Group 0602

1. What is your unit test coverage?
The unit test coverage for our package according to android studio is: 24% of all classes and 34% of methods.

We have a few things we would like to clear up about our test coverage.

Part of the reason our test coverage is apparently low is because we have several classes that are view only and contain no logic such as:

1. The classes for gesture detect grid:
    - GestureDetectGridView
    - GestureDetectGridViewLongPress
    - GestureDetectGridViewLongPress

2. The Minesweeper Activities that initial stages of the game:
    - Sweeper Choose Dimension Activity: easy, hard, etc.
    - Sweeper Starting activity: loading, new game, etc.
    - MinesweeperActivity: displays GUI for Minesweeper

3. The classes for movement controllers, which take into account the actions to be taken according to short pressing or long pressing
    - MovementModel
    - MovementModelComplexPress
    - MovementModelSimplePress

4. The classes for score:
    - GameScoreboardActivity
    - MenuScoreboardsActivity
    - ScoreboardArrayAdapter
    - UserScoreboardActivity

5. Simon activities that display the initial stages of the game:
    - SimonStartingActivity: loading, new game, etc.
    - SimonGameActivity: displaying the GUI for simon

6. Sliding tiles:
    - SlidingGameActivity: displaying the GUI for simon
    - SlidingTilesStartingActivity: loading, new game, etc.

General Classes/Activities:

- ChooseDimensionActivity: which lets the user choose dimension/undo for Simon and Sliding tiles.
- CustomAdapter
- GameLauncherActivity: which displays the games

We would also like to point out we have certain activities that we refactored to the best of our ability, but we still wouldn't able to test without taking an approach out of the scope of the course as they dealt with context:

- SaveAndLoadBoardManager which deals with the saving and loading of games.
- MovementModelClasses for all 3 games: which we attempted to test a couple of methods that didn't need context but were not able to test the rest.

However, for our model classes, we provided full test coverage:

- General classes: Tile, UndoStack, MoveTracker, BoardManager, Board,
- Scores: TextFileManager
- Simon: SimonTile, SimonTileBoard, GameQueue, SimonBoardManager
- Minesweeper: SweeperTile, SweeperBoard, SweeperBoardManager, BombTypes
- SlidingTiles: GameInputValidator, SlidingTileBoardManager, SlidingTilesBoard

2. What are the most important classes in your program?

Essentially, all of our games operate in a board based structure. Therefore, the most important classes are the parent model classes: Board, BoardManager and Tile. Each game extends every single one of these classes, adding any extra features necessary.

Additionally, our movement controller model also played a really important role, as it allowed us to implement simple presses to the screen (short) and complex presses to the screen (long). Movement Controllers provide the structure of the game: by making sure taps are valid, determining when a game is over and overall implemented the rules game play should follow. As such, without movement controllers, there would be no game to play.

Finally, we also believe the gridview classes are particularly important for our program because they allow us to display our boards in grids and manipulate the movement controllers.

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

- Model-View-Controller
- Iterable
- Observer
- Adapter

We made extensive use of the **MVC** pattern. Our Activities are our controllers, and they receive data from the models, and send it do the views. Our views are all stored in XML files, and our Model classes are: Tile, Board, BoardManager, MovementModel. Our Activities receive data from input on the views, send them to the models, who store them, manipulate them, and then send a result back, which the views then display. We linked the Controllers(our Activities) and the Models through the **Observer** pattern. When some model properties get changed(Board, MovementQueue, or MovementModel), they update the activities that are their observers, and the activities then update the views. The MVC pattern helped us separate the logic from the UI, which made the code a lot more testable, and a lot easier to extend and change, as the models do not have any knowledge of the views or controllers that use them. This made development faster, and the code cleaner and more structured. The Observer pattern made it very easy to update Activities on important property changes, in an efficient manner. For example, in our game Simon, in our main Activity(Controller), the controller observes 2 different models: the gameQueue which stores the order the tiles have to be pressed in, and the MovementModel. Using the Observable pattern and the update() method, we can easily distinguish by which observable object triggerred the Observer, and update accordingly. This gets rid of the necessity to constantly check the state of our observables, and instead we just get notified whenever there is a change. Another pattern we use is the **Iterator** pattern, which we use in SlidingTiles(as per phase 1 requirement) in order to devise the appropriate way to iterate through our board, making sense with the logic(when iterating, will move on a row column by column, and once row is finished, will move on to the next row, until no more rows). We also use the iterable pattern for the GameQueue, which uses a listIterator. Using the listIterator, we store an instance of the iterator of GameQueue in the MovementModel for Simon. The ListIterator then keeps track of which tile in the GameQueue the user has to press currently, and can as a result easily check whether the user presses the right or wrong tile. The Controller for the game(the activity) uses another iterator to iterate through the updated game queue and display it in the right order. The Iterable pattern helped create ways to iterate through difficult data structures, and keep track easily of where we are through an iterable.
We also used the **Adapter** pattern, where we created a custom adapter and a ScoreBoardArrayAdapter, that connected 2 incompatible interfaces: a GridView and a List of Data for the CustomAdapter and a ListView and a List of Data for the ScoreBoards. This allowed us to display how we want each row to look in the GridView, given the list of data. The adapter makes sure that no matter how much input there will be, since we are using a gridView and a ListView, the UI will adapt(by allowing user to scroll through elements) to any size of rows, which means we don't have to hard-code any data.

4. How did you design your scoreboard? Where are high scores stored? How do they get displayed?

Where are they stored?

For our scoreboard, we decided to use text files. Each game has a text file where it stores *only the high score* for every user that has played the game. Each score entry is stored as a single line in the file with the format [User,Score].

Additionally, each user has a text file with their name, where it stores *only the high score* for all 3 games. In this text file, each score entry is stored as a single line with the format [Game,Score].

When are they stored?

All 3 games use the same activity **ScoreScreenActivity** where it displays the score obtained in the game and calls the class **TextFileManager** to store the activity. ScoreScreenActivity makes sure we store the new score in the game file and in the user file.

What does the class TextFileManager do?

TextFileManager is a static class with the ability to save and load scores into the text files previously mentioned. It has methods in place to replace old high scores and attach the new ones if necessary. It also has the ability to return maps containing the user/game as the key and the score as the value assigned to that key.

How are high scores displayed?

We have implemented two scoreboards screens. One scoreboard is specific to the user, in which we use TextFileManager to read the contents of the user file and the activity displays them. The other scoreboard is a general game scoreboard (which can and is used for all 3 games) where TextFileManager reads the content of the game file and the activity displays them.

How is the game aware of the current user playing when we want to display our scores/ save new scores?

We implemented a static class SharedPreferenceManager which can write and read from a specific shared preference. In the game launcher, we create a shared preference file where the current user player is stored under the key “thisUser”. This way we don’t have to pass on the name of the user as an intent from activity to activity, we can simply access the shared preference when we need the name of the user. The SharedPreferenceManager is also used for storing the users and passwords in the login/register activities!