15-214 Homework 4

Muhammad Ahmed Shah
mshah1

Rationale

This design achieves **modular continuity** in special tiles through the use of the strategy and template pattern by abstracting the Special Tiles behind the `SpecialTile` which is an abstract. This would allow the integration of special tiles implementing a different effect to be integrated with no change to the rest of the implentation. Most classes keep the logic contained within them so other clases can use them as a blackbox. For example the move class that enforces some rules on how the tiles are placed, may be changed to add new rules or remove some rules if we change the game but the GameBoard class would not be effected. This would allow the implementation to be altered and new classes that provide the functionalities to be added with no changes to the logic in other classes as long as the new classes continue to uphold the behavioral contracts established.

This design achieves **modular decomposability** through adherence to the GRASP. The `GameEngine` class acts as the controller with its job being the point of entry into the code for the GUI. It recieves requests from the GUI and passes them on to the relevant classes and based on their responses notifies the GUI on any updates that it may need to make. This has made possible to create several small yet highly specialized classes that handle a small subset of problems. As an example, we can look at the `Move` class which specializes in tracking and verifying tile placements so it functionas as an information expert for the `move` objects. All the validation logic for a move resides in the `GameBoard` class so the `GameEngine` only needs to call `board.validateMove` to ensure that the move is a valid one. The task of verifying the validity of a word is bestowed wholly on the `Dictionary` class. No other class has any contribution to the decisions of `isValidWord` so modifying the rules there or adding new words will not influence the logic in other clases. The `GameBoard` also calculates the points by extracting them from the `BoardSquare`. Determining the score for a tile placed on a square is completely delgated to the `BoardSquare` so that it specializes in providing information of about a square on the board and its contents. The other classes need not keep track of the point values of the tiles and the multiplier of the square. All this handled by the `BoardSquare` class. In this situation the LetterTile class acts as an information expert for a particular tile and the BoardSquare class for the properties of a square on the board. Similarly the Person class is the information expert for a player. It holds its name and score so no other class needs to keep track of scores. they can just request it from the Person class. By separating and insulating the implementations of the functionalities offered by each class we make development highly localized so that future development can proceed on the individual components without the need to make large changes in the modules using these components.

The downside of decomposing the problem into smaller modules and using each of them almost like a black box is that we need to assume the result of these mini modules be correct. This leads to a higher level of coupling between the `GameEngine` and the other classes while achieving high cohesion. Of course there are contracts which regulate these interactions but if a module violates the contract then there is a high probability that the error will leak into other classes as well leading to a low level of modular protection. This is the price that we must pay for the conveniences offered by having highly specialized modules. Like any assembly line practicing division of labour all proceses rely on the correctness of the preceding process since none of them has any insight into the larger product that they are producing. To overcome this it would require that the preconditions for every class and method be enforced rigourously through exceptions and conditions verifying the results of the previous modules. This design also employs small interfaces to achieve continuity and protection.

The `GameEngine` is the centerpiece of the design so different classes can function independently. So, for example, the `GameBoard` doesn't need to be aware of the player and vice versa.

The GameEngine class employs the **observer pattern** to integrate with the user interface. This allows flexibility in creating and adding new UI classes for the game. The GameChangeListener defines the interface that the GameEngine expects from every class that will provide UI functionality, how the class reacts to the notifications is totally upto it. However it can do so while being oblivious of the finer details in the actual implementation of the game. Moreover it opens up the possibility of having multiple UIs for the game by storing a list of GameChangeListeners in the GameEngine and notifications would be sent to all of them.