Ali Simbanegavi

COMP 3721

L. Ricker

December 12, 2020

Milestone 5

A plethora of changes were designed for the final version of this game. The redesign began with tackling some of the basic critiques received after Milestone 4 about the overall design quality. These included handling exceptions caused by inputting text where numerical input was passed in in the console, limiting the number of cards that players had in their deck, and removing redundant mapping of enumerated values to integers. Unfortunately, more complicated changes were challenging to implement, which highlighted the poor object-oriented capability that the original program had.

Primarily, the process to generify the creation of new tiles/pieces on the board presented a challenge. Previously, only Jewels and Turtles were needed, and both were created by simply extending an abstract Tile class. This became a challenge when walls, ice walls, and crates needed to be incorporated into the game. The Board initially accepted lists of Jewels and Turtles in its constructor to populate the board. This was a violation of the open-closed principle, and the Liskov substitution principle since it was not taking advantage of the fact that subclasses should be substitutable for their base classes.

The approach to mitigating this started with modifying the actual creation process for objects that represented parts of the board. Rather than simply using an abstract class and extending it, a TileFactory relying on a Tile interface was designed to handle creating these objects. This way, responsibility for the creation of the subclasses was delegated in accordance with the dependency inversion principle.

In addition to this, rather than simply defining the behaviour of subclasses of Tiles before runtime, object creation needed to be more dynamic. The solution for this was to use a TileDecorator in conjunction with TileFactory. In essence, the TileFactory would handle requests indicating if the desired tile was a Turtle, Jewel, Wall, or Crate. Once specified, it would pass the creation of each object on to the TileDecorator class. This way, the behaviour of each object explicitly guided its own creation at a low level. Classes extending TileDecorator to facilitate this were MoveableTile (for crates and players) and ColoredTile (for players and jewels). More subclasses like TransformableTile (for melted ice walls) and StationaryTile (for walls) could have been created with this method. By doing this, object creation would have been far more flexible and in adherence with the open-closed principle. An error was encountered when attempting to use this method to initialize new pieces as Tiles. It was not possible to declare a piece like a player’s turtle as a Tile, then instantiate it and call methods on it like setColor() that belonged to a subclass of TileDecorator. This error was unfortunately not surmountable during the time needed for implementation.

Another issue that was encountered during the redesign was changing the flow of responsibility within the Model and Controller. The hierarchy of information and logic flow was resistant to change, as classes like GameModel and GamePlay had responsibilities that were far too dense. Incorporating stop points to control the flow of information once the game started within the Model and Controller classes would have made changing interactions between the classes more feasible. Unfortunately, these packages were overly-intertwined and had dependency issues. The controller created low-level Model objects to use in the GameModel constructor, resulting in a deeply entangled Controller and Model. However the main loop of the program itself was in the Model, which then required the Model to order the Controller around. The Controller should never handle any data logic, and this muddied the responsibilities of the packages. Overall, this entanglement proved difficult to completely overhaul.

The introduction of the Observer design pattern was a partial solution to this quandary. The Controller and Model components are constantly relaying information in complex sequences between each other, such as prompting players for Actions and validating those requests across different classes. The flow of information is significantly streamlined in the Controller can prompt the user for a move, make a single request to the Model to validate this move, and then execute the move by notifying the Model and view to update themselves. The primary observer in the Model would be the Board, which held the configuration of the pieces and was the reference for any validation related to newly requested moves. The management of the information exchange between the Model and Controller classes presented the biggest challenge in terms of updating the program.

Overall, the final version of the game was exceedingly challenging to implement based on the previous version. A complete overhaul of the game was needed, and restarting the creation of the program was not feasible to complete alone within the timeframe between receiving feedback and needing to deploy a completed redesign. Besides redesigning the components that were resistant to change, implementing new features was also laborious and ultimately unsuccessful. Clearly, this highlights the issues that were present in the original design, as a true object oriented program shouldn’t have been this resistant to change. The dependencies were overwhelming to separate, subclass creation as rigid and didn’t facilitate principles like open-closed and Liskov substitution. Without more time and more careful planning in the design phase at the beginning of the project, creating a robust and flexible program based on the initial setup was not possible.[[1]](#footnote-1)

1. Drafts of classes for redesign included in submission under “Final (DRAFT)” [↑](#footnote-ref-1)