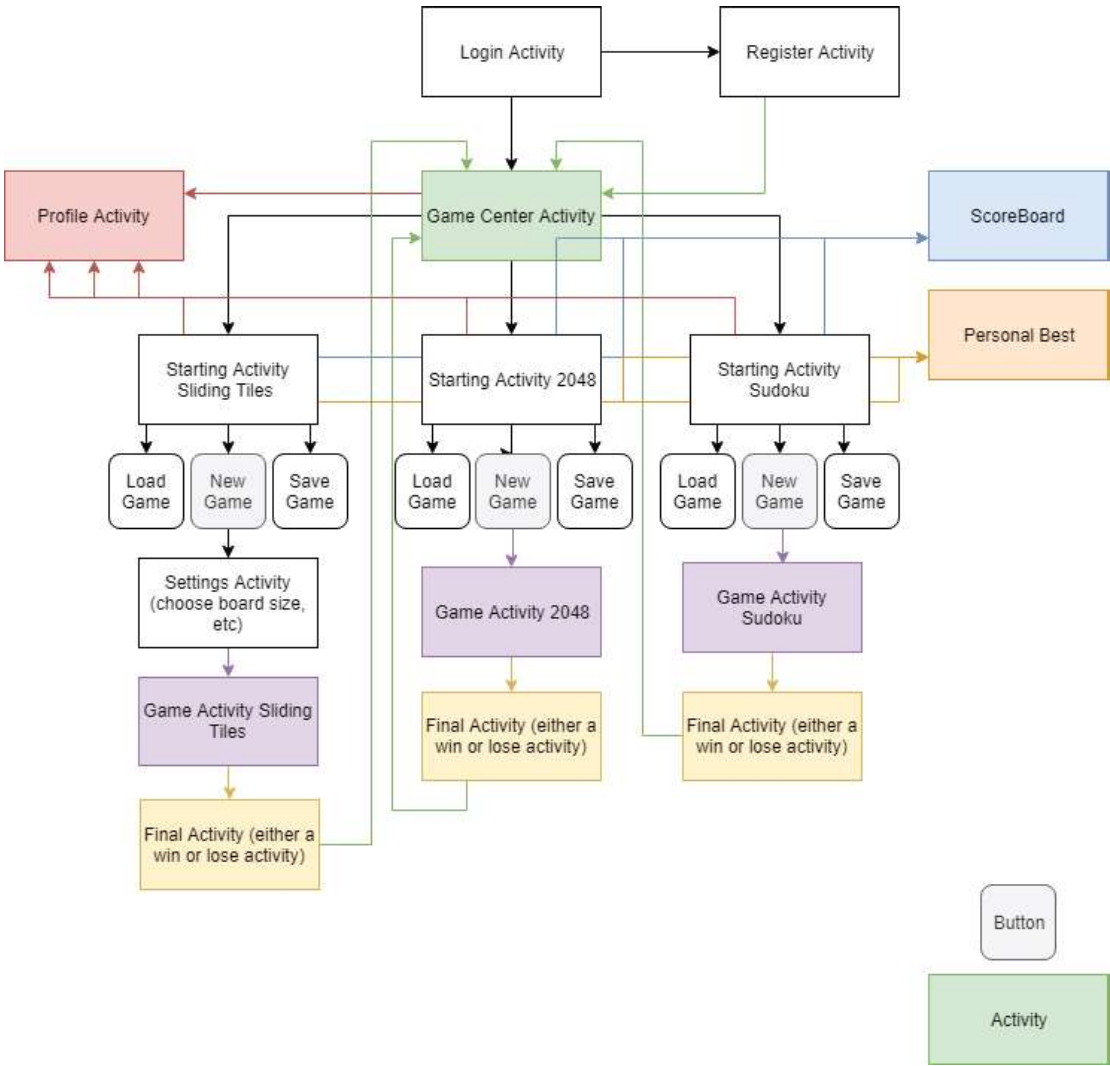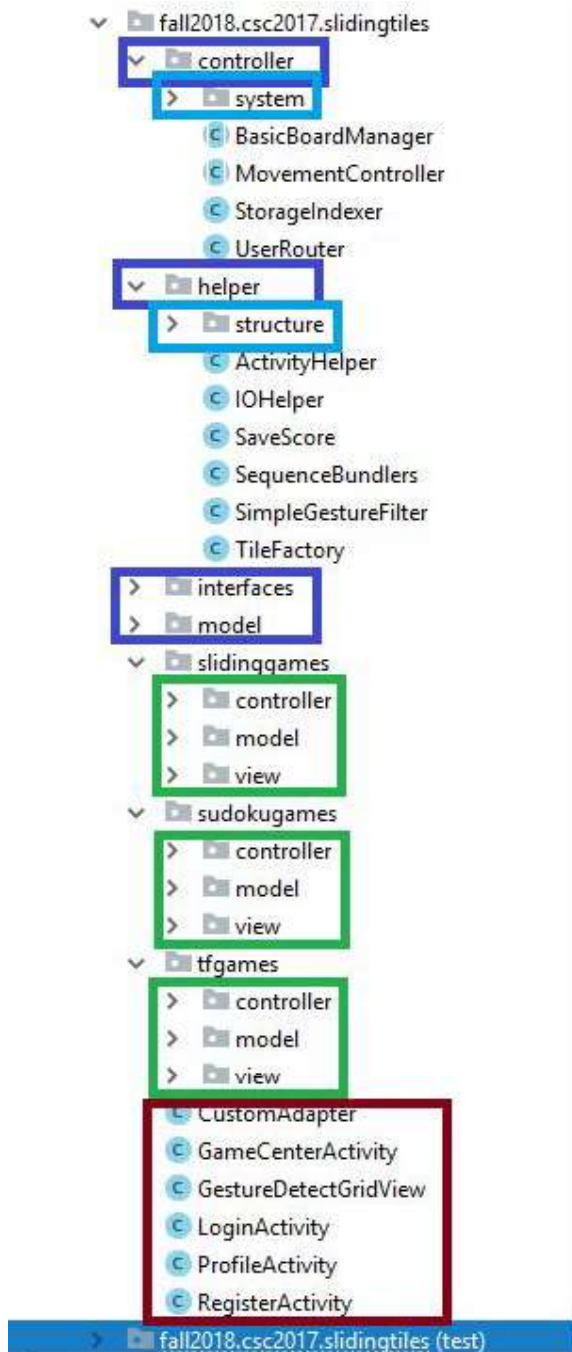Below is a diagram of how our "Activity" classes interact with each other from the user's perspective (i.e. each activity is what the user will visualize while using our app):
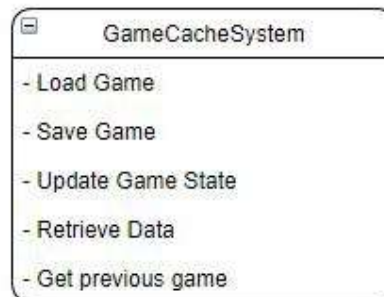
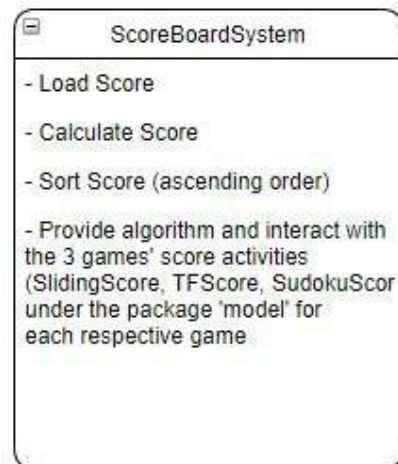# Important Classes and their Implementation:

