Reflection – Hobby Detectives

Jackson Rakena (300609159)

Clarifications & major design decisions

- We assumed that for a player to leave an estate, there must be a door in the direction they wish to go. For example, if there are doors on the top or the left of an estate, then they are only able to move left or to the top, and they will appear next to the door in the direction they chose to exit by.
- We decided to conceptualize the game as a tile-based map, consisting of 24 tiles in each direction. The board is represented by a 2D array of Tiles, and specialist tiles are subclasses of Tile (for example: Estate, UnreachableArea)
- We decided to represent the estates as the first letter of their name (for example, C for Calamity Castle, etc), so it was easy to identify for the player. Doors of estates were also represented by the symbol "*", so it is easily contrasted to the estate tiles. In this fashion, players were represented by the first letter of their character's name.

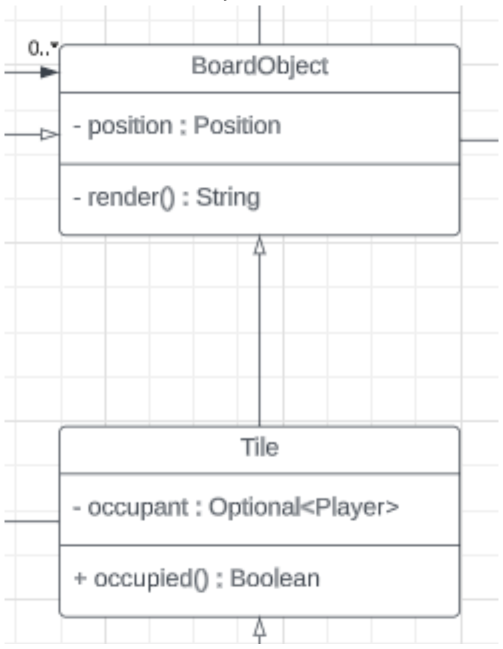
CRC cards

- We used CRC cards to keep our design adequate by physically drawing them out on cards and moving them around a table. This allowed us to group similar cards that collaborated physically close, so we could perceive any issues or holes in our design.
- We also used CRC cards to keep our design adequate by writing the responsibilities for each class, which made it easier to map out the class diagram after we finished writing the cards.

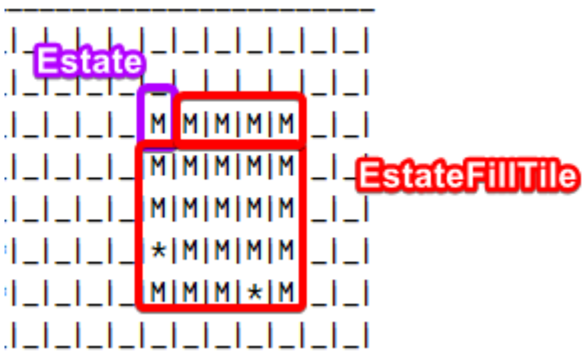
Individual contributions

- (Design) I came up with the idea of using Java enums to represent cards, weapons, and estates (for example, Estate.PERIL_PALACE). This ensures that our variables storing those data types can never be invalid.
- (Code) I came up with the technique where we ask the user to change the tablet to the next player and then wait for an ENTER input, so that the current player does not see sensitive information that should only be seen by the next player.
- (Design) I came up with the CardTriplet class, which stores one card of each type (weapon, estate, character), so that methods that deal with all of the cards can pass them around as one object.
- (Design) I devised the abstract BoardObject superclass, which all board objects (Tile, Estate) inherit from. The BoardObject class contains a variable of type Position (a 2D vector), as every object on the board must have a (x, y) position. It also has an abstract "render" method returning a String, as all board objects must return one character that represents what must be drawn to the board to represent them. It also means that the board can be represented as a BoardObject[][] two-

