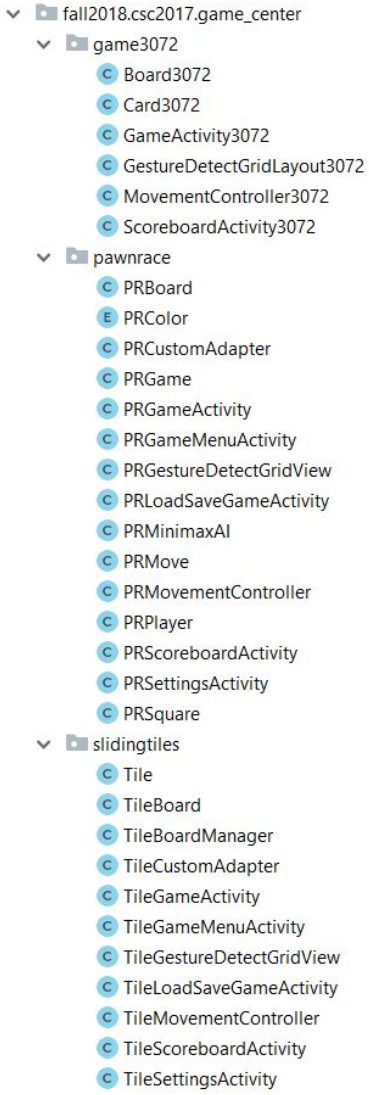
CSC207 Group Project Presentation

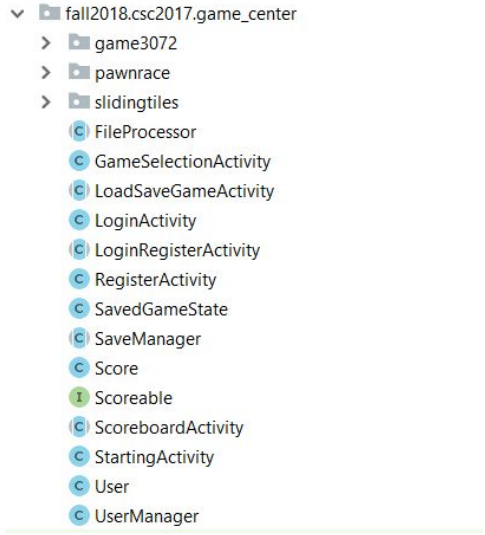
Group 0641

Outline of Presentation

- Code walkthrough
  - Login/register system
  - Scoreboard system
  - Load/save system
  - Game selection
  - Game implementations
    - 3072
    - Pawn Race
    - Sliding Tiles
- Design Patterns
- Testing
  - Code coverage