Firstly, as per the screenshot of our package organization on the right, we have tried to organize each individual class into its corresponding location in as much detail as possible to ease the marking process but also for intuitive purposes. For example, under the package "helper", each 'helper class' is often re-initialized several times in other classes to perform the same, repeated operations, which help avoid duplicate code and large classes/methods.

Now, we will use some of our CRC cards to demonstrate a few selected examples that are crucial to our program:

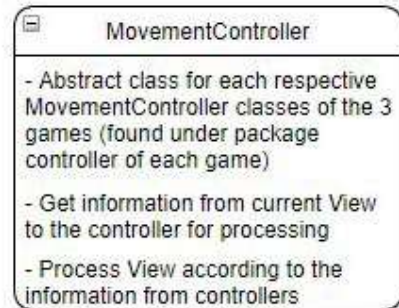1. Game Cache System (under package 'system')

GameCacheSystem
- Load Game
- Save Game
- Update Game State
- Retrieve Data
- Get previous game

2. Score Board System (under package 'system')

ScoreBoardSystem
- Load Score
- Calculate Score
- Sort Score (ascending order)
- Provide algorithm and interact with the 3 games' score activities (SlidingScore, TFScore, SudokuScor under the package 'model' for each respective game

The package tree shown on the left:

- fall2018.csc2017.slidingtiles
  - controller
    - system
      - BasicBoardManager
      - MovementController
      - StorageIndexer
      - UserRouter
  - helper
    - structure
      - ActivityHelper
      - IOHelper
      - SaveScore
      - SequenceBundlers
      - SimpleGestureFilter
      - TileFactory
  - interfaces
  - model
  - slidinggames
    - controller
    - model
    - view
  - sudokugames
    - controller
    - model
    - view
  - tfgames
    - controller
    - model
    - view
  - CustomAdapter
  - GameCenterActivity
  - GestureDetectGridView
  - LoginActivity
  - ProfileActivity
  - RegisterActivity
- fall2018.csc2017.slidingtiles (test)

3. UserPanel (under package 'system') [Bonus for extra feature in Phase 1]

```
┌─────────────────────────────────┐
│ ⊟         UserPanel             │
├─────────────────────────────────┤
│ - Get information of the current│
│ player                          │
│                                 │
│ - Update and process new informatic│
│ of current player               │
│                                 │
│ - Used in Profile Activity      │
└─────────────────────────────────┘
```

4. Movement Controller (under package 'controller')

```
┌─────────────────────────────────┐
│ ⊟      MovementController        │
├─────────────────────────────────┤
│ - Abstract class for each respective│
│ MovementController classes of the 3│
│ games (found under package      │
│ controller of each game)        │
│                                 │
│ - Get information from current View│
│ to the controller for processing│
│                                 │
│ - Process View according to the │
│ information from controllers    │
└─────────────────────────────────┘
```

