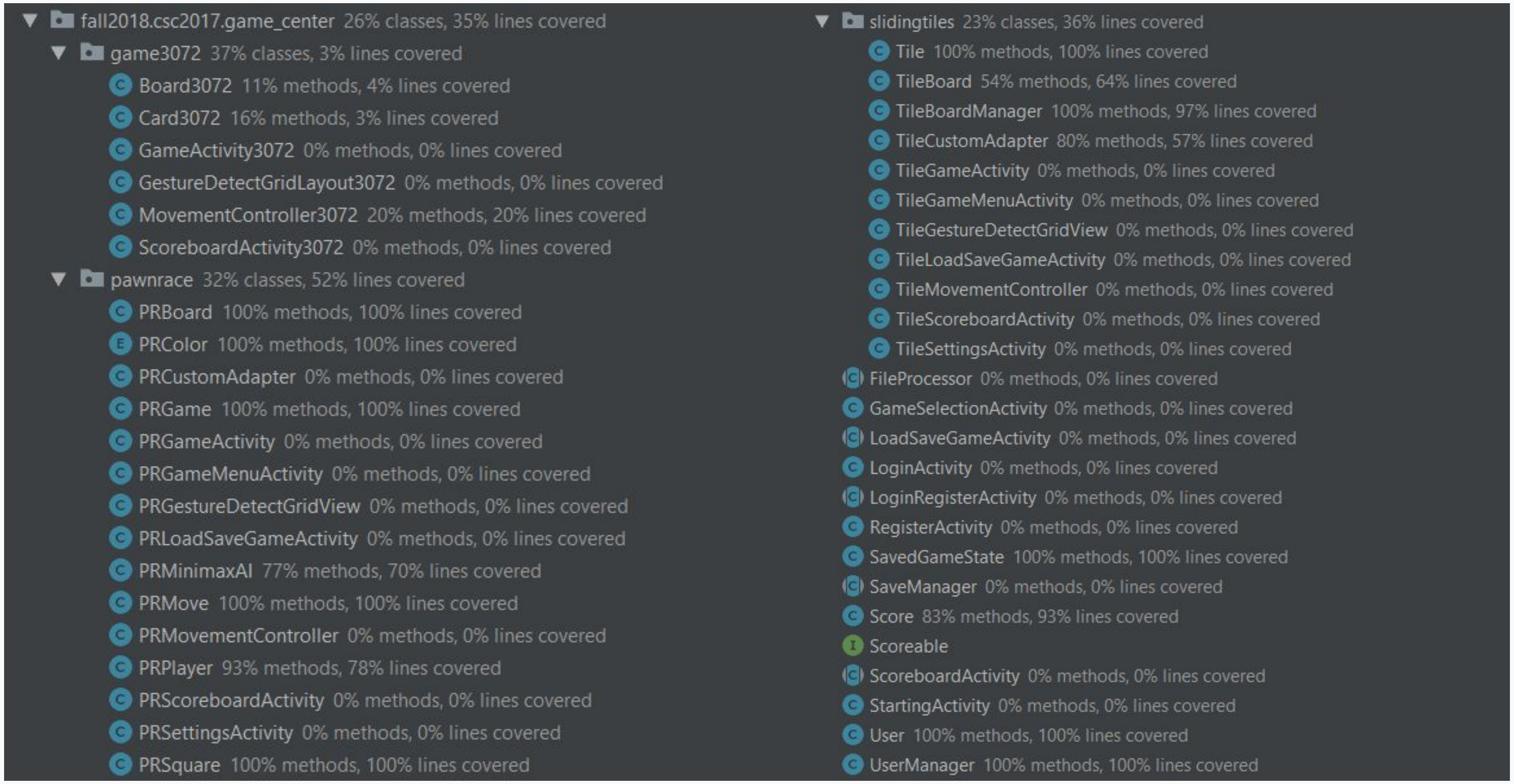
Game Center Presentation

Group 0641

Outline of Presentation

- Code Walkthrough
    - Class Structure
    - Most Important Classes
    - Log-in / Register System
- Scoreboard system
- Load/save system
- Game selection
- Game implementations
    - 3072
    - Pawn Race
    - Sliding Tiles
- Design Patterns
- Testing
    - Code coverage

Class Hierarchy Overview

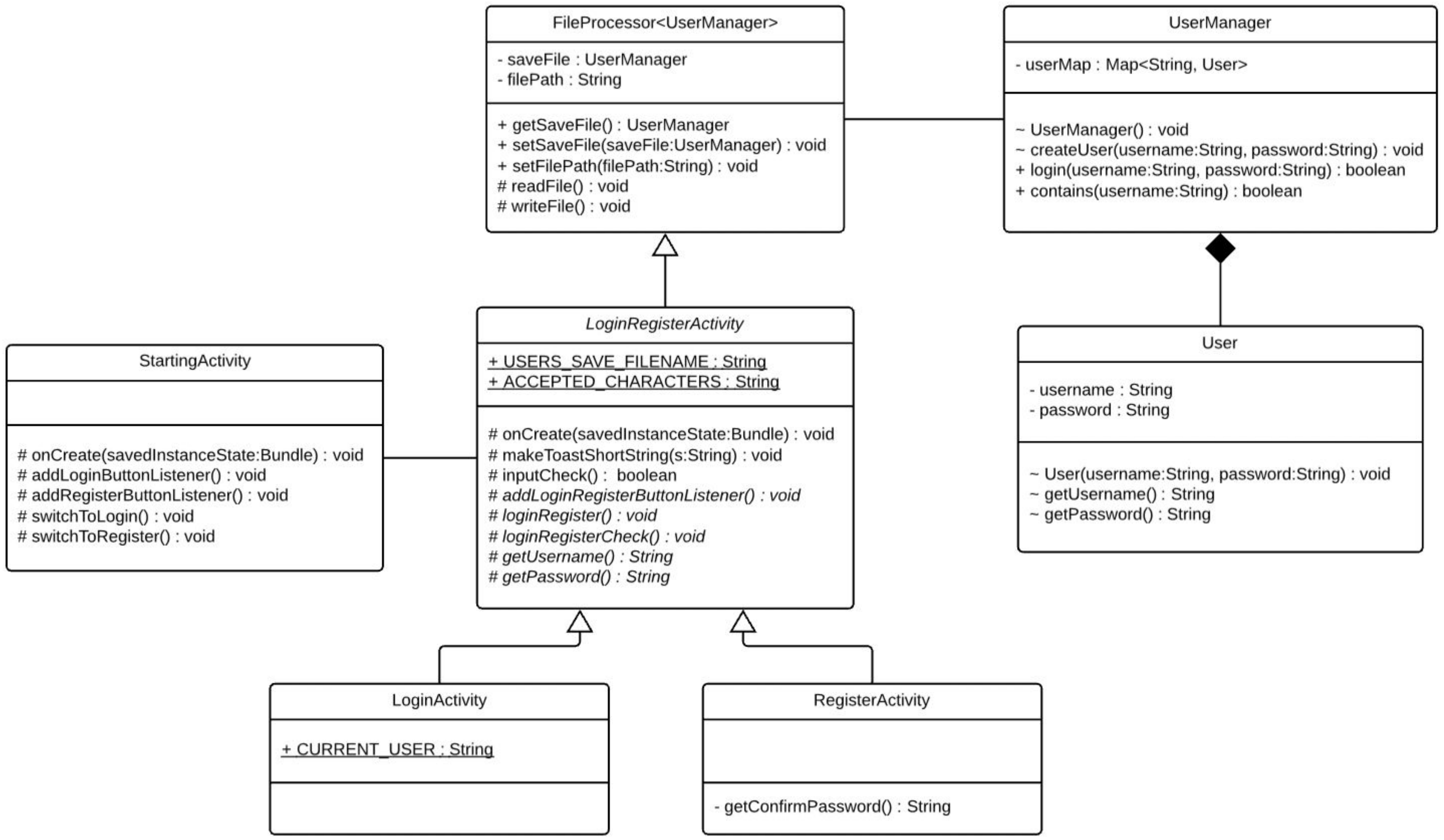


```
▼ fall2018.csc2017.game_center 26% classes, 35% lines covered
  ▼ game3072 37% classes, 3% lines covered
      Board3072 11% methods, 4% lines covered
      Card3072 16% methods, 3% lines covered
      GameActivity3072 0% methods, 0% lines covered
      GestureDetectGridLayout3072 0% methods, 0% lines covered
      MovementController3072 20% methods, 20% lines covered
      ScoreboardActivity3072 0% methods, 0% lines covered
  ▼ pawnrace 32% classes, 52% lines covered
      PRBoard 100% methods, 100% lines covered
      PRColor 100% methods, 100% lines covered
      PRCustomAdapter 0% methods, 0% lines covered
      PRGame 100% methods, 100% lines covered
      PRGameActivity 0% methods, 0% lines covered
      PRGameMenuActivity 0% methods, 0% lines covered
      PRGestureDetectGridView 0% methods, 0% lines covered
      PRLoadSaveGameActivity 0% methods, 0% lines covered
      PRMinimaxAI 77% methods, 70% lines covered
      PRMove 100% methods, 100% lines covered
      PRMovementController 0% methods, 0% lines covered
      PRPlayer 93% methods, 78% lines covered
      PRScoreboardActivity 0% methods, 0% lines covered
      PRSettingsActivity 0% methods, 0% lines covered
      PRSquare 100% methods, 100% lines covered
  ▼ slidingtiles 23% classes, 36% lines covered
      Tile 100% methods, 100% lines covered
      TileBoard 54% methods, 64% lines covered
      TileBoardManager 100% methods, 97% lines covered
      TileCustomAdapter 80% methods, 57% lines covered
      TileGameActivity 0% methods, 0% lines covered
      TileGameMenuActivity 0% methods, 0% lines covered
      TileGestureDetectGridView 0% methods, 0% lines covered
      TileLoadSaveGameActivity 0% methods, 0% lines covered
      TileMovementController 0% methods, 0% lines covered
      TileScoreboardActivity 0% methods, 0% lines covered
      TileSettingsActivity 0% methods, 0% lines covered
    FileProcessor 0% methods, 0% lines covered
    GameSelectionActivity 0% methods, 0% lines covered
    LoadSaveGameActivity 0% methods, 0% lines covered
    LoginActivity 0% methods, 0% lines covered
    LoginRegisterActivity 0% methods, 0% lines covered
    RegisterActivity 0% methods, 0% lines covered
    SavedGameState 100% methods, 100% lines covered
    SaveManager 0% methods, 0% lines covered
    Score 83% methods, 93% lines covered
    Scoreable
    ScoreboardActivity 0% methods, 0% lines covered
    StartingActivity 0% methods, 0% lines covered
    User 100% methods, 100% lines covered
    UserManager 100% methods, 100% lines covered
```