Overview of Classes

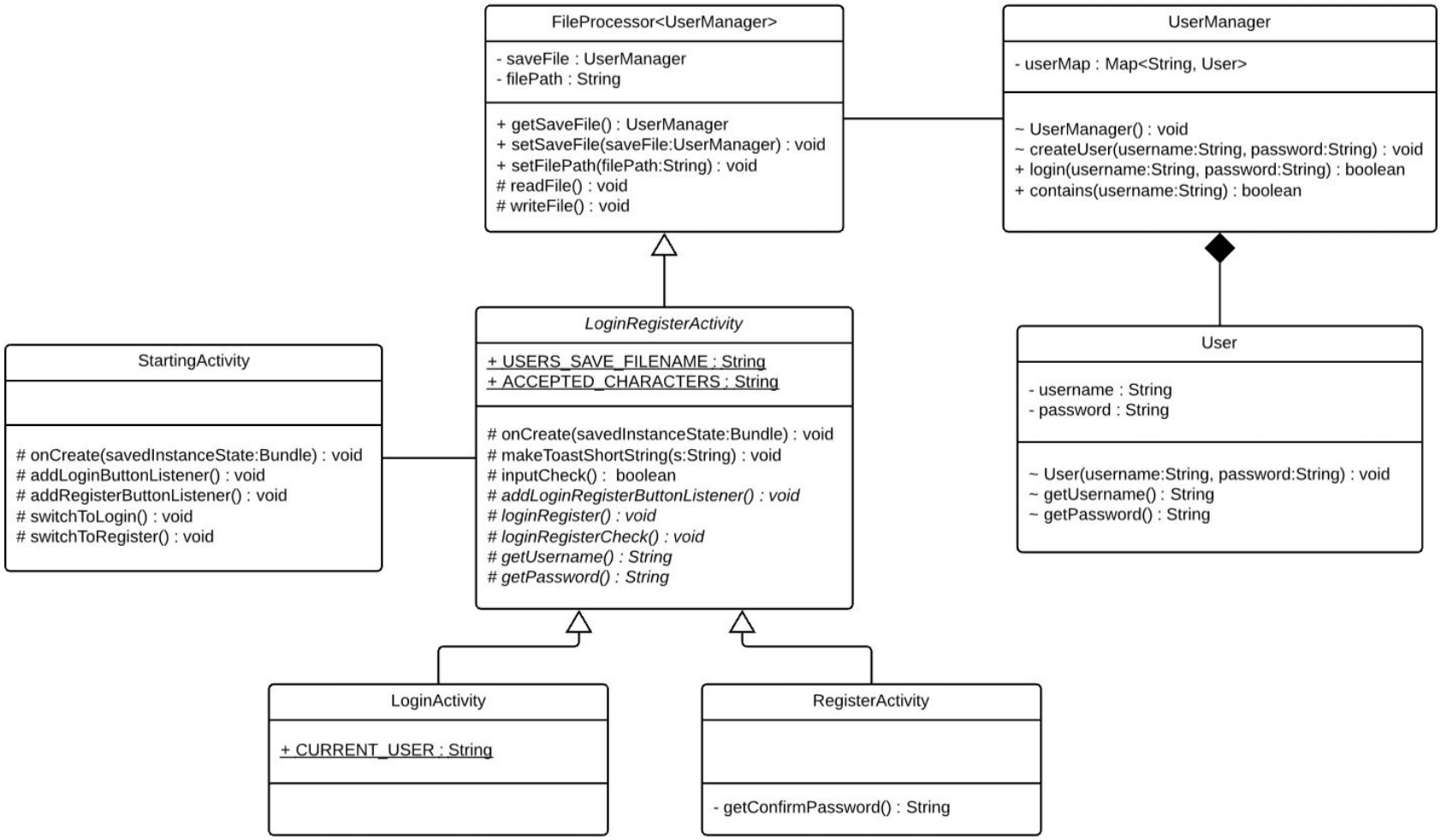




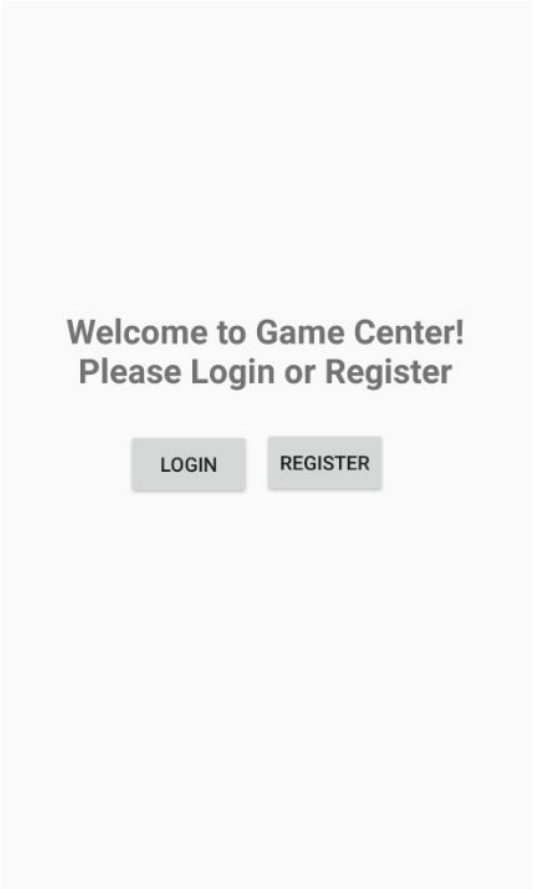
Most Important Classes

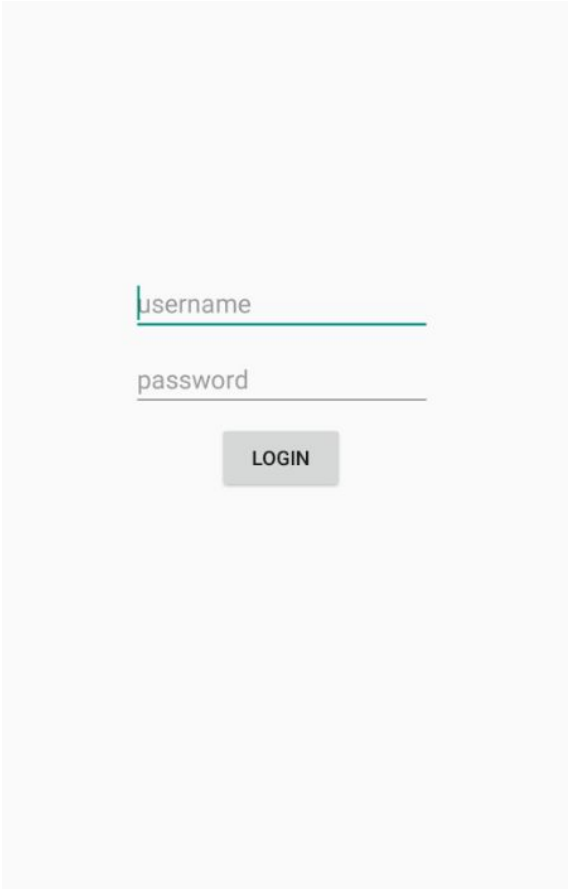
- **FileProcessor** - the single most important class in the whole project; the abstract class that serializes objects: used to store the UserManager, SavedGameState for every user in every game, and List<Score> for every game
    - Expanding on this, SaveManager, ScoreboardActivity, and LoadSaveGameActivity 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

