Pattern Design

Introduction

Our project is the recreation of the Can't Stop board game in a digital format. It includes several different features, like the ability to save and load the game, customize the number of players and player names, and the ability to play against two difficulties of AI players. In order to fit in such features, we had to use several different design patterns from the GRASP framework to create a game whose code is easy to maintain and build upon.

Design Patterns

Our code has many different classes, each interacting with one or more other classes to either change a display element like removing a runner and placing a marker, or update a record such as keeping track of how many markers a player has. The first GRASP pattern, *information exper*t was used when deciding how to maintain data-based information. For example, in order to place markers, the Game class would call setCurrentPlayerMarker() which would then move the responsibility to the Player object whose turn is in progress. The Player class would then store the marker coordinates in its own list.

We managed to achieve a level of *high-cohesion* and *low-coupling* for certain classes of code. Particularly the relation between the Game and Player class, these classes maintain player information upon requests from the GameWindow class, thus allowing them to be reused easily. But the GameWindow class suffers the most from low cohesion and high coupling as it handles all the player inputs, and commands for visual and data based changes. It is a *façade controller* that currently handles too many responsibilities. In future iterations, our focus would be to split the responsibilities of this class to at least three different classes to handle player inputs, visual changes, and data changes separately.

When designing the players, we had to design methods for two different types of players, i.e, Human players and Computer players. Computer players share many of the same data as human players but have methods to let them automatically play and make regulated decisions. Therefore, a polymorphism-based approach was used to handle this situation. A Player class is extended to implement HumanPlayer and IntelligentPlayer (computer) class. This allows the Game class to assign responsibilities to Player objects without it having to check for computer or human players. *Pure fabrication* is a pattern we used several times such as the creation of SaveLoadUtil and GridSquareData to improve and modularise the functionality of some features. Several classes like DicePanel and CombinationPanel were also created to lessen the load on GameWindow class.

These are some of the patterns we were able to utilize to create our code.

Future optimization

Our code is by no means perfect and complete. There are several ways we could improve our code to ensure a smoother gameplay experience for the user and add more functionality. As discussed before, the GameWindow class needs to undergo decoupling and tweaks to achieve high cohesion. Creating a new class to display and change the board would be a priority. A User-Operator class to keep track and

perform player events and a User Interface class to display interactable elements would also improve the cohesion-coupling of our code.

We could also utilize patterns like *Protected Variations* and *Indirection* to improve some of the logistics of data recording. Currently, the GameWindow class may make some changes by directly accessing a player object. New methods in the Game class may allow us to maintain low coupling and allow some level of hidden design between classes.