CODE WALKTHROUGH

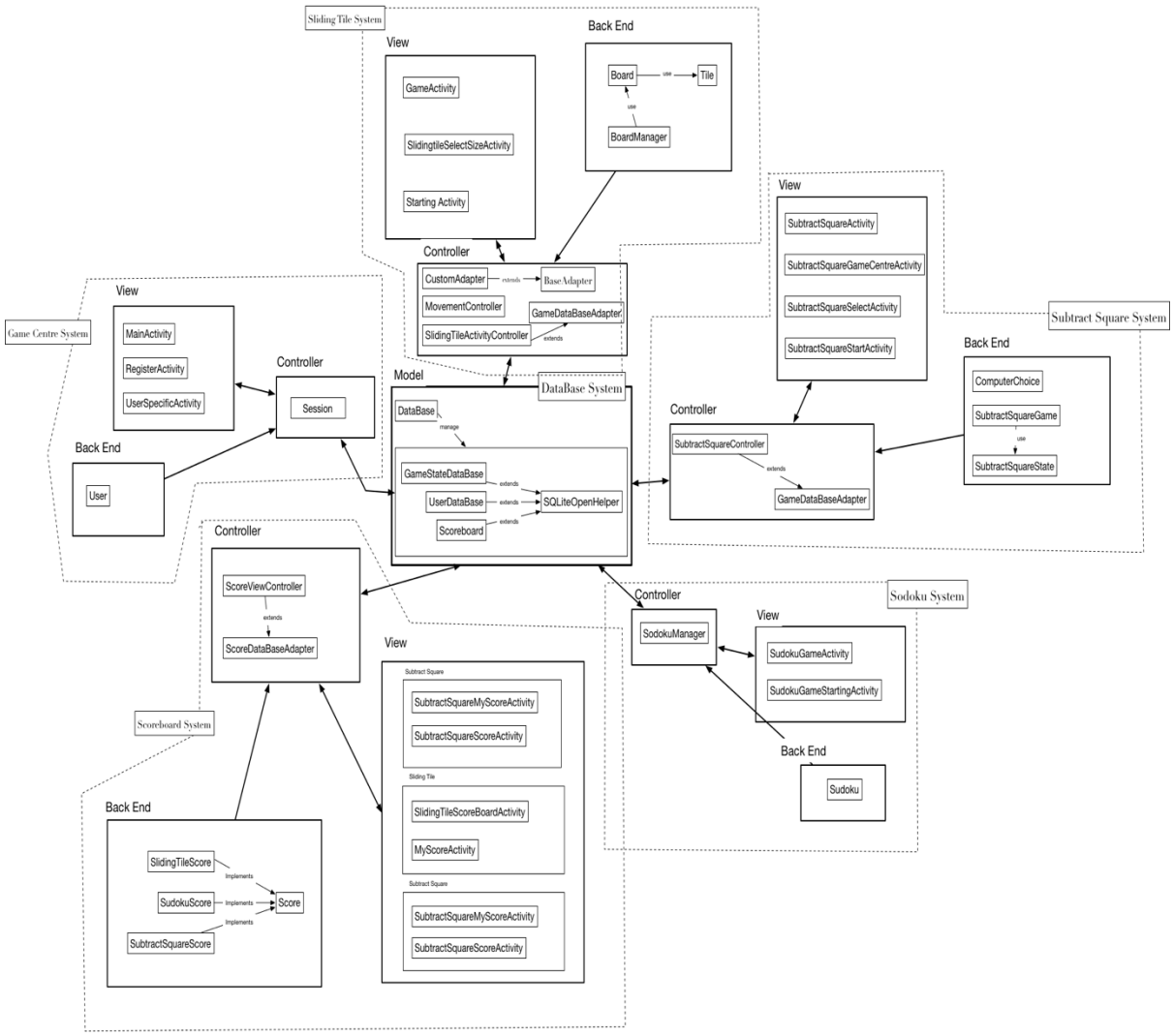
Game Centre 2.0, by group_0621

Agenda:

