Phase 2: Walkthrough

Group 0611: Brandon Elefano, Jay Parthasarthy,
Jonah Simpson, Robert Wang, Xinyi Zhang

December 6, 2018

1 Introduction/Games

Our group extended the sliding tiles app created in Phase 1 to include 3 additional games. We will demonstrate our snake, knights tour, and matching tiles games in this walkthrough.

The `MainMenuActivity` launches our tile game app. Users choose the game they want to play by toggling the game title. By default, the game that first appears when the app is launched is Sliding Tiles. When user clicks the title indicating they would like to play a different game, they can cycle through our four available game titles. Then, when launching a new game, a saved game, or a game specific scoreboard, we check the game title to load the correct game or scoreboard.

To be able to save games, the user must register for an account or log in on this screen. New functionality in phase 2 includes scoreboard appearing in the main menu. We will now demonstrate the creation of a new game.

The popup activity is called when a new game is called. For each game, users have the option of choosing one of several difficulties. In the back end, when start game button is pressed, the program takes the size chosen by the user and fetches the Bitmap for the picture of the correct board size, creates the board and tiles from the Bitmap, and initializes a new game with the size and background.

The main game activity is then called by the popup activity. This contains methods such as updating the user interface in one second intervals, updating game timer, tile buttons, and checks for game completion.

The game class and board class for each game contains game logic specific to each type of game.

- Sliding Tiles
  - Undo functionality: for games where undo is implemented, we push the state of the game to a stack for each move made. When the user wishes to undo, the state of the last move is popped
  - Save functionality: explained in important classes
  - Always winnable

- Snake
  - Dynamic
  - Undo functionality
- Knight's Tour
  - Undo functionality
  - Save functionality: explained in important classes
  - Game continues until you beat the game
  - Leaderboard is essentially a list of the last 10 users to successfully complete the board
- Matching Tiles
  - `MatchingTilesStartPopUp`: 3 possible difficulties
  - Save functionality: explained in important classes

2 Important Classes

- Main Activities
  - Each activity is a view, but still has controller elements- i.e. update timer
  - Each activity has Game as an attribute, which acts as the main model for the controller / view to interface with. An example is MatchingTileGame / Board.
- MovementController
  - This is initialized with every GameActivity, and is the main controller, deals with player unit.
  - It is generic enough to be used with any type of GameActivity- we have varying methods for either tap or swipe input.
- UserManager
  - Service - Binds to Activity Classes it interacts with
  - Links Save Games to Users - One save game per game, per player
    - Also stores information pertaining to the path of the image used for the game (often it is a default image contained within the res/drawable directory
- ScoreBoard
  - Works for all games- different games are saved with attributes that identify it to a speciic game.

- ImageToTiles
  - Slices a bitmap image up into a grid with even sized tiles
  - Saves images in internal memory (temp)
  - Works for all games

3 Design Patterns

- Module/View/Controller - module - the board class stores tiles and other information. Also in the popup activity for each game, when a button is pressed, it stores the user selection as an intent and pass it to the game activity. The usermanager and scoreboard are both modules that stores player information and player score.

  view - Pop-up activities that correspond to view files. The xml files also decide the layout of the board. The Oncreate method when a new page is created also generates the layout. controller - the logic in board class include endgame for checking end game condition, calculation of the game score upon completion and "makemove" that change the state of board. Controller also include the OnClickListener in interfaces that process what happens when a button is clicked.

- Strategy Design Pattern - different methods of implementing undo through undo interface. Different strategies are implemented inside the board class.

- Dependency Injection Design Pattern - an example would be that each game class depends on the game board class since each game includes an game board. However since the game generates a new board each time, it is an instance of hard dependency. the popup class for each game influences how the game board is generated

- Observer Design Pattern - the board and the game class of a game implement observable design pattern while the game activity class implements the observers. Whenever the method "touchmove" is called in a game class through a detected gesture that is passed to movement detection, the observable class informs the observer that a valid move is made to the board and the screen need to update the result accordingly.

4 Unit Test Coverage

- Low Test Coverage Reasons:
  - 13 Classes are activities; contain methods and functionality that cannot be tested purely for logic
    - Listeners, references to XML markup files

        * Activity Classes incorporate UI design; most of the logic occurs within the games themselves, which are thoroughly tested
    - Require serializable/mocking
    - Issues with Roboletric - Library that shadows Android functionality (UserManager - service)

- The ones with 100 percent test coverage are the games/boards themselves. We can directly test the logic with these classes to ensure each game works.
    - The SlidingTiles package contains 100 percent class coverage
    - The code that wasn't covered are getters and setters.
    - It initializes the board as being completed before each test, then adjusted based on each of the tests
    - Every test method consists of numerous different checks
        * getPreviousMoves - Checks if no previous moves are made, if a move made matches the white tile's previous position, and if another move is properly pushed to the top of the stack - Essential for the undo functionality
        * makeMove - test four valid moves (one in each direction), then tests trying to move the blank tile to the same tile, then trying to to move the blank tile across the board
        * GetTile - self explanitory; tests if the gettile returns the right tile, and tests if the tile isn't the wrong tile
        * GetState - Same Idea
        * GetMovesMade - checks if it starts at zero (when no moves are made) and makes sure it increases as moves are made

5 Scoreboard

5.1 The `IScoreboard` interface

`IScoreboard` represents the scoreboard data (the Model). It has two methods for retrieving scores:

- `getGlobalHighScores` retrieves the global high scores for a given game type.
- `getUserHighScores` retrieves the high scores for a given game type and a particular user.

These methods return a sorted list of the `ScoreField` type, which simply contains getters for a user-name and a score value. The returned `ScoreFields` are the high scores to be displayed on request.

The only other method `addGame` is meant to be called when a user finishes a game, to update the score data with this game if necessary.

5.2 Use and ownership

The `LeaderBoardActivity` is the view and controller for the score-board model. It maintains an implementation of the `IScoreboard`. When the activity loads from a game activity, it will add the game to the scoreboard. Based on the game type and UI selectors for local/global and game variants, the activity queries the `IScoreboard` and displays the returned `ScoreField`s in a list.

5.3 Implementation

The class `Scoreboard` implements `IScoreboard`. Internally, it represents each leader-board as a sorted list of `ScoreField`s. This allows queries to be efficient because the lists are already sorted.

When games are added to the scoreboard, the class inserts the score into the appropriate lists. If the length overflows the number of maintained scores (10), the lowest score is removed.

We achieve persistence of the score data by writing to a file all the scores and game types. In the `Scoreboard` constructor, these scores are read from the file and inserted.