Most Important Classes

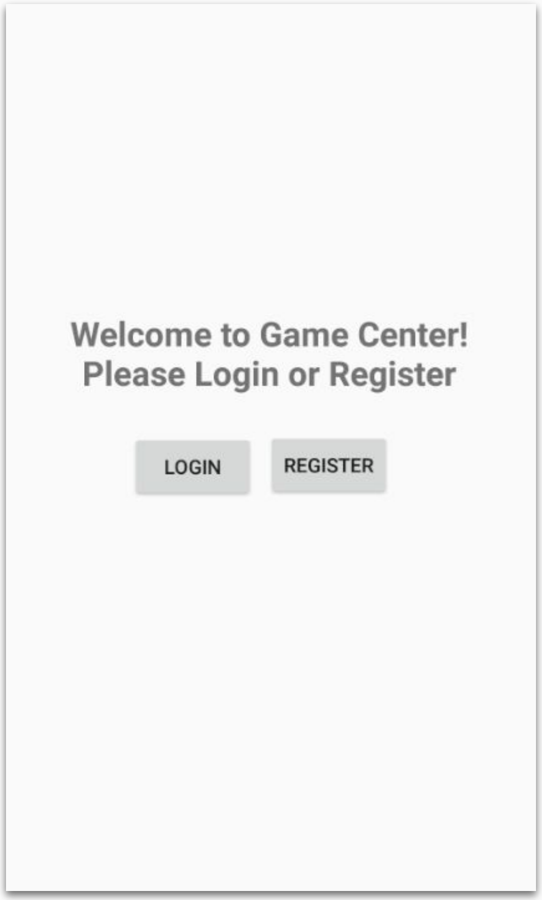
- **FileProcessor** - the single most important class in the whole project; the abstract class that serializes objects: used to store the UserManager, SavedGameState for every user in every game, and List<Score> for every game
  - Expanding on this, SaveManager, ScoreboardActivity, and LoadSaveGameActivity are also important abstract classes that extend this class, which contains abstract saving and scoring functionality for at least 2 of the 3 games
- For Tile game: **TileBoardManager** contains the most information about the state of the game and therefore is used for the SavedGameState
- For Pawn Race: **PRPlayer** is the class that is saved
- For 3072: **Board3072** is the class that is saved

Login/Register System Class Structure

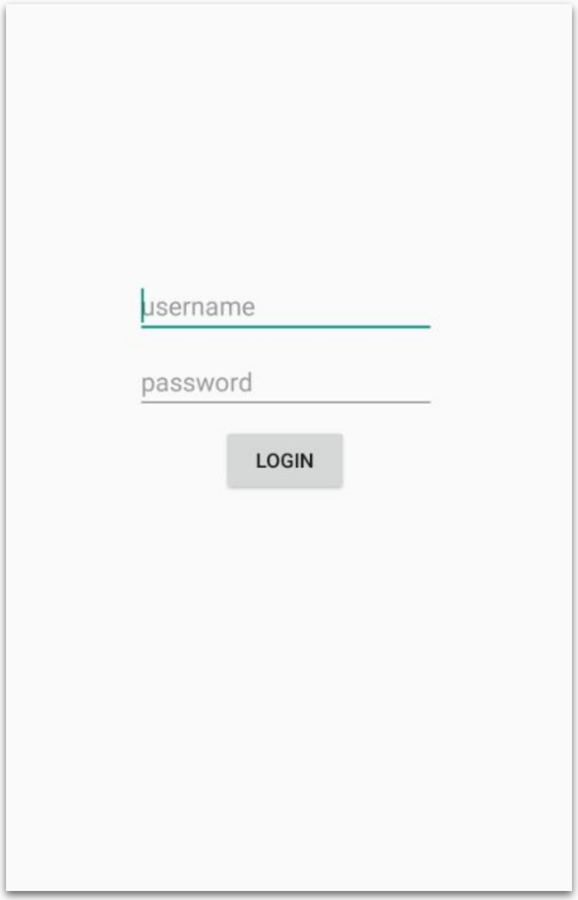


FileProcessor<UserManager>
- saveFile : UserManager
- filePath : String
+ getSaveFile() : UserManager
+ setSaveFile(saveFile:UserManager) : void
+ setFilePath(filePath:String) : void
# readFile() : void
# writeFile() : void
UserManager
- userMap : Map<String, User>
~ UserManager() : void
~ createUser(username:String, password:String) : void
+ login(username:String, password:String) : boolean
+ contains(username:String) : boolean
LoginRegisterActivity
+ USERS_SAVE_FILENAME : String
+ ACCEPTED_CHARACTERS : String
# onCreate(savedInstanceState:Bundle) : void
# makeToastShortString(s:String) : void
# inputCheck() : boolean
# addLoginRegisterButtonListener() : void
# loginRegister() : void
# loginRegisterCheck() : void
# getUsername() : String
# getPassword() : String
StartingActivity
# onCreate(savedInstanceState:Bundle) : void
# addLoginButtonListener() : void
# addRegisterButtonListener() : void
# switchToLogin() : void
# switchToRegister() : void
User
- username : String
- password : String
~ User(username:String, password:String) : void
~ getUsername() : String
~ getPassword() : String
LoginActivity
+ CURRENT_USER : String
RegisterActivity
- getConfirmPassword() : String