1) Structure of program
2) Java Code
   - Scoreboard implementation
   - Design pattern
   - Unit test
   - Other code proud of
3) Q&A

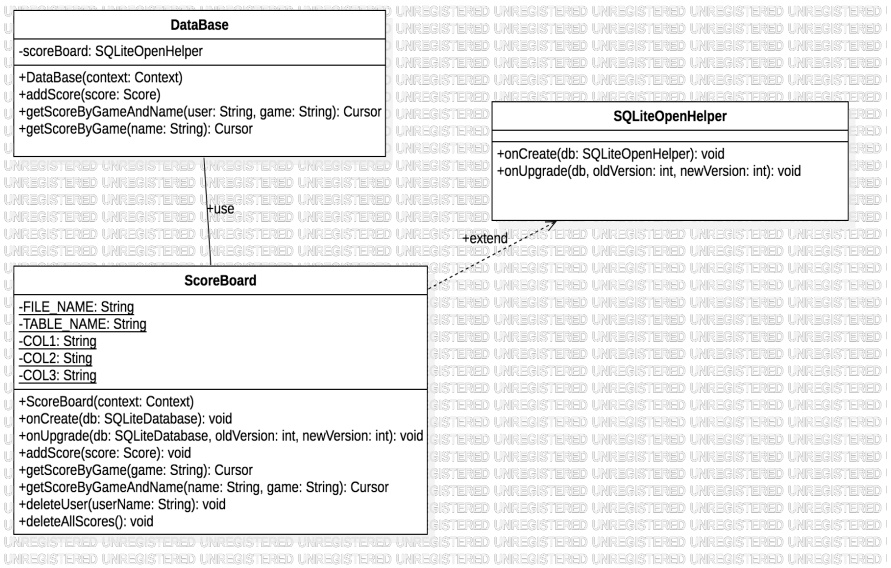
---

Program Structure:

