Unit testing coverage

As you can see, we wrote comprehensive unit testing for many of our classes. The classes at or near 100% are the classes which have next to no dependence on the graphical user interface but rather on the backend logic of the games. The GUI dependent functionalities were tested by running the app, as opposed to writing the unit tests.

Classes that were unit-tested:

1. Board
2. MasterMindCombination
3. MasterMindManager
4. MatchingBoardManager
5. Score
6. SlidingTileBoardManager
7. Tile

Important Classes

MasterMindCombination

This is an important class used by MasterMindManager to manage the individual guesses and the answers. Contains a lot of the logic in making the MasterMindManager work properly (for example, this class allows MasterMindManager to generate the answer for the game and create an instance for a guess). It does this by implementing a comparison-like operator to check how correct a guess is, so that all the MasterMindManager needs to do is contain the general game logic. Furthermore, this class is a result of careful team planning and proper documentation.

MasterMindGameActivity

Contains all GUI logic for MasterMindGame. Important as it contains a large portion of the game logic visually represented to the user. Furthermore, a lot of design choices, refactoring, and overall care were put into designing the code to make it as efficient as possible to display the possible results. This graphical interface is independent and designed completely from scratch.

Score

This class contains the final information of a singular play-through of a game. This comprises the score, username, and game size. Furthermore, it also implements comparable which allows the Scoreboard to easily sort the Score classes.

GameSaveStates

This is a Serializable class, containing all the saved games. It implements the Singleton design pattern by making the constructor private as only one instance of this state is saved and run throughout all of the classes. AccountInfo This is a Serializable class, containing all the user account information. It implements the Singleton design pattern by making the constructor private as only one instance of this state is saved and used.

Design Patterns

We implemented several design patterns, including Singleton, Private Class Data, Dependency Injection, Strategy Pattern, and Model-View-Controller. Singleton prevented multiple instances of classes such as GameSaveStates, AccountInfo, and Scoreboard, as all games make use of the same instance. Private Class Data prevents modification of classes such as Score and AccountInfo after initialization which prevents any user of the code from tampering with variables that should not be changed. Dependency injection is used for example in the implementation of ScoreBoard in that Scores are passed to the ScoreBoard rather than information about the scores. This allows us to be more free in modifying the implementation and faculties of the score, without messing up the ScoreBoard. For the strategy pattern, we implemented scores to become comparables so that Collections.sort have the desired result when applied to a list of scores whereas applying to an array of primitive data types would have an entirely different result. Had we decided to implement a game where the lowest score was not the best score, we would easily have been able to implement this by extending the Score class and overwriting the compareTo method. The project also uses the model-view-controller pattern on a more general level in the GUI methods for the MasterMindGameActivity where the model is MasterMindCombination and MasterMindManager, the view is the xml diagram that is drawn up for the game. The controllers are the buttons and the Activity.java files that control the flow of the game.

Scoreboard implementation
We designed the scoreboard to display the lowest scores on top (since all 3 games are optimisation games of some sort where the objective is to complete the game in the least number of moves). It is a class that is serializable so we can easily store all of its information. Furthermore, we designed a score class that is used by the scoreboard so that it can contain multiple facets of information of the game's score (such as the username associated with a score, the score itself, attributes related to the specific game played). More importantly, the scores class method extends Comparable, which makes it very easy to sort for the Scoreboard, which contains a List of all possible scores. This makes displaying the scores easy as we can simply call the Collections.sort method on the List containing all the possible scores, thus making it easy to display the information on a textview GUI component on the activities. A more general implementation of our scores is shown through the 3 separate ser files we have which each represent the object of a score.