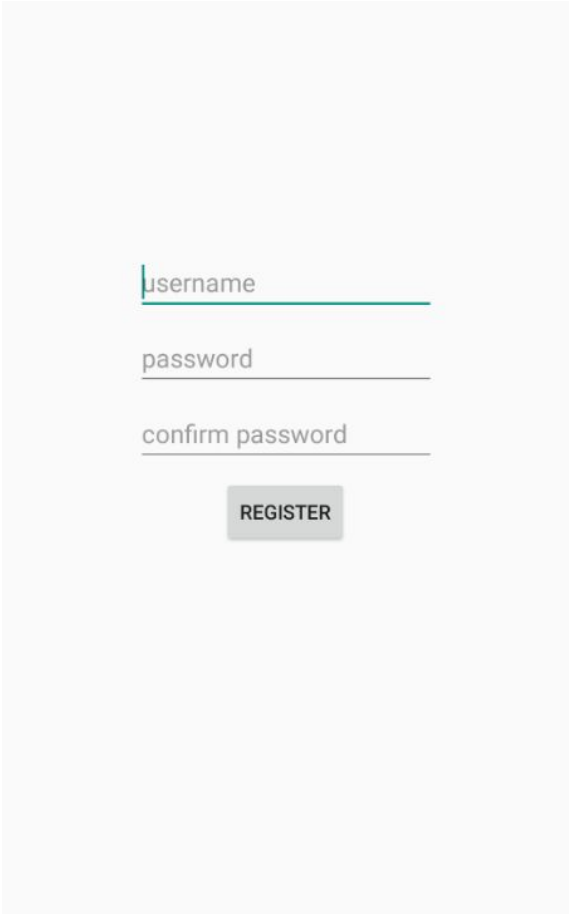
Login/Register System



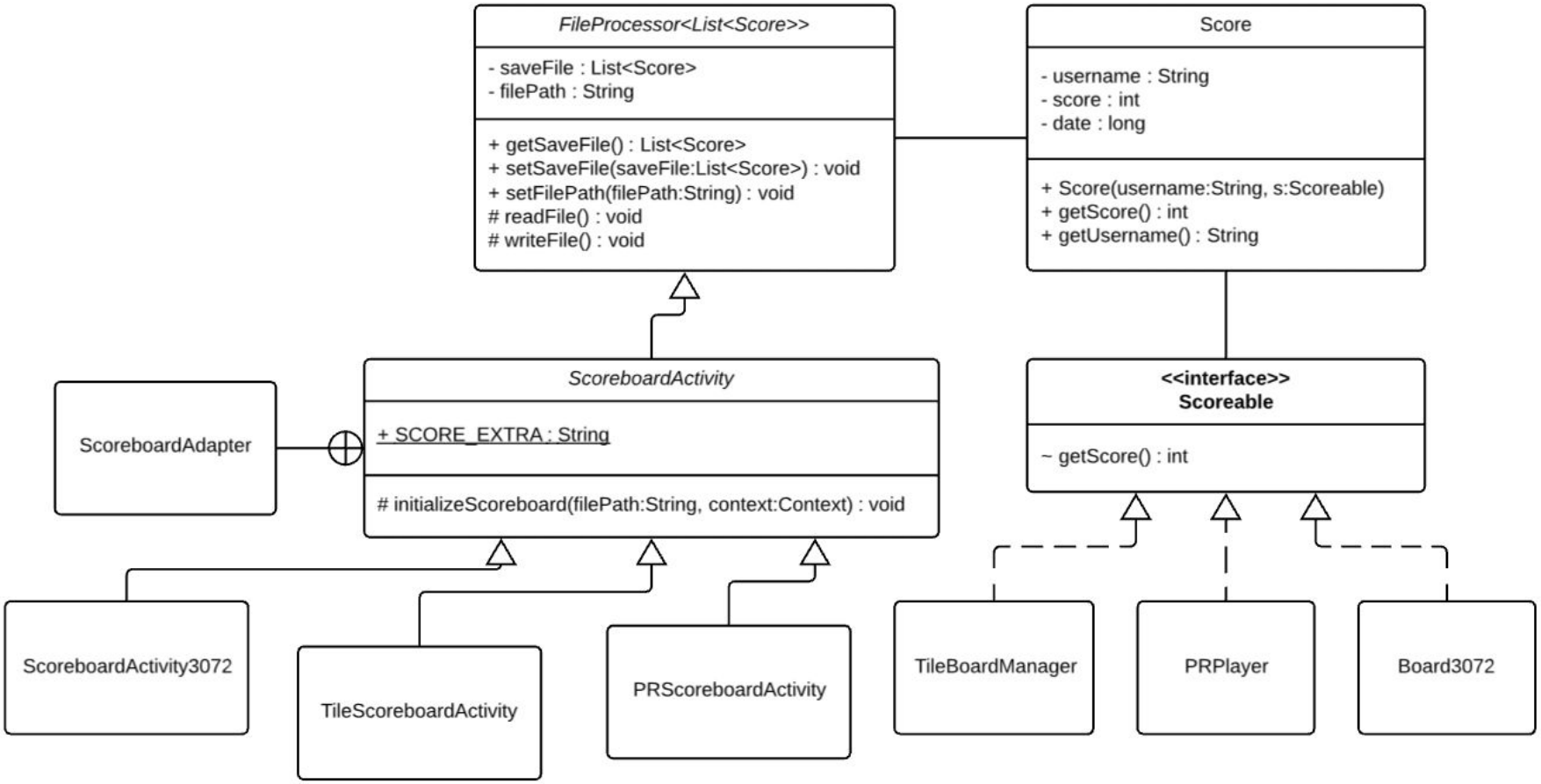
Login/Register System



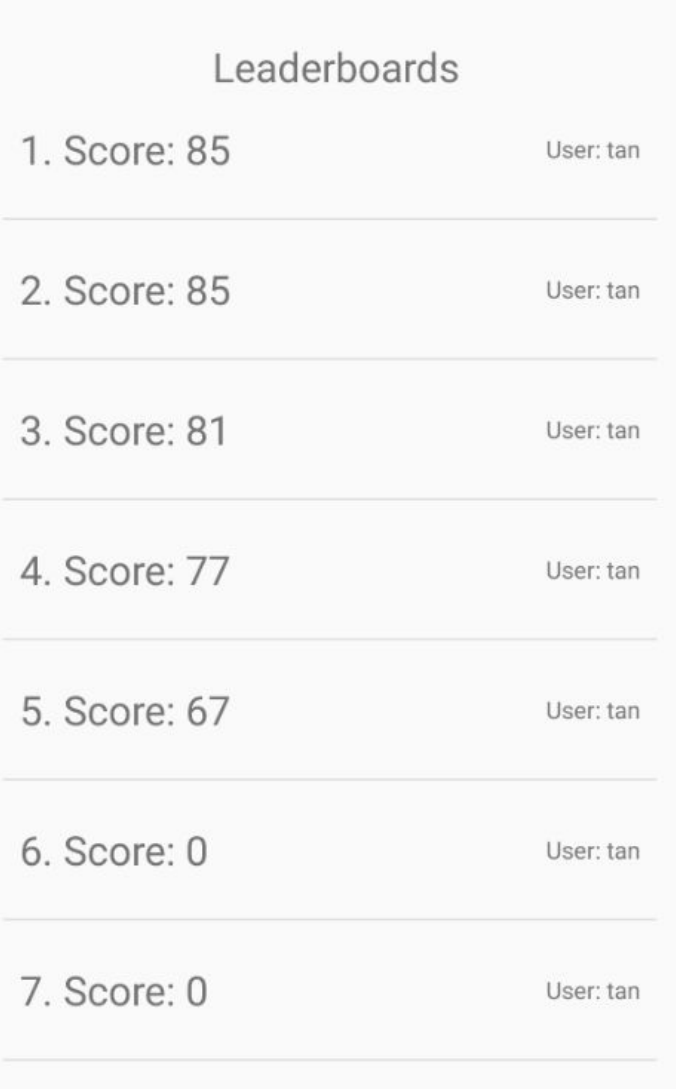




Scoreboard System



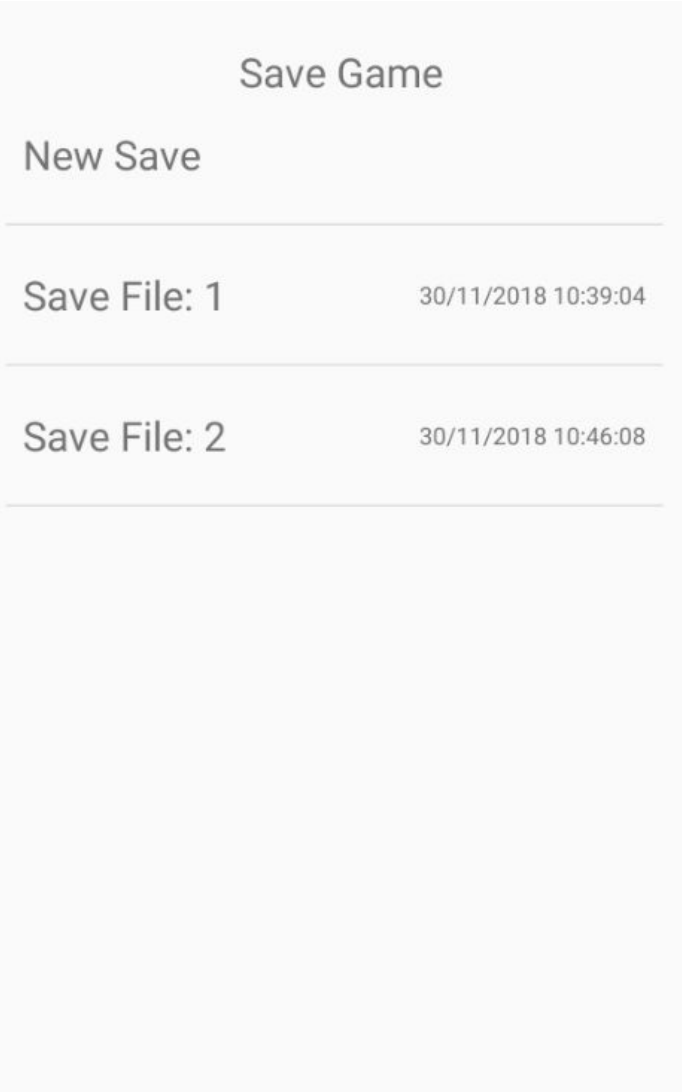
Scoreboard System