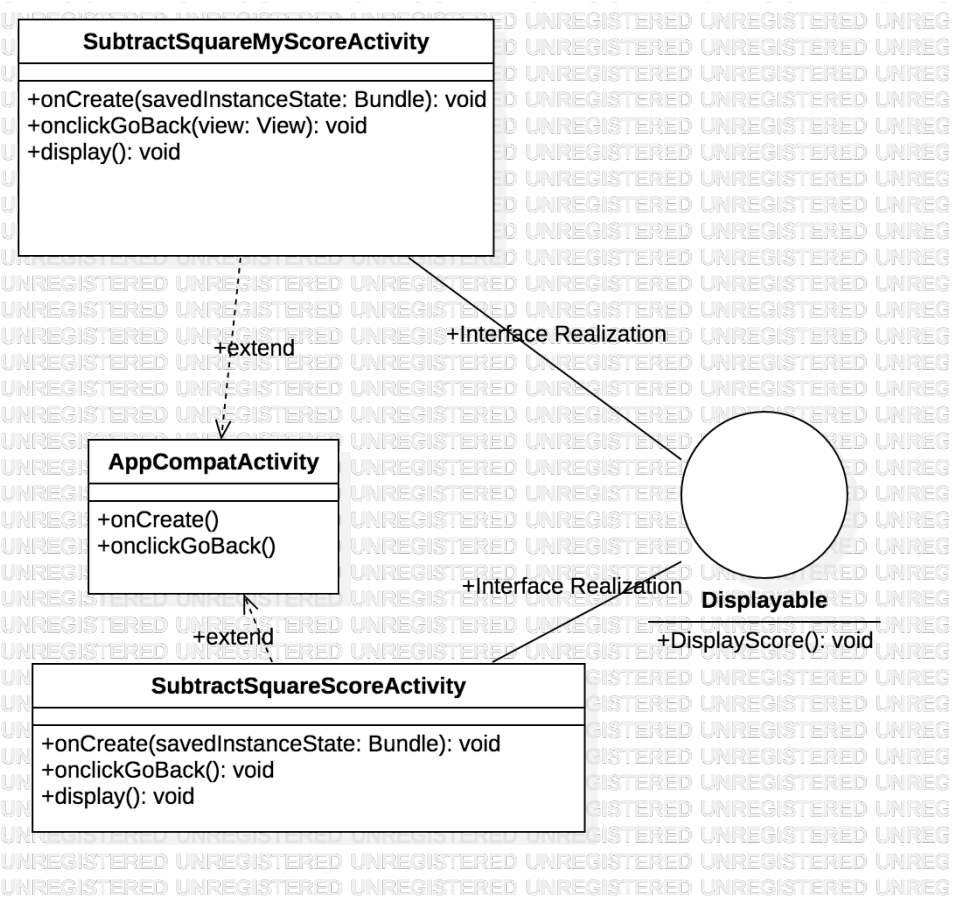


Scoreboard Implementation:

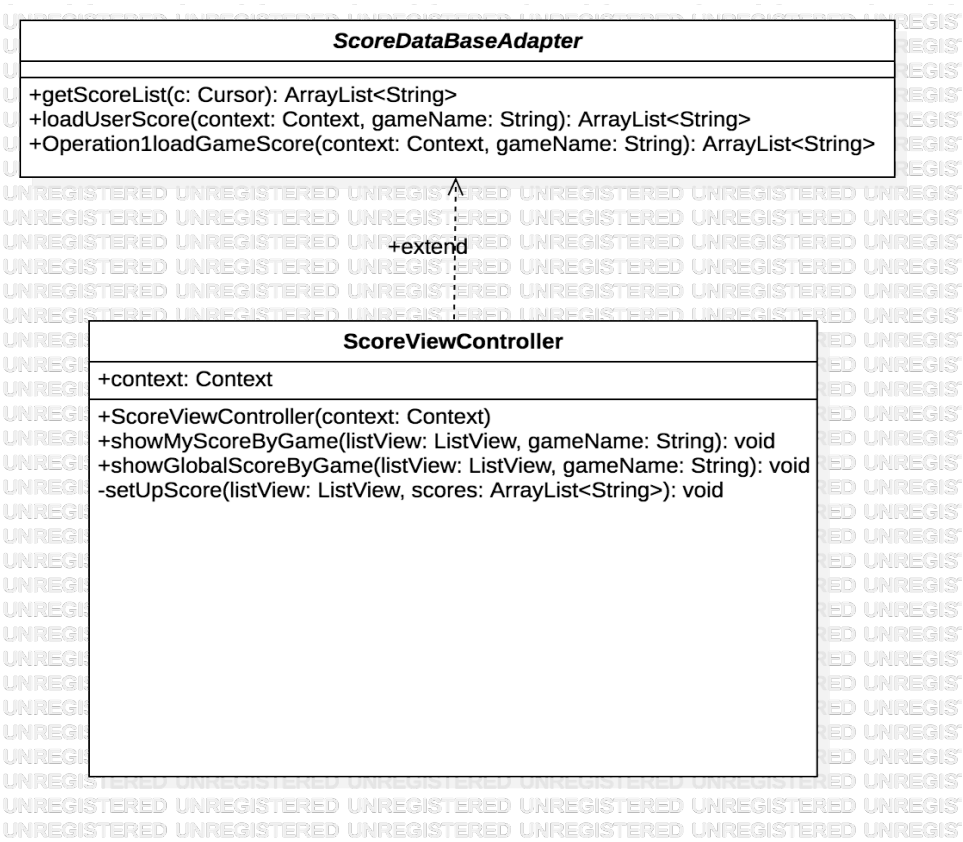
a) Model:



