Object-Oriented Design II Principles

Team A

For this exercise, analyze your term project design against at least three of the principles discussed in the *Object-oriented design II* class listed below. Your analysis should identify at least one area in your design where you believe you have high adherence to a principle. Describe the evidence supporting that assessment. You must also identify at least two areas of your design where you think you could improve its adherence to the principles**.** Describe the design changes you will make to achieve better adherence.
This exercise is worth two exercise points. An extra exercise point will be awarded for each principle you analyze beyond three.

Controller

The controller principle is when a class (in object oriented programming) manages or directs the flow of data between two entities. In WebCheckers, there are two notable classes that act as controllers. One is the GameCenter class and the other is the PlayerLobby class. GameCenter deals with the logic flow of games. This entails starting a game, adding and removing users depending on whether they are playing games or not, logging users in and out of games, keeping track of player matches and online players. GameCenter can also check if a game exists and get the board that is being used in a game. GameCenter is used to touch aspects of games without going to the model class, hence it acts as an interface between the ui and model tiers. This is good model-view-controller design because the user never interacts directly with the models, but rather is looped through a controller first. GameCenter can be improved as a controller by perhaps pushing some of the user logic into another controller, like player services. GameCenter does need to know who is playing a game, but perhaps some of the adding and removing from lists can be pass through the PlayerLobby controller instead.

PlayerLobby is another example of a controller. It is used to handle and store user data. It does so by taking information from the ui (user) and stores it in databases for access by other classes. This is good compartmented behavior, segmenting the ui away from the model tier. PlayerLobby handles adding users to the user accounts (valid usernames only), setting users logged in and logged out and seeing if users exist. There is currently a fair amount of redundancy between the PlayerLobby and GameCenter classes, and that is a point of weakness with the controllers. These classes should be looked at and revised for useability and the necessity of redundant methods should be analyzed. Overall we do a pretty good job of using controllers the way they're meant to be used, and we have a clear delineation between the ui, model and controller tiers. However, both GameCenter and PlayerLobby encroach on each other’s space a little, and that should be corrected.

Polymorphism

Not used - could improve.

Liskov Substitution

Anthony Cianfrocco - The Liskov Substitution principle in object-oriented programming means that any subclass that overrides a method of its superclass must have the same pre- and post-condition requirements. The subclass method’s pre-conditions cannot be narrower than the superclasses and the post-conditions cannot be broader. In WebCheckers, GetSignInRoute is a good example where Liskov Sub is in high adherence. This class implements the Route class. The Route has a method called handle() which is used by GetSignInRoute and gets the home route and its attributes when called. The pre conditions on Route say that it should be initiated only when the URL for the specific route is called. The post conditions say that is returns the the contents to be set in response. The GetSignInRoute does exactly this.

GetHomeRoute’s adherence to this principle could be slightly improved. The pre conditions are satisfied, however the post conditions are lacking a little. The response for this class isn’t returning exactly what should be in the response as the Route class says it should. The GetHomeRoute returns the current user as a potential player when it shouldn’t, therefore breaking the rule. Also, GetGameRoute could be improved. It is not giving the right mode options to the response that should be sent for the game board. If this is fixed, then the response will be correct and will adhere to the post condition that Route requires.

Open/Closed

The Open/Closed principle from SOLD states that classes should be open for extension but not modification. This principle is not currently used inside of our project, as the none of the entities in the model require inheritance. This could be used in the Square class. There are two different types forms an object could be. Each Square could be a white square or black square. More importantly white and black square have different associated behaviors.

If we were to incorporate the Open/Closed principle for the Square class, we would first define a Square class which contained all of the shared attributes and methods. Then we would create a Black Square class which would extend the Square superclass.

Later when we implement the game replay the Open/Closed principle could be used in order to create a PlayedGame class which extends the original Game class. Here the Game class would be extended in order to store the executed moves and render a players game board at any given move. These features would be extensions to the methods and attribute which already exist in the Game class.

Pure Fabrication

Merry Ren - Pure Fabrication describes the creation of classes that are not in the domain model but used for the purpose of achieving high cohesion and low coupling. PlayerLobby is a example of this since, technically, the data-structures contained in it can be directly written into the GameCenter.java creating one large class that encompasses all the mechanics of the site. However this greatly reduces the cohesiveness of the code in it, because the logic of the game is not largely related to the logic of the list of players. Then by the same logic, because so much of the program is contained in the one single application tier class, the same design also increases coupling since all of the classes in the UI tier functions use this class. Furthermore, including everything with the actual game and player sign-in both also exposes the underlying mechanics of our application to that UI tier with direct access. However, PlayerLobby.java mitigates this by separating that logic. It still allows UI to access requested material through the corresponding methods in GameCenter but the player related data itself will not interact directly with any of the Routes in UI. Thus a key advantage of this is that in future modifications, the methods related to player logic can be modified individually without having to worrying about modifying anything in the UI.

One way to strengthen this project is to also apply pure fabrication to the list of games (a hashmap) and the list of opponent pairs (also a hashmap) that currently reside in the GameCenter class. While these data structures still relate to game logic, they will not likely need to be accessed every call to an instance of GameCenter. Hence separating out this logic to PlayerLobby or another new class would promote good design practice.