Luke Lovett (lalovett)
Ethan Akers (akerse)
Christian Gonzalez (cjgonzal)

Overall Design

MVC, Model

The model is implemented as a server side shared model between all clients that attach to it. We included in the model everything that should be updated across all players that connect to the game. We also allow the model to keep time and update itself as time passes.

MVC, View & Controller

Each client has their own view and controller. The controller comes in two forms, the first form is the game controller which is used to interact and play the game. The second form is the tool for setting up the games parameters (the controller user, which will be discussed at length later).

Where and how design patterns were used

Simple Factory

We made use of the simple factory pattern on top of our Player class to create a simple factory for instantiating new players.

Singleton Pattern

We made use of the singleton pattern to limit the instantiation of the model such that only one can be created. We did this to prevent the issue of multiple models existing and having to sort out which one should be connected to. The singleton pattern allows us to reference the model throughout the program without directly passing it through as a variable.

Strategy Pattern

The strategy pattern was used to implement the rules that are used when growing the resources such as the upper limit of how many resources can exist on a given tile, and whether or not a resource should be added to the tile at each time step.

Observer Pattern

The observer pattern is not used explicitly with the Observer interface and Observable class. Instead, the pattern is used in that our Model serves as an observer that is notified whenever there is a key input from a connected client. That input is then used to update the appropriate player's position which can then be displayed in the view.

Features

Player interaction with resources and other players: This is controlled entirely on the server side. When a key is pressed, the client expresses an intention to move one unit in the corresponding direction to the server. The server checks the position of all other players, and if none already occupy the square the player has requested to move to, the server allows the move. Whether the player is allowed to make the move or not, the server sends back the updated list of all current player locations, including the player that requested the move initially. The player then asks if any resources were collected by moving to this location. The server replies with the player's current resource count (which is now higher if they collected a new resource by this movement). This means that a player cannot collect a resource by simply sitting in a location. The server only checks to see if a resource is consumed when a player makes a request to move. Restricting these aspects of the game to the server makes it impossible for a player to create a hacked client that can determine and modify its own resource count or where/how fast it can move (which is highly unlikely to happen for this game, but we like to plan for the future). Lastly, the player's resource count is displayed in the client in the top left-hand corner.

Controller User

In our application, the controller user takes the form of a simple graphical user interface with two sliders, two checkboxes, and one button. The interface pops up when Model.class is run, and the game does not start until the interface's "Start" button is clicked. Players can connect before this time, but none of their input is handled. This user interface controls three aspects of the game:

- **Starting assignment of resources:** The first slider controls what percentages of the resources spawn in which cluster. When the submit button is clicked, the value of the slider is passed to the createStartingGrid() method. First, two random positions on the grid are selected to be the epicenters of the clusters (one is guaranteed to be in the top half of the screen and the other is guaranteed to be in the bottom). Then, the euclidean distance of each cell in the grid from the epicenters is calculated. If a cell is "close enough" to one of the epicenters (this is calculated based on what percent should appear in each cluster), it is assigned a random resource value between one and six. This results in two roughly circular clusters of sizes proportional to the desired percentage of total resources.
- **Game timer:** The duration of the game is determined by the second slider. Players can choose a time (in minutes) between 0 and 20. Since our game grid updates once each second, in our update method we decrement the number of seconds remaining by one each time. When the number reaches zero, the timer that drives the game stops and the grid is no longer updated. On the user interface of each player, the number of seconds remaining is displayed in the top left-hand corner. We implemented this without making any kind of modification to the the ModelInterface.

- **Player and resource visibility:** Player visibility and resource visibility can both be toggled via their own respective checkboxes in the controller user interface. They are checked by default, meaning that players can see both other players and other resources. If resources are set to be invisible, players are simply sent an empty grid when they request the game grid. Since, as was mentioned above, the server determines whether or not players consume a resource, and the server holds the actual grid, this does not affect resource collection. Player collision works almost exactly the same. When a client requests the list of players, it is sent an array that is blank in every position except in the position of its own ID. Again, since player collision is controlled on the server side, the change in visibility has no impact on the function of the game.

Changes to the UML

- All of the GUI components, which are instance variables in Model.java
    - `private JFrame frame;`
    - `private JSlider res;`
    - `private JSlider length;`
    - `private JCheckBox playerV;`
    - `private JCheckBox resV;`
    - `private JLabel fracLabel;`
- The PlayerFactory class and its one method, createPlayer()

Our assignment 3 submission was built in such a way that implementing the changes necessary for assignment 4 required adding relatively little. With the exception of the GUI, it only extended functionality that was already there.