* Implementation progress:
  + In progress

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:
* Implementation progress:
  + Completed

**SOLID Principle Compliance**

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

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.

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.