Group 0653
Walkthrough Of Game Center

When GameCentre opens, it prompts you for a username and a password to login. A new user should click "**Sign-Up**" to be prompted with a signup screen. A valid username and password must both not be empty, and passwords must be longer than 5 characters. Invalid characters are not allowed in these fields (else we would have issues adding the Username/Password to the MySQL database). After a user signs up, their username and password combination is added to a user database. When a user logs in, we check to see if the username/password combination exist in the database and are valid.

Logging in brings the user to the **GameCentreActivity**, where the user's can select one of three games: "Sliding Tiles", "Minesweeper" or "2048". From this screen, the user can see their personal highscores they've set for each of the games. Clicking any of the three games brings the user to the **GameMenuActivity**.

From the **GameMenuActivity**, the user can choose to start a new game, load a previously saved game, or view the scoreboard for this specific game. The user can click an info button to display a brief list of instructions for the given game. If the user starts a new game, they are brought to the **NewGameActivity**. From here, the user should specify a valid file name for the file of the game they are about to play. The file name must consist of alphanumeric characters, and not be empty. You cannot name a file as the same name as a previously named file. If the game allows, the user should specify the number of undos that they want to be able to use in the game. After that, the user should specify the size of the board they want to play on, if the game has variant boards. The user can now start the game.

Minesweeper

The user's goal is to clear a minefield by marking mines and clicking the tiles that have no mines. The user's first click cannot result in revealing a mine. Whenever a user clicks a tile, either a bomb or a number marking the number of adjacent bombs is revealed. Revealing a bomb results in a loss. The user can long-press a tile rather than tap it in order to mark the tile with a flag, denoting that a bomb is under that tile. Using the process of elimination, the user can deduce where the bombs are given each tile's number of adjacent bombs. The user is able to win the game by flagging all bomb tiles and revealing all non-bomb tiles.

Sliding Tiles

The user's goal is to order the board in row-major order. User's can click on a tile adjacent to the blank tile to swap the blank tile with the adjacent tile. Every board can be solved.

2048

The user's goal in this game is to amass the highest score they possibly can. The game cannot be won, but when the user has no more moves left to make, the game ends and the score is added to the scoreboard. User's interact with the board by swipe controls to move the tiles on the board. When two tiles meet that have the same value, the tiles merge, doubling the value of the tile after merging. After each move, a low value (2 or 4) tile is added to the board.

During any game, the user can choose to **save** the game using the menu at the bottom of the screen. The user can **quit** using the quit button, and if the game permits it, the user can **undo** to a previous game state using the undo button. Any time a game ends, the save file for that game is deleted. The autosave function autosaves our game every few moves for all games.

When a user loads a game from the **NewGameActivity**, they are prompted with a selection of games which had been previously saved. From this dialog box, they can delete a save file or load into it.

When a user opens the **scoreboard**, they can view the scoreboard values for the game they are currently on. The different variants for the current game appear as individual scoreboards. The user can swipe or use the UI to view the scores for a certain variant.

What is your unit test coverage?

The following classes had high line coverage:

- 100% - CoordinatePair.java
- 100% - GameController.java
- 100% - GameFactory.java
- 100% - GameState.java
- 100% - GameCardItem.java
- 99% - MinesweeperBoard.java
- 100% - MinesweeperController.java
- 100% - MinesweeperFactory.java
- 100% - MinesweeperTile.java
- 98% - SlidingTilesBoard.java
- 100% - SlidingTilesController.java
- 100% - SlidingTilesFactory.java
- 100% - SlidingTilesTile.java
- 95% - TwentyFortyEightBoard.java
- 100% - TwentyFortyEightController.java
- 100% - TwentyFortyEightFactory.java
- 100% - TwentyFortyEightTile.java

The rest of the classes had low line coverage. Many of the classes that have no coverage were not possible to test, as they were either reliant on the application's context which could not be given in a JUnit test, or they were View classes, which were not appropriate to test.

What are the most important classes in the Project?
We decided to organize our classes into different packages to make our project's files more manageable.

ScoreboardDBHandler
This class handles the interaction between the GameCenter and the MySQL database. Whenever a game completes, we use this class in order to add a ScoreboardEntry to the database. Whenever we consult the ScoreboardActivity or the GameCentreActivity, we also ask the database for the user's highscores or the game specific highscores.

ScoreboardActivity
This class displays the scores for all games, and holds a ScoreboardListFragment. Without this class, we are not able to compare our scores for different players.