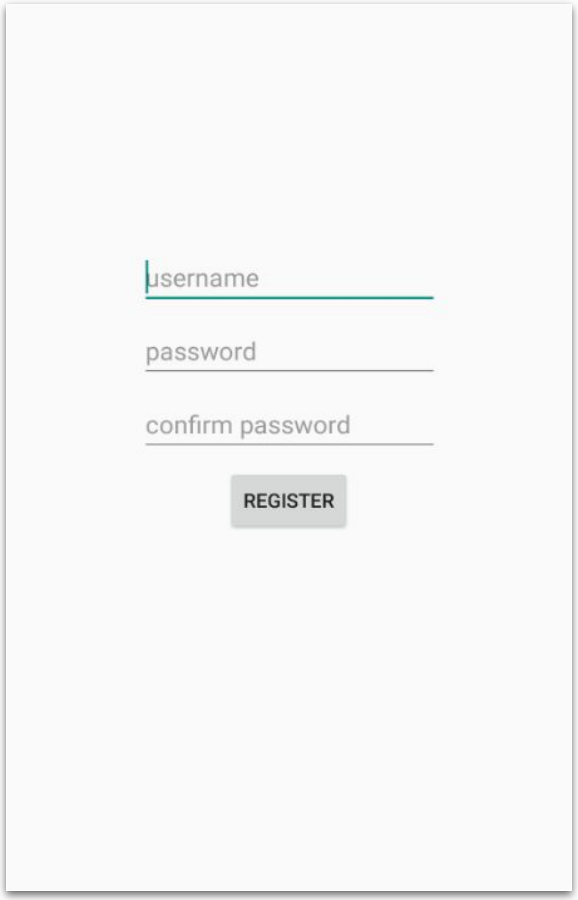
Login/Register System User Interface



User Onboarding Screen

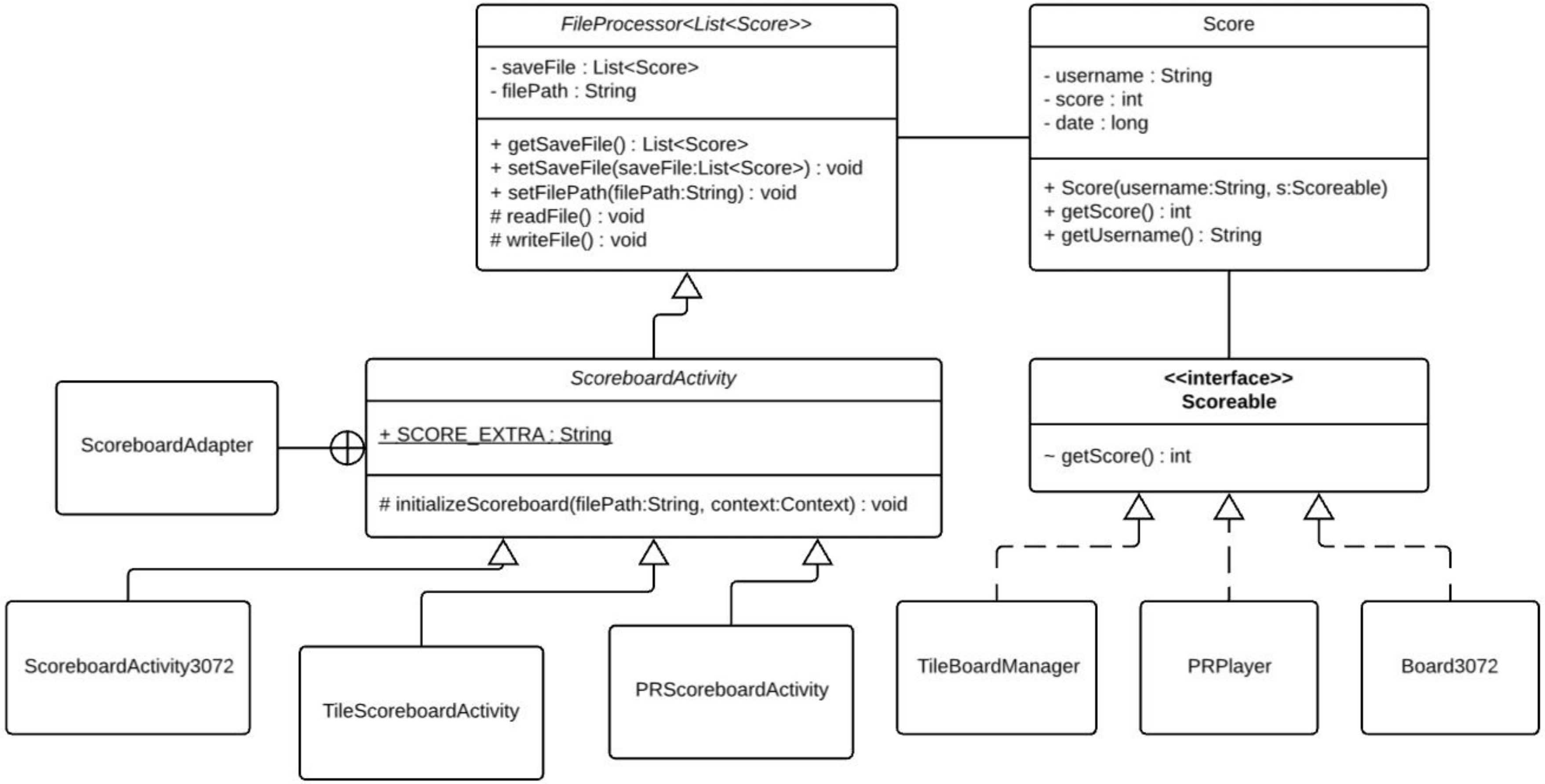


User Login

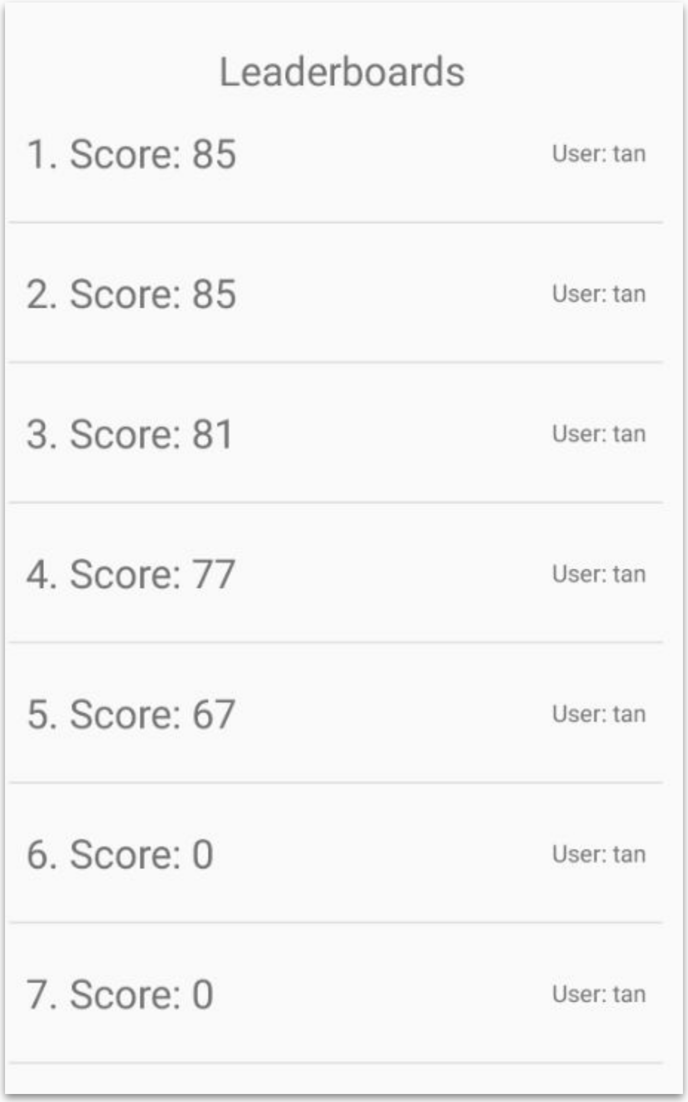


New User Registration

Scoreboard System Class Structure



Scoreboard System User Interface


Leaderboards

1. Score: 85	User: tan
2. Score: 85	User: tan
3. Score: 81	User: tan
4. Score: 77	User: tan
5. Score: 67	User: tan
6. Score: 0	User: tan
7. Score: 0	User: tan