5. BasicBoardManager (under package 'controller')

```
┌─────────────────────────────────┐
│ ⊟      BasicBoardManager         │
├─────────────────────────────────┤
│ - Abstract class for each game's│
│ Board Manager                   │
│                                 │
│ - Control Tiles for each game   │
│                                 │
│ - Process changes in score and  │
│ provide feedback                │
│                                 │
│ - Determine and control the starting│
│ and ending states for each game │
└─────────────────────────────────┘
```
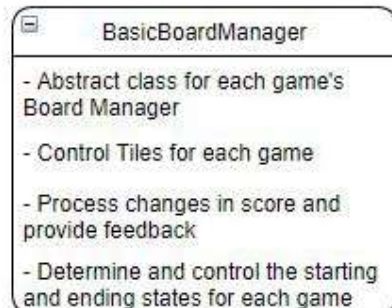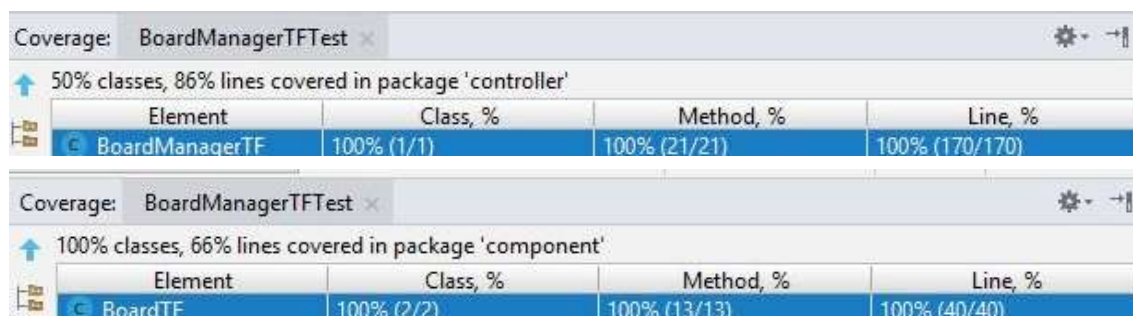
Unit Test Coverage: An Example

We have written a complete unit test class (under 'test' package) with 100% coverage for
- 2048 Board Manager and 2048 Board (see below)

We have also completed test files for
- Sudoku Board and Board Manager
- Sliding Tiles Board and Board Manager

Throughout the project, we have attempted to proceed with Test Driven Development (TDD) when designing the classes in the first place-- that is, we first wrote the tests for functionality testing then proceeded with writing the method itself. Unit tests are only written for controller and model classes (view/activity classes at the UI level).

Coverage:	BoardManagerTFTest ×			
50% classes, 86% lines covered in package 'controller'				
Element	Class, %	Method, %	Line, %
BoardManagerTF	100% (1/1)	100% (21/21)	100% (170/170)

Coverage:	BoardManagerTFTest ×			
100% classes, 66% lines covered in package 'component'				
Element	Class, %	Method, %	Line, %
BoardTF	100% (2/2)	100% (13/13)	100% (40/40)

Design Patterns and Why we chose them:

Factory Design Pattern:
- since all of our games are tile-based, the factory design pattern enables us to simply call methods such as <createTile()> and pass in the type wanted and corresponding values.

MVC:
- Following Model–view–controller pattern, we have sorted each class to these specified packages for each of the three games.
- For example,
    - Model (tiles, board and scores);
    - View (activities and action listeners);
    - Controller (boardManagers and Movement Controllers).

Singleton:
- An example would be a GameCacheSystem, where we create a global system for one user at a given time.

- This avoids shotgun surgery, since we just have to change or update the code in the singleton class rather than all 3 games as all 3 games have the same behavioral properties.

Abstract:
- Some implementations of abstract classes can be seen in controller modules and models, such as movementcontroller, basicboard, basicmanager and basictiles.
- This is especially convenient for refactoring since one of its advantage is that it allows us to use a more general class type in functions. It also makes some of our functions more flexible and is able to be applied to all classes of certain type.
- Help us understand what to implement and get an outline of what we're going to do to implement a class with a relatively complicated logic.

Observer:
- The typical example of our observer is the board class, which effectively interacts with the activities to display the change of data to views.
- Observer pattern makes it easier to display information of our backend code

Iterator:
- The typical example is that of the board classes, it helps us get the tiles without explicitly exposing too much of our variables, strengthen the encapsulation of our classes.
- With a iterable board, it's easier for us to use the board since we only need to use a 'for loop' to get our expected results without having to know how the board is implemented.