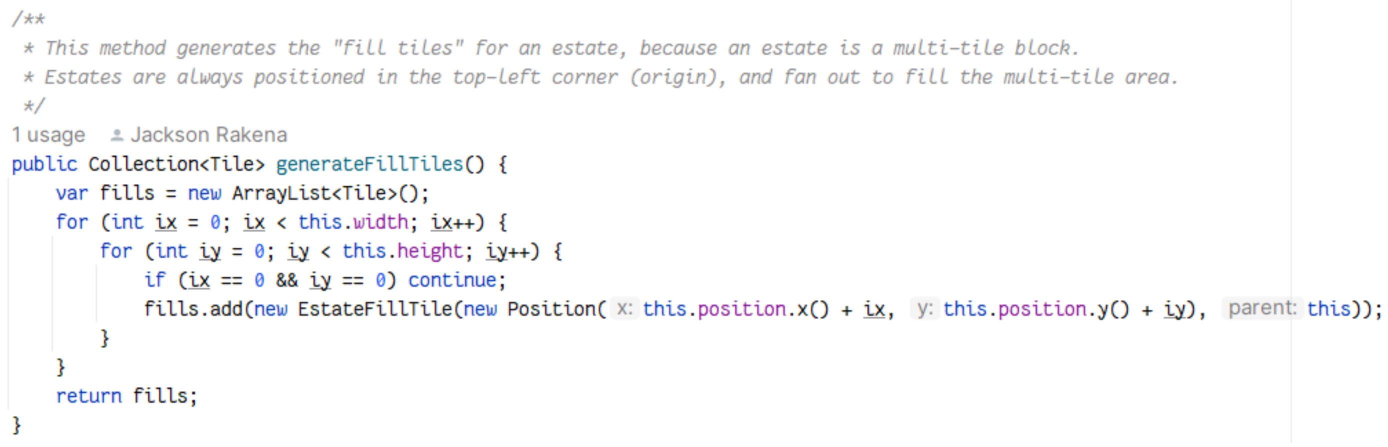
dimensional array, and makes rendering easier.



- (Code) I wrote the main implementation for our multi-tile rendering system, which is used for the estate (which extends over multiple tiles, but logically is one tile), and the unreachable (grey) areas. This is done by using the `Collection<Tile> generateFillTiles()` method, which generates a list of Tiles that are of type EstateFillTile. The fill tile overrides render() to render the estate letter (like C for Calamity Castle), and also stores a pointer to the Estate class.



When the player attempts to move into an EstateFillTile, the game engine checks to see if that tile's estate has a door at that position, and if it does, the player is consumed into the Estate. Fill tile positions are stored relative to the origin of the estate, so the tile to the right of the top-left corner will have position (1,0).



```
/**
 * This method generates the "fill tiles" for an estate, because an estate is a multi-tile block.
 * Estates are always positioned in the top-left corner (origin), and fan out to fill the multi-tile area.
 */
1 usage   Jackson Rakena
public Collection<Tile> generateFillTiles() {
    var fills = new ArrayList<Tile>();
    for (int ix = 0; ix < this.width; ix++) {
        for (int iy = 0; iy < this.height; iy++) {
            if (ix == 0 && iy == 0) continue;
            fills.add(new EstateFillTile(new Position( x: this.position.x() + ix,  y: this.position.y() + iy),  parent: this));
        }
    }
    return fills;
}
```

Challenges

- A significant design challenge is that some of our group members would actively discourage design ideas that they perceived as being too complex for their Java understanding. For example, the Position class that stores two integers in one object and has utility methods for performing 2D

vector arithmetic was opposed by some group members, in preference for having two `x` and `y` variables instead.
- Another challenge was that a large amount of logic was being handled in the Board class. At the worst point, Board had 20 methods (+ constructor). We addressed this by separating game logic (logic not directly related to drawing on the board) to a new Game class, which stored game-related data and logic. We have updated our design documentation to reflect these learnings.

Possible improvements

- The input system in general could be redesigned. At the moment, it asks the user for each move in one string (i.e. LLUURR for left->left->up->up->right->right), but it could take the users input move-by-move, which could be easier to use for the user, as they would not have to think about the entire move at once.
- The input validation system could be improved, potentially tracking all the errors with the user input (in a list of errors, perhaps), instead of rejecting the input after the first one.
- CRC cards could have been used more in-depth, especially in the later phases of design, to ensure that each class's responsibilities were well defined. During design, we often shifted responsibilities around, because we did not have a concrete plan for which object did what (for example, sharing responsibilities between Board and Game)