- Scoreboards for Pawn Race and Sliding Tiles can be accessed outside of play by tapping on the scoreboard menu option

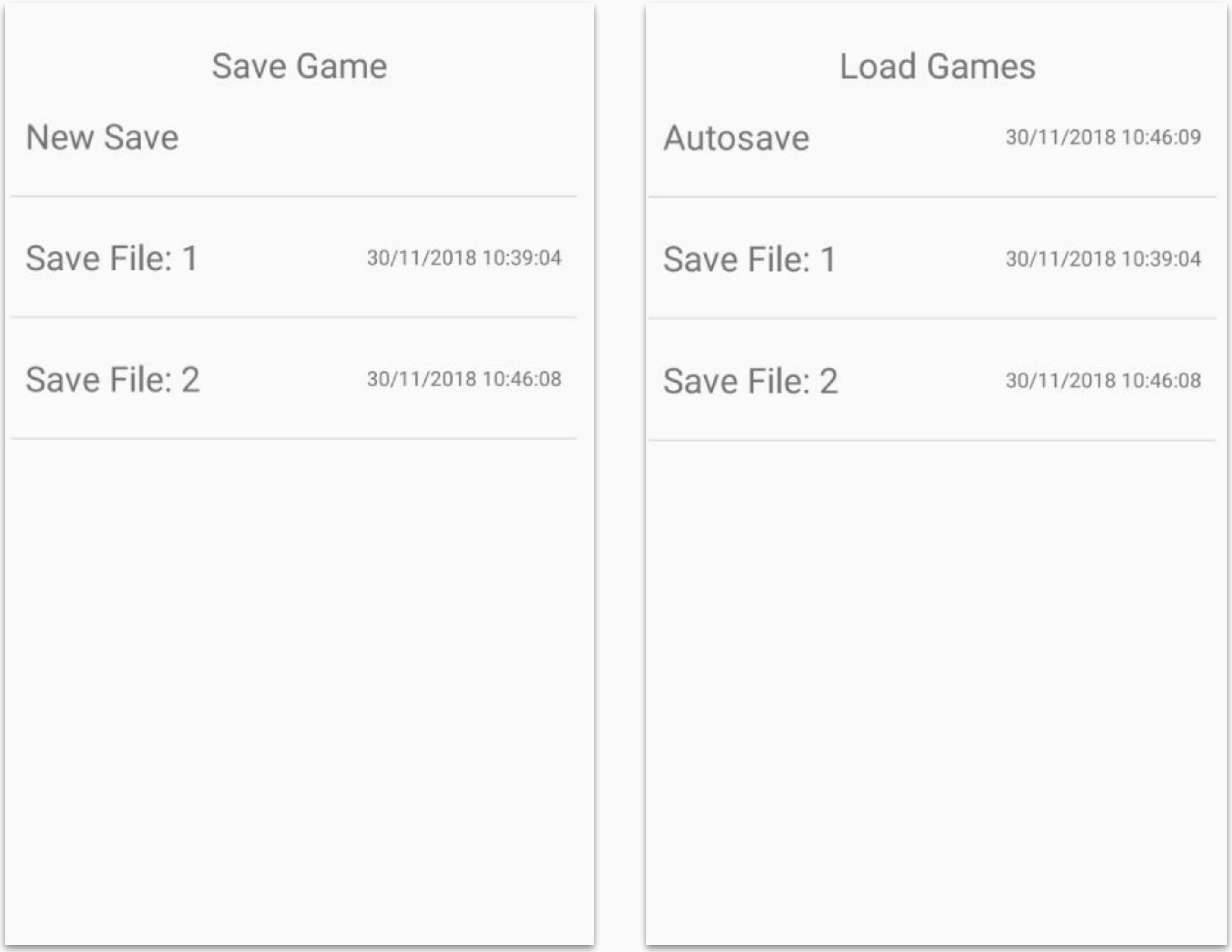
- After winning or losing a game, the scoreboard for that game with your newest play session included, will be displayed.

- Each game calculates scores differently!

Load/Save Game System

Similar to the Scoreboard system except with an extra step of going through the “SavedGameState” class for every game and every username.

Load/Save Game System



Save Game

New Save	
Save File: 1	30/11/2018 10:39:04
Save File: 2	30/11/2018 10:46:08

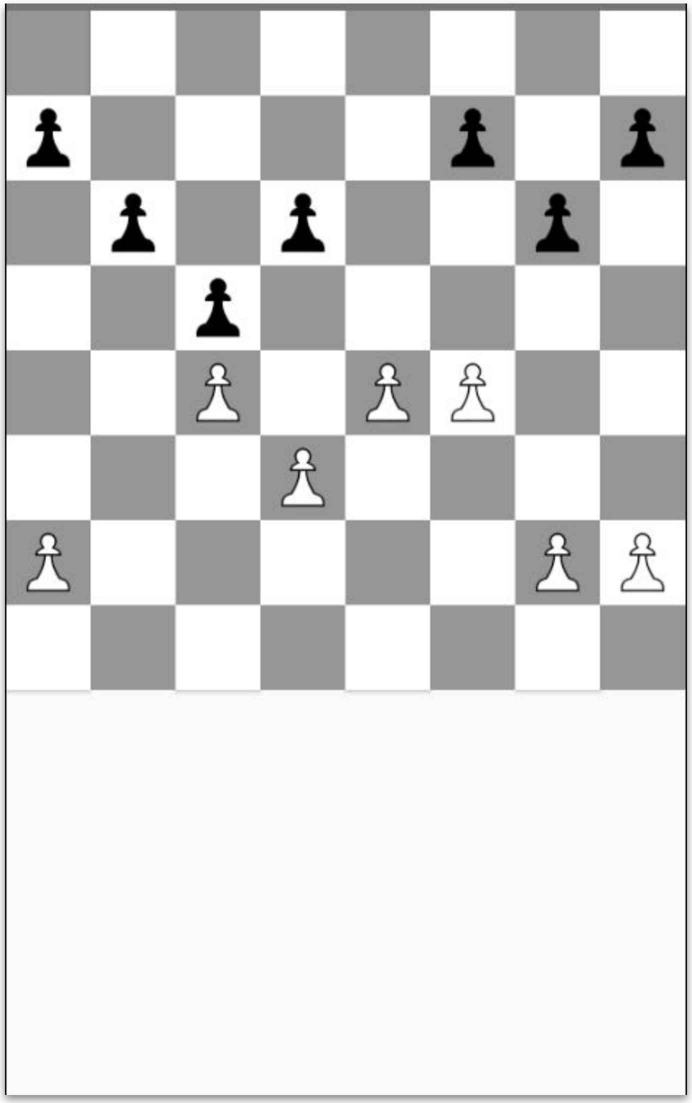
Load Games

Autosave	30/11/2018 10:46:09
Save File: 1	30/11/2018 10:39:04
Save File: 2	30/11/2018 10:46:08