Scoreboard: Implementation and Intuition Behind

Front-End:
- For the visual design of the Sliding Tiles scoreboard system, we have chosen to implement a local version of the scoreboard for the current user which could be accessed by clicking the button "Personal Best", and also a global scoreboard, which takes into account the ranking for all registered users. Both buttons could be accessed in the starting activity.
- For simplicity and clarity, we have chosen to simply include the top 3 players for both local and global scoreboard of Sliding Tiles game including all 3 different board-sizes. For the other two games, we have only included the top 5 global players in the scoreboard.
- The score for each game is calculated as follows:
    - Sliding Tiles: least number of moves (touches) to solve the board
    - 2048: least number of moves to get to the 2048 tile
        - Example for 2048:

- Each game's scoreboard is also inherited by the super class "BasicScoreBoardActivity", which provides the primary layout for displaying the top high scores in the format "username", "score" (sorted in least number of moves taken)
- The score data (number of moves) will first be stored whenever a game is completed (as per the figure above) by each game's corresponding activity and layout class (i.e. "activity_final_score_tf.xml" under layout package and "FinalScoreTFActivity" under 2048's view package) when a game is completed (either a win or lose).
  - We have first created a "BasicFinalScoreActivity" that is used as a father template to be extended from each game's score activity.
  - In this super class, we will be able to get the current board manager for each individual game by accessing the current user through class "UserPanel".
- In each game's respective (global) Scoreboard Activity:
  - The high scores(in fact, the least number of moves) are managed and stored by a hashmap with <key: username, value: array of scores>. Each game has each own corresponding file for its respective username and all of his/her scores.
  - Each game's scoreboard activity will also initialize Scoreboard System (under package 'system'), which will help determine and sort the high scores (and their corresponding users) in the specified format.
-

Back-end:
- The back-end of the scoreboard system consists of mainly three parts: system, score models and SaveScore tool. ScoreboardSystem is the general controller, score models are child classes of GameScore, and SaveScore is basically a management class for saving score files into our expected path.
- At first, whenever our scoreboard is called, our ScoreboardSystem will first try to get our models of scores and use algorithm to sort the models which contains information of scores. Our scores are wrapped up in each model in a form of a Map, while the

use of our models is helping wrap data and make dependency for the Scoreboardsystem by fit the map data into the format that the scoreboardsystem desires.
- Whenever the front-end part of our scoreboard is ready, the scoreboardsystem will use displayScore method to return the processed data (usually the sorted data), a SequenceBundler, to the view, and then, according to the implementation of our BasicScoreActivity, it has method such as "displayScore" as well to display the data on the screen.
- Classes involved are: ScoreBoardSystem, BasicScoreBoardActivity, GameScore(model), SlidingScore(inherited from GameScore), SudokuScore(inherited from GameScore) and TfScore(inherited from GameScore).

A little about design:

- For aesthetics and consistency purposes, the entire design of the program is revolved around the central color scheme as follows (selected from Adobe Color CC):
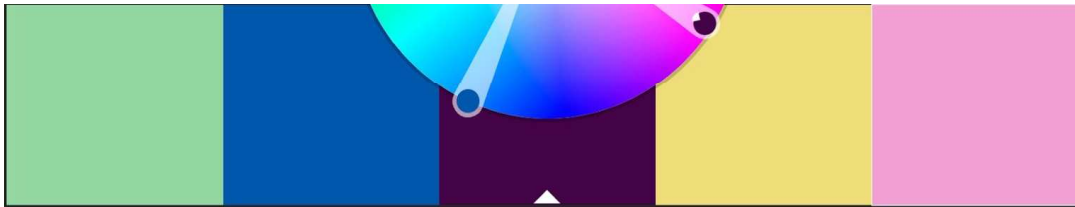
- The color scheme is not only applied to font colors, but also to backgrounds and images/drawables as well (using the "match color" tool from photoshop to match the original image color to this color scheme)
- For a more intuitive visualization, we have tried to incorporate the design concept of skeuomorphism as much as we can, which could be found throughout the entire program such as our Image Buttons (i.e. undo and profile button). A demonstrative example would be the game center, where each game is represented by self-explanatory Image Buttons to avoid as much excessive usage of text as possible and to achieve a more user-friendly and intuitive design.