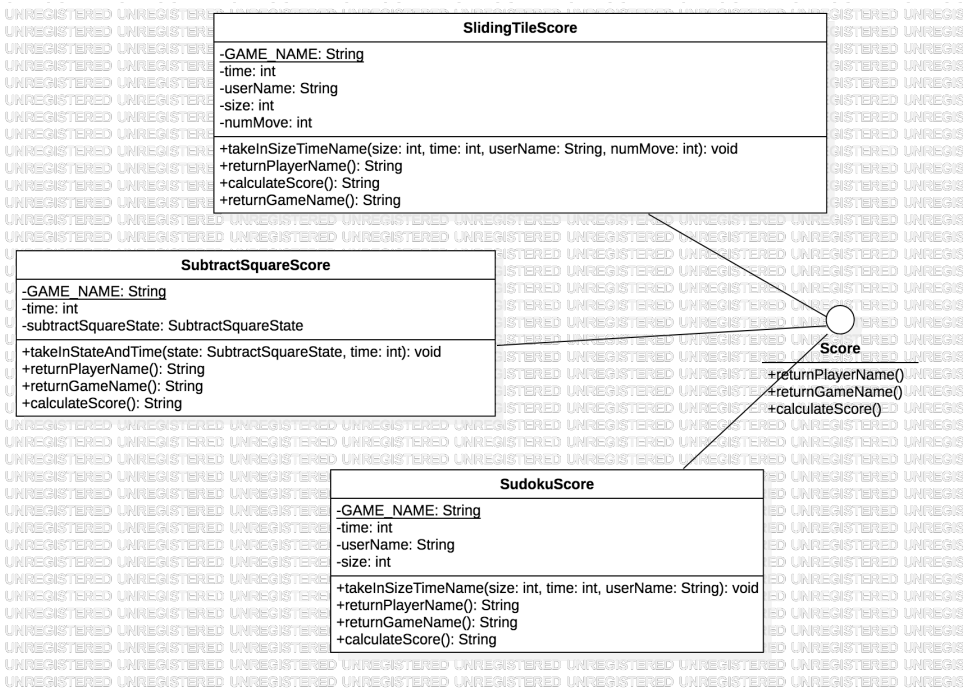

b) View (Example: subtract square):



c) Controller:

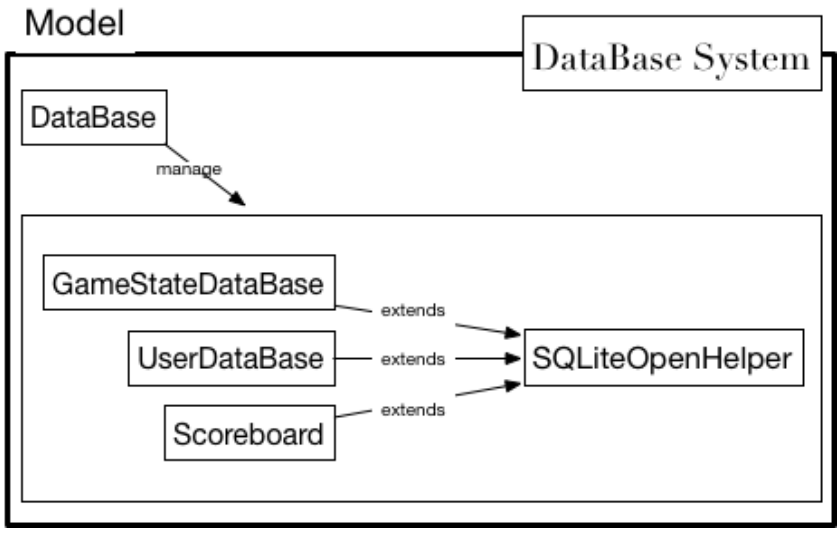



d) Back End:



Design Pattern:

a) Facade:




In database system, we have three different classes to store game states, users and scoreboards separately. So, every time we need to access different classes to get different kind of data. In order to solve this problem, we use façade design pattern to provide a unified class, which is DataBase class in our case, to a set of classes (GameStateDataBase, UserDataBase, Scoreboard) in a subsystem. As a result, we can access all data by just calling DataBase, which makes the subsystem easier to use.

b) Singleton:

In Game Centre system, we use Session class to store information of the current user. Since there must be only one current use, we use Singleton pattern to implement this class.