Pawn Race Implementation

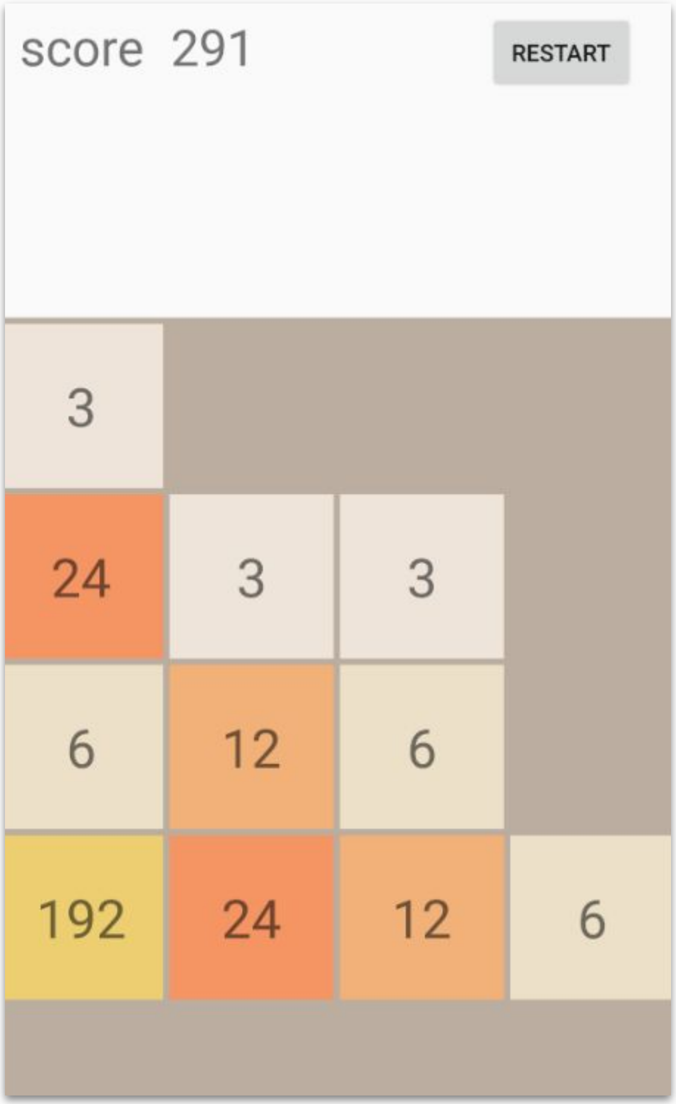
- Quite a complex game:
  - Player plays chess with the computer using only pawns
  - First to get a pawn to the other side of the board wins
  - Each game has a random gap on both the white and black side
- First implemented as a text input player vs. player game when I (Robert) was learning Java last year
- Association/encapsulation of classes as follows:
  - Player ⟵ Game ⟵ Board ⟵ Square
- Implemented a minimax algorithm to help the computer calculate moves with a completely original heuristic function (so it's not that good but still :/)
- Fully implemented undo/autosave/load/save/scoreboard functionalities in the same way as sliding tiles + additional settings

Pawn Race Implementation



3072 Implementation

- Somewhat complex game:
    - Similar to 2048 but instead of combining two multiples of 2's, two multiples of 3's are combined.
    - Game stops when 3072 is reached or no more available moves present.
    - The number, 3, are randomly placed on empty tiles after each movement
- Fully implemented Scoreboard functionality which appears after the game is over.



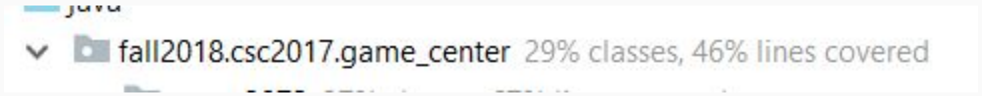
Design Patterns Used

- Model-View-Controller
  - Used for every game activity classes - where the game activity serves as the user interface, the grid view defines the layout, and the the movement controller acts as an independent controller to process user interactions.
- Observer
  - The observer pattern is used to notify the model to update the display when the game state has changed, this is used in every game.
- Adapter
  - Adapters are used for each game to add specific functionality to the already defined base adapters (for example, how each ListView displays the list).
- State
  - Our Pawn Race AI follows the state pattern by its dynamically adjusting depth - its calculation depth changes based on how many moves have been made.
- Strategy
  - The load/save game functionality uses the strategy design pattern as the load/save game activity is a singular activity that can either provide load functionality or save functionality.
- Template Method
  - Our file processor contains template methods that can store any serializable object. In our case, we use the same template for storing save files as well as scores.

Significance of Design Patterns

- Model-View-Controller
  - Classes are well organized as models, views and controllers are separated, which does not lead to Code Smells - Large Classes and Long Methods.
  - Easier to test code
  - Easier to add or modify code
- Observer
  - Allows one-to-many dependency between objects such that the change in one object will automatically notify its dependencies.
  - Usually used in Model-View-Controller patterns (as the View part)
- Adapter
  - Allows classes to work together that could not due to incompatible interfaces
- State
  - When an object’s internal state changes, it will appear as the class is changed.

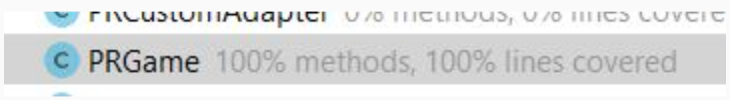
Unit Test Coverage



fall2018.csc2017.game_center 29% classes, 46% lines covered

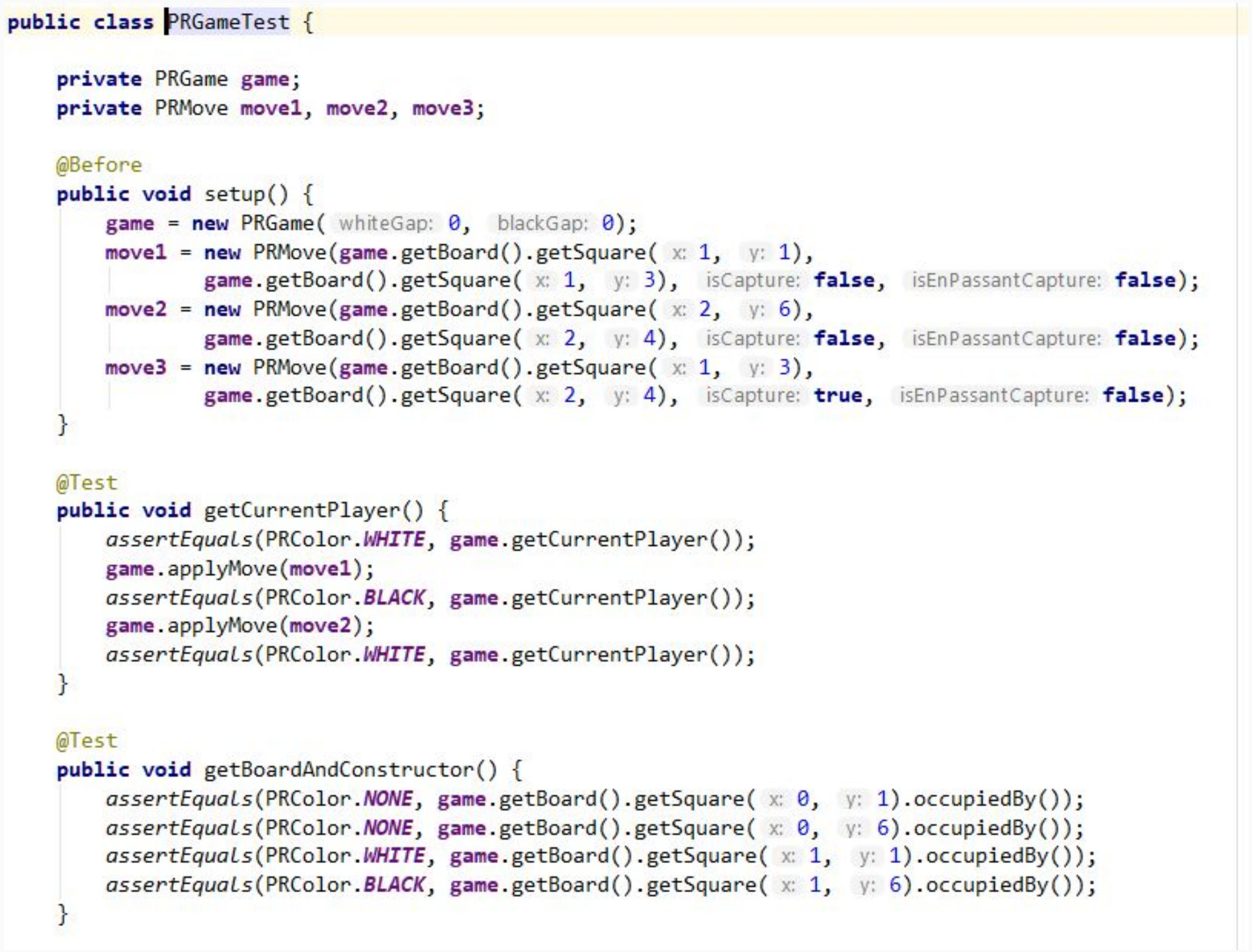
- 29% classes is because we have a lot of abstract activity classes
- 46% lines covered sounds about right as we roughly have the same amount of logic vs. front end UI implementations
  - As you can see, we use quite complex GridViews and ListViews, which were not required to be tested

