Walkthrough

1. What is your unit test coverage?

We tested all the models that have logic in it and we test all the controllers.

We have over 100 test cases.

When testing controllers, we test the behaviour correctness of the controller when certain method is invoked. For example, in MovementController.class for Push the Box game, we created a new MovementController instance and assign several MapManager with maps that are the results of all possible movements. Each test case is a movement of the person on different maps, including when the person moves in four directions to a blank floor, towards the wall, or pushes the box. We test all valid and invalid movements.

Since we used MVP design pattern, view classes and model classes without logics are excluded from test coverage. Most activity classes are excluded from test since what they do are basically assigning elements on the view.

All the controllers are 100% coverage except for the adapters or methods involve Threads.

2. What are the most important classes?

The most important classes for the three games are the game presenters and game managers.

For example, for Memorization Master, the game presenter controls all logics that present to users and handle user inputs to interact with the game logic. Game manger holds on to the information to update game status, and when presenter gives signal, manager will update information that need to be presented. Game presenter handles the game's logic. When to display to user, when to accept user input, when to stop user input either because game is over or it's displaying time, etc. Other classes only handle view for the users according to logic from these two classes.

Specifically, in Memorization Master, presenter first starts a cycle that display sequence of Tiles that users need to remember and set iterator to the start of the sequence. The sequence of Tiles is stored in a field in game manager. After displaying is finished, presenter will enable user input. When user tap on a tile, the method "verify" will be invoked. When verify success, the iterator will move to next item that needs to be verified. Alternatively, if verify failed field "life" will decrease by 1 and iterator remain the same. When life reach 0, the presenter will stop all user input and display game over dialog that allows user to see their score and choose to either quit or play again. When iterate through the whole sequence, presenter will call manager to update the sequence and start cycle again.

For Push the Box, the game presenter gets user inputs detected from the view, and the presenter calls game manager to update the map. The game manager stores the element information on the game map. The presenter calls the manager to update changes when user gives input to move the person or undo the previous movements they made. Then the manager will verify if the request of movement or undo is valid, if so, information of the map

will be updated, and the presenter will be notified. Lastly, the presenter will check whether the game should end, and present the result on the view. The presenter also will autosave the game when information is updated.

The structure of the Sliding Tiles game is similar to Push the Box game. Basic logics are in game manager. The presenter gets input from view and calls the manager to process changes. The manager stores the information of the tiles and notify presenter when changes are made. The presenter then let the save manager to save current state of the game and calls the view class to display changes for the user.

3. What design pattern did we use?

We use iterator, observer, decorator, dependency injection, and strategy design pattern.

Decorator Design Pattern

When designing the model used by all three games: UserManager, SaveManager, and ScoreManager, the strategy design pattern is applied. Since all managers have different responsibilities, to follow the single responsibility principle, we have to separate those classes. However, there are some functions that will be shared by all these classes. To solve this issue, we decide to put those shared function into a class and store the reference of the class in the managers' classes. We first create three data streams, UserDataStream, SaveDataStream and ScoreDataStream which prepare the data for the UserManager, SaveManager, and ScoreManager respectively. And then, we use these data streams to decorate the GlobalDataStream which contains the shared function used by all data streams. By doing this, each specific data stream derived new functionality based on the GlobalDataStream. Another benefit is that we follow the interface segregation principle. Each manager will only possess the interface that provides the necessary functionality.

Iterator Design Pattern

BoardManager and MemoManger both implements iterable interface which is beneficial since the algorithm of the traversals are encapsulated in the class and make the class more cohesive.

Observer Design Pattern

In the "push the box game", the class Person implements the observable interface while the class BoxPresenter implements the observer interface. It ensures that the presenter will update the view once the person makes a valid move. The Board and the BoardGamePresenter also follow this pattern.

Dependency Injection Design Pattern

The presenter needs context to update or read the model, but it is a bad practice to store the reference of the context in the presenter since it creates unexpected dependencies between the model and the view. To prevent the coupling, dependency injection is used. We inject the context from the activity to the presenter every time when a method that needs context is invoked. For instance, UserPresenter is responsible for login and signing up the user. It needs

context to access the local file in order to fulfill its responsibility. Therefore, the context is passed through most of the methods in UserPresenter from activity to let the presenter handle the logic. Dependency Injection Design Pattern boosts the unit testing since it is easier to take apart the program that has low coupling and test each part individually.

Strategy Design Pattern

When calculating scores and shuffling the board, different methods might be applied. MemoScore, TileScore, and BoxScore all need different strategies to calculate the total mark. Therefore, different score calculators that implement ScoreCalculator interface are passed into the constructor when initializing the ScoreManager class. The ScoreManager class will then use the given calculator to calculate the score for each specific game. Strategy Design Pattern helps to obey the open-closed principle since we only need to create a class that implements the ScoreCalculator interface in order to extend the functionality of the program.

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

For load and save scores:
First of all, all the information are stored in serialization files. Therefore, all information that of Scoreboard needed are kept in the serialization files. Therefore, when the game is finished, we append the information of scores back to the serialization file. When we handling the scores in scoreboard-manager, the first step is to take scores of all games from the serialization files from score-manager and apply the filter to get scores specific to a game or a user since each score object and score-manager can filter out scores for specific games based on scores using score-manager. Since each game has its own logic, Timing and place for saving score to serialization file for each game is different. In Push Box, scores are stored when a user finishes a level. In Memorization Master, scores are saved when the current-user clicks the wrong tile.

For Score-processing after load the all the score:
Secondly, score-object track username and scores of the user for the specific level of the specific game, and we have subclass inherit score_object when getting the score_object when getting the score. This way, we can get all scores for specific game from score-manager. For each subclass score-manager, there is methods to get scores from highest to lowest for specific games. There is also a method to get highest three scores for each user of each level of difficulty of the game. For example, in Memorization Master Game, there is a Memo Score Board Manager, and a MemoScore. Memo Game Score Board Manager has a method to get all scores of the Memorization Master Game from highest to lowest. Also, there is a method to get scores of the specific user of a specific game in all rounds.

For displaying the scores:
Thirdly, after processing, scores are used to generate a view. When displaying the scores, the highest 3 score of the user are used to set texts under highest 3 score category. Also, a list of

score-objects from highest to lowest is acquired by score-board. The list is used to populate to view in the activity.