The Use of Patterns in Current or Future Designs

The Creator design pattern helps to lower maintenance and leads to low coupling. Our implementation utilizes two instances of the Creator design pattern. The “StopBoard” class creates and maintains instances of the “StopColumn” class, which in turn creates and maintains instances of “StopTile” class. Together these classes define the structure of the game board. Use of the Creator pattern in this sense lowers coupling and ensures that the flow of control between backend classes is tightly maintained, reducing the chance of unintended side effects from rendering and updating the board.

Polymorphism is utilized to provide efficient coupling between the board class and the computer players. Inheritance is used to abstract the functionality of the computer player classes. The parent class called “Comp” defines the core functionality of the computer players. There are two child classes called “EasyComp” and “HardComp”. The child classes use different criteria to determine which actions to take during the progression of their turn, and as such have differing implementations of the same core functionality. This level of inheritance allows the StopBoard class to treat each type of computer player as its parent class and leaves the details of implementation up to the child classes once instances of them are created for play. This abstraction leads to cleaner handling of the computer player objects during game setup and play.

Perhaps the biggest issue with the codebase is the high level of coupling between the UI and the back-end logic which implements the rules and current status of the game. The Information Expert design pattern could have helped to separate these concerns. An additional “Game Statistics” class could have been implemented to maintain the positions of each player’s runners and pieces, how many columns they have won, and what colour their pieces are, along with other relevant statistics. Such a class would reduce the size of the main board class, increase cohesiveness of the classes which it would have broken up, and provide a well-organized means of collecting and retrieving game data. This class would facilitate the extension of the user interfaces’ functionality and allow us to include a “game statistics” sidebar in the main game UI which would improve user experience and ease of use.

Additionally, the issue of an incohesive UI class could be further resolved through use of the “Command Holder” design pattern. Instead of letting the user interface class listen and respond to user actions, we would define separate classes for each relevant action which implement the Command interface. This way, user actions and the system’s response to them would be separated from the UI, and UI components which trigger user actions would implement the CommandHolder interface. CommandHolders will call the “execute” method of the associated Command class when their event is triggered and “execute” would pass the updated details to the game statistics class mentioned above. This approach would reduce coupling between the classes, increase cohesiveness of existing classes, and generally promote separation of concerns and good object-oriented design.

These changes would be a first priority in any future iterations of the project, so as to increase reusability and extendability of existing classes, making the project more organized and future maintenance easier.

Prepared by: Zahra Tasnim, Tyler Morgan (Group 7)