Unit Test Coverage 100% Class - PRGame



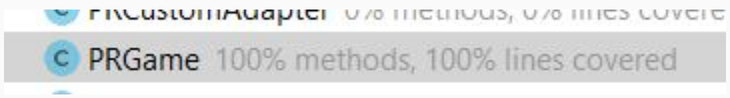
PRCustomAdapter 0% methods, 0% lines covere

PRGame 100% methods, 100% lines covered

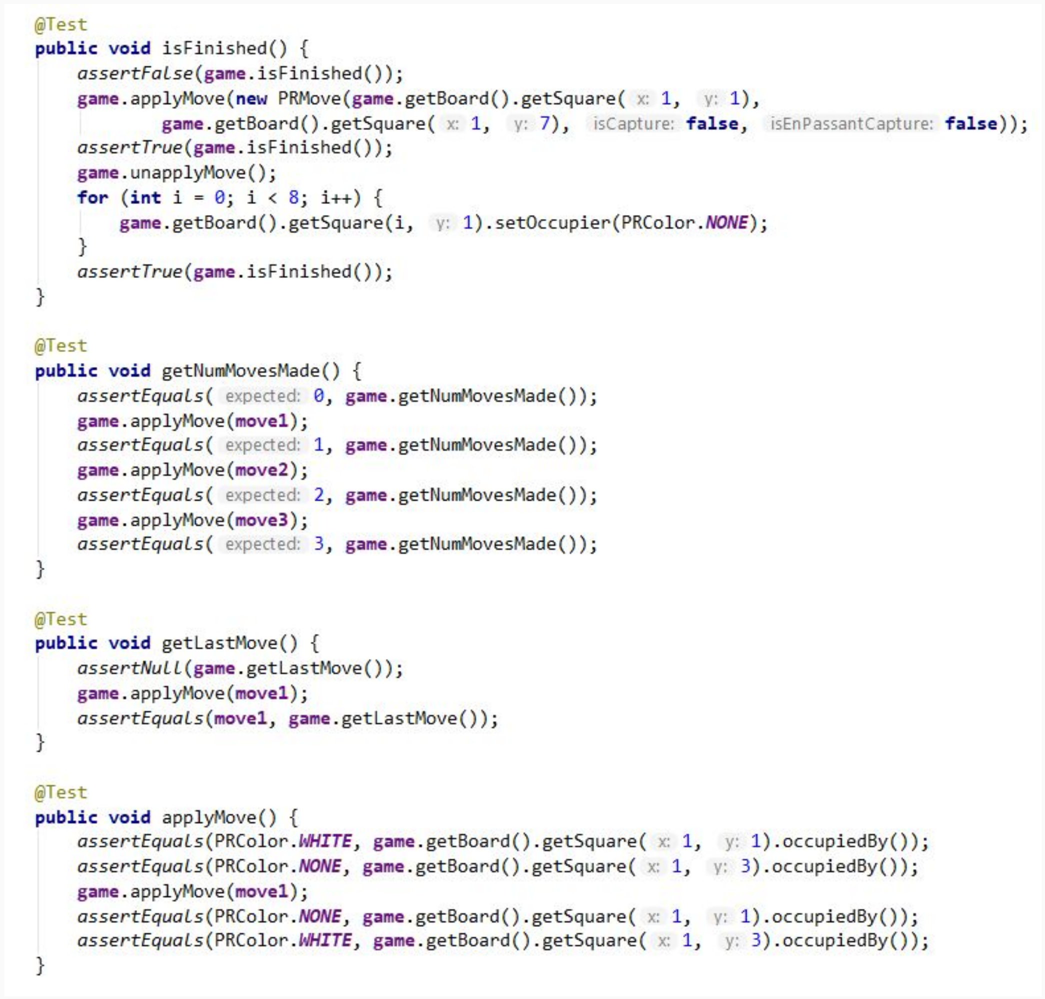


```
public class PRGameTest {

    private PRGame game;
    private PRMove move1, move2, move3;

    @Before
    public void setup() {
        game = new PRGame( whiteGap: 0,  blackGap: 0);
        move1 = new PRMove(game.getBoard().getSquare( x: 1,  y: 1),
                game.getBoard().getSquare( x: 1,  y: 3),  isCapture: false,  isEnPassantCapture: false);
        move2 = new PRMove(game.getBoard().getSquare( x: 2,  y: 6),
                game.getBoard().getSquare( x: 2,  y: 4),  isCapture: false,  isEnPassantCapture: false);
        move3 = new PRMove(game.getBoard().getSquare( x: 1,  y: 3),
                game.getBoard().getSquare( x: 2,  y: 4),  isCapture: true,  isEnPassantCapture: false);
    }

    @Test
    public void getCurrentPlayer() {
        assertEquals(PRColor.WHITE, game.getCurrentPlayer());
        game.applyMove(move1);
        assertEquals(PRColor.BLACK, game.getCurrentPlayer());
        game.applyMove(move2);
        assertEquals(PRColor.WHITE, game.getCurrentPlayer());
    }

    @Test
    public void getBoardAndConstructor() {
        assertEquals(PRColor.NONE, game.getBoard().getSquare( x: 0,  y: 1).occupiedBy());
        assertEquals(PRColor.NONE, game.getBoard().getSquare( x: 0,  y: 6).occupiedBy());
        assertEquals(PRColor.WHITE, game.getBoard().getSquare( x: 1,  y: 1).occupiedBy());
        assertEquals(PRColor.BLACK, game.getBoard().getSquare( x: 1,  y: 6).occupiedBy());
    }
```

Unit Test Coverage 100% Class - PRGame

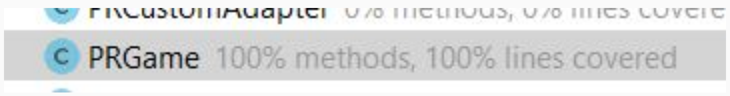


PRCustomAdapter 0% methods, 0% lines covere
PRGame 100% methods, 100% lines covered