Each of our games is a grid based tile-esque game. Minesweeper, 2048, and Sliding Tiles all have classes extending Board, Controller, Factory, Fragment, and Tile. For each of the games, the respective game's **Board** class is extremely important, as that is the class that holds the values for the tiles of the game's board. For instance, in Minesweeper, MinesweeperBoard.java which extends **GameState** is the current state of the the Minesweeper game. It is the class that holds all the logic behind Minesweeper, and determines what happens when a user interacts with the Minesweeper's board. In addition, in 2048, the TwentyFortyEightBoard is what determines what happens after a user swipes, including the merging and the spawning of new tiles. Moreso, when we save and load our games via serialization, it is the respective game's **Board** class which we serialize and load.

Each individual game has a **Controller** class extending from **BoardController**. The individual controller's are used to translate the user's input to actions that the **Board** can interpret.

The **GameFactory** is a factory for all the objects we must initialize for a game. It proved useful when it came to writing JUnit tests, and for building the objects we needed to start a launch a game. The GameFactory was also responsible for interpreting the settings the user gave to it, to create a proper GameState of that variant.

What design patterns did you use? What problems do each of them solve?

We implemented the following design patterns in our project: Observable, Model View Controller, Iterator, and Factory.

Observable

Implementation

- GameState abstract class implements observable. This causes the board's for all 3 of our games to be observable.
- GameFragment implements Observer. They observe GameStates for updates. This is an essential part of MVC design pattern that we use (detailed below).

Problems Solved

- This lets us detect whenever GameState updates without constantly polling the GameState.

Model View Controller

Implementation

- For each of the 3 games, we used MVC. The Fragment and different views displayed the game and recieved inputs. When the user interacts with the game, GestureDetectGridView sends data to a GameController subclass appropriate for the game.
- The GameController processes inputs and updates a subclass of GameState.
- Whenever a GameState is updated, then through the observable pattern, the fragment and its views are updated.

Problems Solved

- This allows us to separate the models and views, and makes sure most of the game logic lies in the controller.
- As the model is separated, only GameState is required to setup a game (ie. from a save file). The view depends on the model, and the controller doesn't change for different save states.

Iterator

Implementation

- In SlidingTilesBoard.java. Used to iterate through the tiles of the Sliding Tiles game.
- In TwentyFortyEightBoard.java. Used to iterate through the tiles of the 2048 game.

Problems Solved

- Allows us to iterate over the elements of a two dimensional board, without worrying about multiple dimensions and having to use a messier nested for loop.
- Allows us to not need to worry about how the tiles are stored when we iterate over them.

Factory

Implementation

- Each of our three games have a respective factory.
- Allows us to create different boards for different games of a multitude of sizes.
- The factory holds a list of settings for each of our game. Clicking new game creates a new game with the specified settings
- Works with our UI to show different settings in NewGameActivity.java

Problems Solved

- Obscures the creation process for any of the game elements required for each of our three games.
- Allows for difficulty settings, ability to change undo amount, ability change board size before creating a game with the given specifications.

How did you design the scoreboard? Where are the high scores stored? How are they displayed?

The scoreboard was designed with Android Studio's implementation of MySQL in mind. We wanted to use MySQL as we decided it solved many of the issues we faced with data storage. We needed to find a way to make sure our data would not be lost when the app was closed or the process was interrupted. SharedPrefs was not a good option as it would result in a lot of cumbersome code, and would be annoying to work with when we are dealing with the large amount of data we planned to work with. We also wanted to have a convenient way to- retrieve and parse the data, and SQL queries seemed to be a convenient way to get the specific data we were looking for. For example, we can use specific SQL queries to get the max score for a game with multiple variants (i.e. Sliding Tiles 3x3, Sliding Tiles 4x4, Sliding Tiles 5x5), while also getting the max scores for all other games. This is useful for finding user-specific highscores.

The highscores are stored in a Scoreboard table within a Scoreboard database. Each entry in the table consists of the name of the game, the score, and the username of the user who set the score. Anytime a game is won, the score is added to the scoreboard, akin to old arcade games.

From the game center selection activity, we can choose any of our three games, bringing us to the game's main menu. From there we are able to select the scoreboard for the given game, which displays the scores for only that game, but including the different invariants for each game (variants are the different game sizes). The scoreboard is displayed in ScoreboardActivity.java which has a ScoreboardListFragment for each variant for the game. The ScoreboardListFragment has a ListView which lists the scores for each person across each variant. The ScoreboardListFragment also holds the all time high score for the variant.