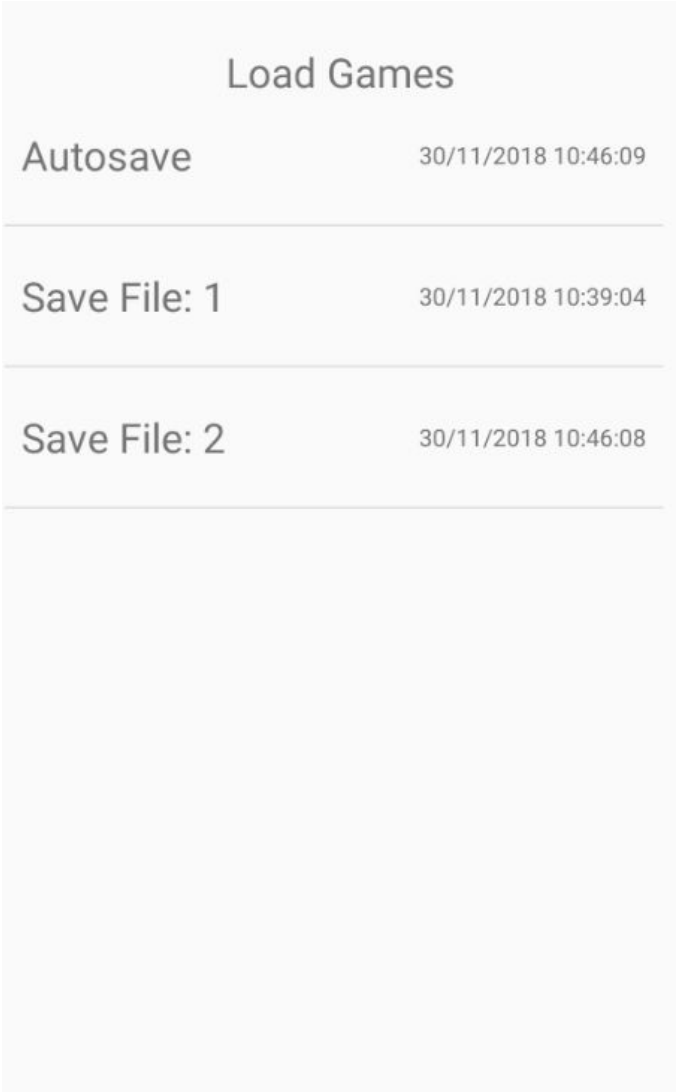


Load/Save Game System

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

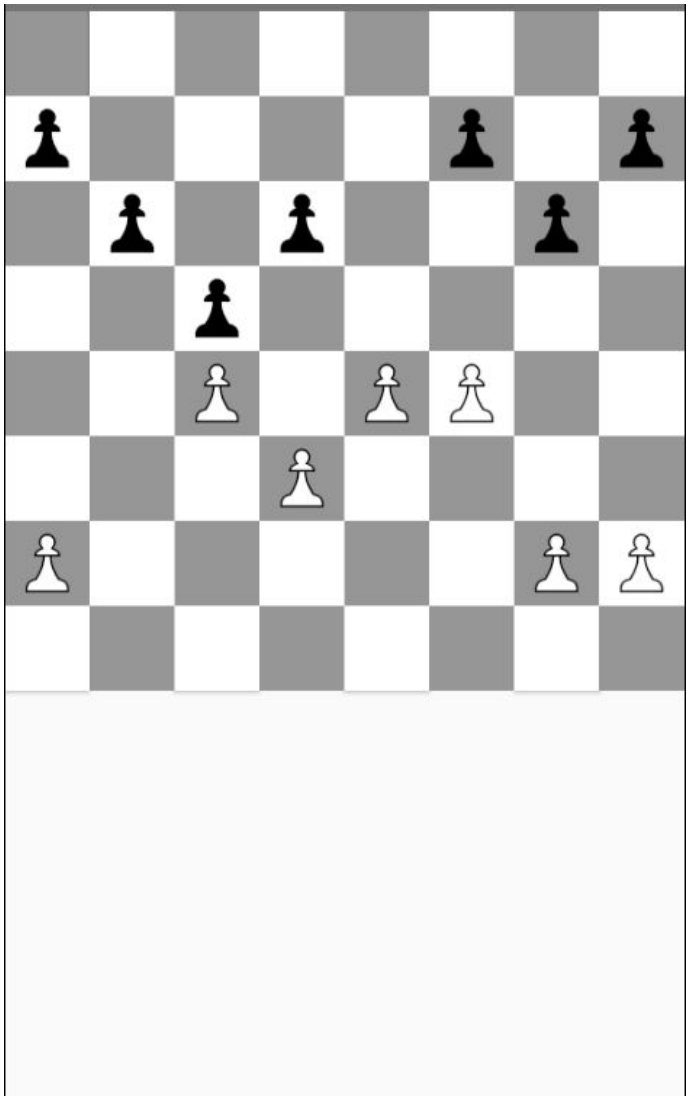




Pawn Race Implementation

- 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

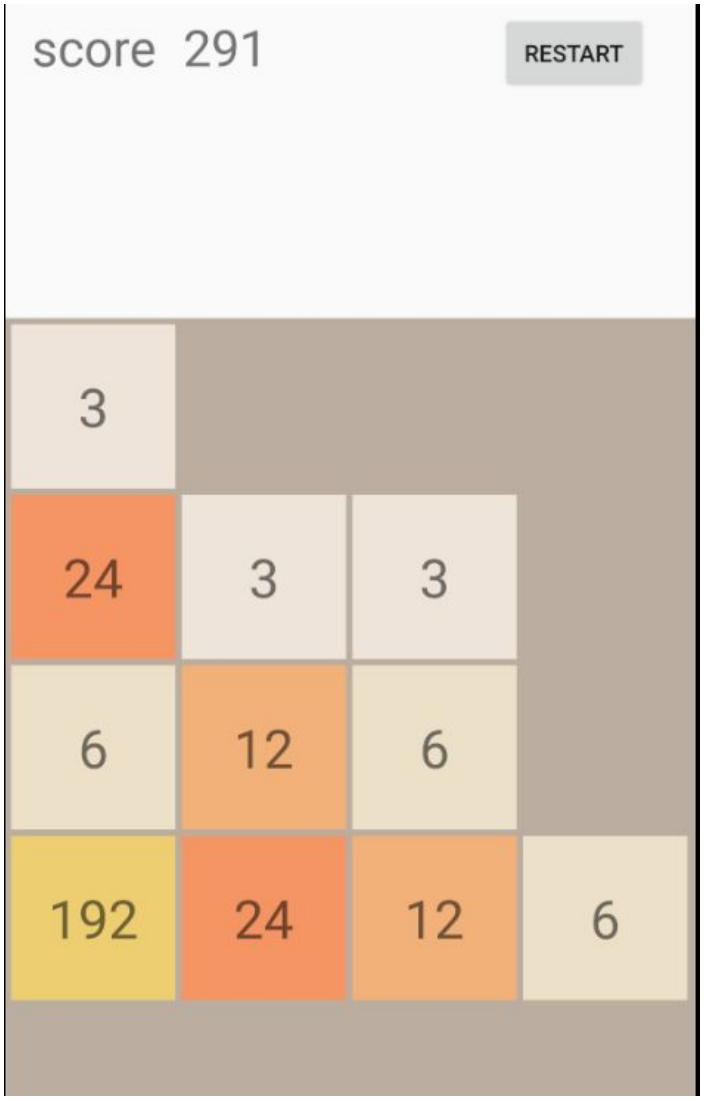
Pawn Race Implementation



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.

3072 Implementation



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

- 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


```
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


```
@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


```
@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

```java
/**
 * 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

```java
/**
 * 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);
}
```