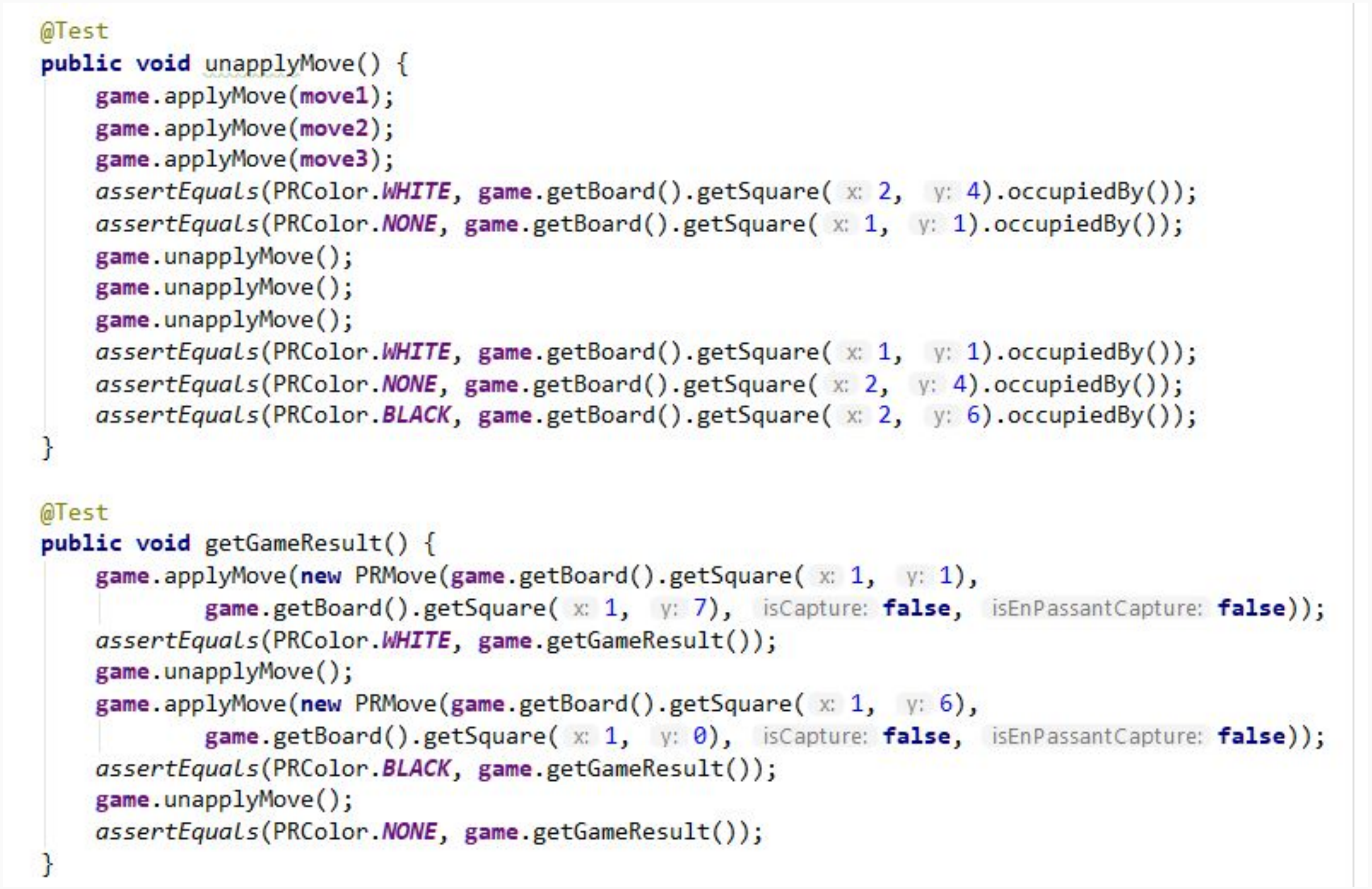
```java
@Test
public void isFinished() {
    assertFalse(game.isFinished());
    game.applyMove(new PRMove(game.getBoard().getSquare( x: 1,  y: 1),
            game.getBoard().getSquare( x: 1,  y: 7),  isCapture: false,  isEnPassantCapture: false));
    assertTrue(game.isFinished());
    game.unapplyMove();
    for (int i = 0; i < 8; i++) {
        game.getBoard().getSquare(i,  y: 1).setOccupier(PRColor.NONE);
    }
    assertTrue(game.isFinished());
}

@Test
public void getNumMovesMade() {
    assertEquals( expected: 0, game.getNumMovesMade());
    game.applyMove(move1);
    assertEquals( expected: 1, game.getNumMovesMade());
    game.applyMove(move2);
    assertEquals( expected: 2, game.getNumMovesMade());
    game.applyMove(move3);
    assertEquals( expected: 3, game.getNumMovesMade());
}

@Test
public void getLastMove() {
    assertNull(game.getLastMove());
    game.applyMove(move1);
    assertEquals(move1, game.getLastMove());
}

@Test
public void applyMove() {
    assertEquals(PRColor.WHITE, game.getBoard().getSquare( x: 1,  y: 1).occupiedBy());
    assertEquals(PRColor.NONE, game.getBoard().getSquare( x: 1,  y: 3).occupiedBy());
    game.applyMove(move1);
    assertEquals(PRColor.NONE, game.getBoard().getSquare( x: 1,  y: 1).occupiedBy());
    assertEquals(PRColor.WHITE, game.getBoard().getSquare( x: 1,  y: 3).occupiedBy());
}
```

Unit Test Coverage 100% Class - PRGame



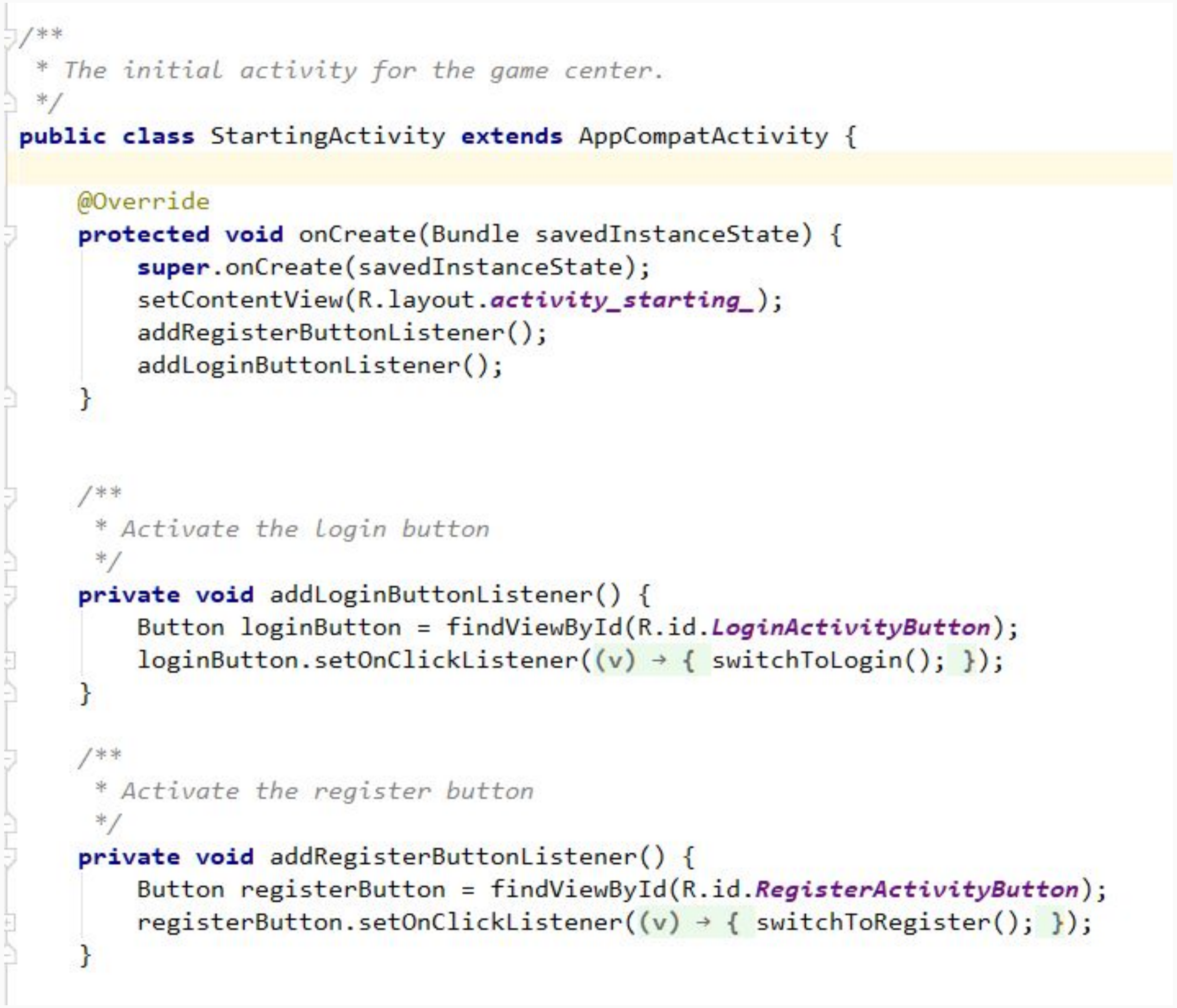
PRCustomAdapter 0% methods, 0% lines covere