The single instance is a private static attribute. The accessor function is a public static method:

```
public class Session {

    private static Session session;

    private static DataBase dataBase;

    private static User currentUser;

    private Session() {
    }

    public static Session getInstance(Context context) {
        if (session == null) {
//              userDataBase = new UserDataBase(context);
            dataBase = new DataBase(context);
            session = new Session();
        }
        return session;
    }
```

c) Model-View-Controller:

We consider each separate system consists of data, presentation of that data that is responsive to changes in the model and translator between interaction between

User and View and actions for the Model to perform. That's why we use Model-View-Controller pattern in every system as shown in the 'structure of program' part.

d) Factory Design Pattern:

Since different games have different ways to calculate score, we implement three classes to represent score of each game (SubtractSquareScore, SudokuScore, SlidingTileScore). Therefore, we need to call different classes, when we refer to the score of different games. To solve this problem, we provide a class called ScoreFactory for creating families of related objects without specifying their concrete classes by using Factory design pattern. For more details:

```
/**
 * return different score class based on game name.
 *
 * @param gameName the name of the game
 * @return Score
 */
public Score generateScore(String gameName) {
    if (gameName == null) {
        return null;
    } else if (gameName.equals(SUBTRACT_SQUARE)) {
        return new SubtractSquareScore();
    } else if (gameName.equals(SLIDING_TILE)) {
        return new SlidingTileScore();
    } else if (gameName.equals(SUDOKU)) {
        return new SudokuScore();
    }
    return null;
}
```

Above is the method under ScoreFactory. It generates different score class based on the parameter 'gameName'. As a result, every time we want to get a score class of a specific game, we only need to call ScoreFactory.

e) Adapter Design Pattern:

Since we need to convert the interface of database into another interface clients expect, we use adapter design pattern and implement adapter classes in each system. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. That's why we use it in every controller block as shown in the 'structure of program' part.

f) Iterator:

In SubtractSquareGame class, we want to implement a method to access past states sequentially without exposing its underlying representation. That is why we use iterator design pattern. In order to achieve that, we let SubtractSquareGame implements

iterable<SubtractSquareState> and write a iterator() method with helper class SubtractSquareStateIterator.

```
@NonNull
@Override
public Iterator<SubtractSquareState> iterator() { return new SubtractSquareStateIterator(); }

private class SubtractSquareStateIterator implements Iterator<SubtractSquareState> {

    int index = 0;


    @Override
    public boolean hasNext() { return index != numState(); }

    @Override
    public SubtractSquareState next() {
        SubtractSquareState result = getState(index);
        index++;
        return result;
    }
}
```

g) Observer Design Pattern:

Again, for SubtractSquareGame class, we want SubtractSquareController get notified and update automatically when previous one changes state. That's why we use Observer pattern to define a one-to-many dependency between these two classes. More specifically, we let SubtractSquareController implements Observer, and let SubtractSquareGame extends Observable.

Unit test:

Our unit tests are mainly for Back End classes. For example, in Sliding tile system, we test Board, BoardManager and Tile. In Subtract Square Game, we test ComputerChoice, SubtractSquareGame, SubtractSquareState. In Sudoku system, we test SudokuManager. Also, we choose to test these classes because they are the core of our project.

We don't test Database and GameCentreActivity, because they are the front end of the project, which are used to store data and display. Therefore, they don't have logic.

The best unit test class in our project should the unit tests for Board, BoardManager and Tile in Subtract square system. All of them cover 100%.

Other code:

a) How we generate Sudoku.
b) …