PRGame 100% methods, 100% lines covered



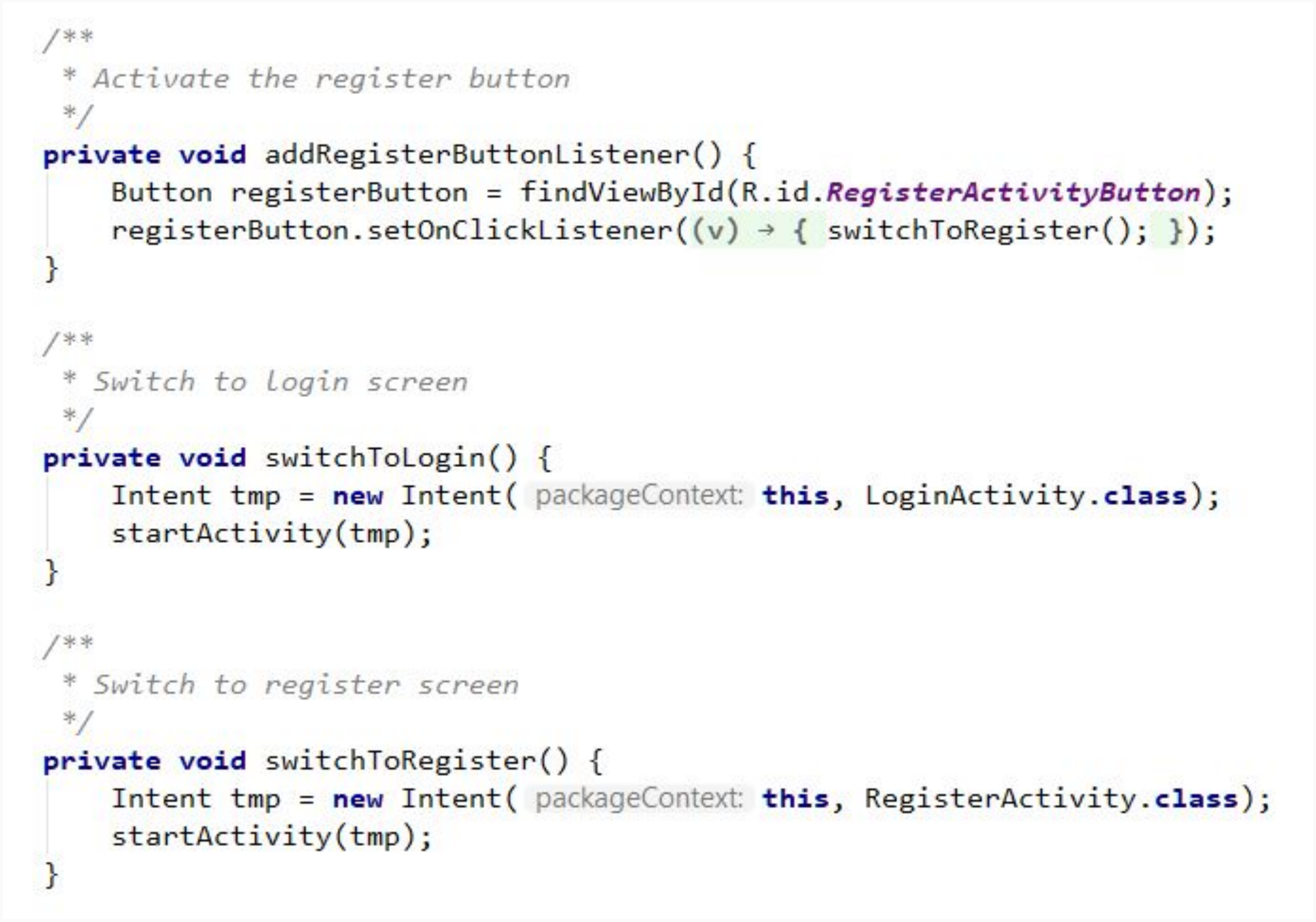
```java
@Test
public void unapplyMove() {
    game.applyMove(move1);
    game.applyMove(move2);
    game.applyMove(move3);
    assertEquals(PRColor.WHITE, game.getBoard().getSquare( x: 2,  y: 4).occupiedBy());
    assertEquals(PRColor.NONE, game.getBoard().getSquare( x: 1,  y: 1).occupiedBy());
    game.unapplyMove();
    game.unapplyMove();
    game.unapplyMove();
    assertEquals(PRColor.WHITE, game.getBoard().getSquare( x: 1,  y: 1).occupiedBy());
    assertEquals(PRColor.NONE, game.getBoard().getSquare( x: 2,  y: 4).occupiedBy());
    assertEquals(PRColor.BLACK, game.getBoard().getSquare( x: 2,  y: 6).occupiedBy());
}

@Test
public void getGameResult() {
    game.applyMove(new PRMove(game.getBoard().getSquare( x: 1,  y: 1),
            game.getBoard().getSquare( x: 1,  y: 7),  isCapture: false,  isEnPassantCapture: false));
    assertEquals(PRColor.WHITE, game.getGameResult());
    game.unapplyMove();
    game.applyMove(new PRMove(game.getBoard().getSquare( x: 1,  y: 6),
            game.getBoard().getSquare( x: 1,  y: 0),  isCapture: false,  isEnPassantCapture: false));
    assertEquals(PRColor.BLACK, game.getGameResult());
    game.unapplyMove();
    assertEquals(PRColor.NONE, game.getGameResult());
}
```

Unit Test Coverage 0% Class - StartingActivity



```
/**
 * The initial activity for the game center.
 */
public class StartingActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_starting_);
        addRegisterButtonListener();
        addLoginButtonListener();
    }


    /**
     * Activate the login button
     */
    private void addLoginButtonListener() {
        Button loginButton = findViewById(R.id.LoginActivityButton);
        loginButton.setOnClickListener((v) → { switchToLogin(); });
    }

    /**
     * Activate the register button
     */
    private void addRegisterButtonListener() {
        Button registerButton = findViewById(R.id.RegisterActivityButton);
        registerButton.setOnClickListener((v) → { switchToRegister(); });
    }
```

Unit Test Coverage 0% Class - StartingActivity



```
/**
 * Activate the register button
 */
private void addRegisterButtonListener() {
    Button registerButton = findViewById(R.id.RegisterActivityButton);
    registerButton.setOnClickListener((v) → { switchToRegister(); });
}

/**
 * Switch to login screen
 */
private void switchToLogin() {
    Intent tmp = new Intent( packageContext: this, LoginActivity.class);
    startActivity(tmp);
}

/**
 * Switch to register screen
 */
private void switchToRegister() {
    Intent tmp = new Intent( packageContext: this, RegisterActivity.class);
    startActivity(tmp);
}
```