Refactoring

Improvement Analysis

Methods

method	LOC	RETURN	v(G)	IF_NEST	NP
com.sem.pool.game.GameState.handleBallPotting()	38	1	8	1	0

method	LOC	RETURN	v(G)	IF_NEST	NP
com.sem.pool.game.GameState.handleTurnAdvancement()	34	1	7	2	0

- The two methods have 38 and 34 lines of code, which could be reduced to make the methods more comprehensible. Usually, the recommended LOC count is 30, so we should aim for that.
- The methods have a cyclomatic complexity of 8 and 7 respectively, which could certainly be improved to make the methods far less complex and more testable. Right now, this would mean more or less 2^8 (or 2^7 for the second one) possible total paths in the code, which overcomplicates the methods too much. We should aim for a CC of at most 3 or 4 to make the number of possible paths feasible (2^3 or 2^4)
- Furthermore, the current CC made the second method hard to understand and test in practice, so it would definitely benefit from a refactor.

method	LOC	RETURN	v(G)	IF_NEST	NP
com.sem.pool.scene.Scene3D.Scene3D(Environment,Camera,List<Ball3D>,Table	35	1	2	0	7

- The constructor method has a relatively large number of parameters. This is something that we should improve upon, because constructors like these in practice are hard to use due to having to handle a high number of parameters. We should aim to have at most 4-5 parameters, or even less if possible to not overcomplicate the construction of the objects.

method	LOC	RETURN	v(G)	IF_NEST	NP
com.sem.pool.game.Game.advanceGameLoop(float)	23	1	5	3	1

- The method above has a CC of 5, which could be improved. Furthermore, there are 3 nested if statements, which make the code quite hard to comprehend and test in isolation. Hence, we can improve on this quite a bit.

method	RETURN	v(G)	IF_NEST	NP	NCLOC	CONTROL
com.sem.pool.scene.Scene3D.recenterCueBall(CueBall3D)	1	6	1	1	27	8

- The method has a total of 8 control statements, making it complicated in terms of its control flow graph. As such, it should be split up to make it less complex.

Classes

class	CBO	DIT	LCOM	LOC	NOAC	NOIC	NOOC
com.sem.pool.game.GameState	15	1	1	335	29	11	0
com.sem.pool.scene.Ball3D	33	2	2	268	23	11	2

- The two largest classes in our main package of the game are the GameState and Ball3D classes, having more than 200 lines of code. We should aim to refactor them so that they have around 200 lines of code so that they are less convoluted, less complex and more readable.
- Furthermore, both of these classes have quite a lot of new (non-inherited/overriden) methods added, which make these classes quite complex. Reducing the lines of code could lead to reducing this count as well, making the classes more understandable. We will aim for roughly 20 new operations added.

class	CBO	DIT	LCOM	LOC	NOAC	NOIC	NOOC
com.sem.pool.screens.Pool	14	1	10	168	19	11	0

- The main rendering class, Pool, seems to be lacking cohesion quite a bit. This could be improved significantly, and the aim would be to have an LCOM of around 3-5 so that the class is more cohesive and does not handle too many responsibilities at once.

class	CBO	DIT	LCOM	LOC	NOAC	NOIC	NOOC	CSA
com.sem.pool.game.GameState	15	1	1	335	29	11	0	11
com.sem.pool.scene.Scene3D	22	1	3	241	18	11	0	9

- The two classes above have a large number of attributes (11 and 9 respectively), making the classes become quite difficult to analyze and keep track of, and thus not conforming to one responsibility too much. As such, we should aim to improve this to around 5-6 so that it is easier to keep track of each attribute and where they are used.

class	CBO	DIT	LCOM	NOC	RFC	WMC	CSA
com.sem.pool.game.GameState	15	1	1	0	54	54	12

- The class shown has the highest weighted method complexity in the application, making it the most complex classes to understand. Thus, the class should be split up and reorganized to make it less complex.

Refactoring Performed

For the metrics before refactoring, the previous section should be referred to to see the results prior to refactor.

Methods

Handle Ball Potting

```
    for (Ball3D ball : currentPottedBalls) {
        if (!typesAssigned && !(ball instanceof CueBall3D)) {
            allPottedBalls.add(ball); // until types are assigned
            // keep track of balls potted
        }

        if (ball instanceof RegularBall3D) {
            potRegularBall((RegularBall3D) ball);
        } else if (ball instanceof EightBall3D) {
            this.eightBallPotted = true;
        } else if (ball instanceof CueBall3D) {
            this.cueBallPotted = true;
        }

        // Remove the ball from the remaining balls set
        remainingBalls.remove(ball);
    }
```

```
/**
 * Pots a single ball by handling any necessary state
 * changes baseed on the ball type.
 * @param ball  Ball to pot
 */
private void potSingleBall(Ball3D ball) {
    handleUnassignedBallPotting(ball);

    if (ball instanceof RegularBall3D) {
        potRegularBall((RegularBall3D) ball);
    } else if (ball instanceof EightBall3D) {
        this.eightBallPotted = true;
    } else if (ball instanceof CueBall3D) {
        this.cueBallPotted = true;
    }
}

/**
 * Handles keeping tack of potting balls when the ball
 * types are not yet assigned to the players.
 * Does nothing if the ball types are already assigned.
 * @param ball  Ball to pot
 */
private void handleUnassignedBallPotting(Ball3D ball) {
    if (!typesAssigned && !(ball instanceof CueBall3D)) {
        allPottedBalls.add(ball); // until types are assigned
        // keep track of balls potted
    }
}
```

```
for (Ball3D ball : currentPottedBalls) {
    // Handle logic for potting the ball
    potSingleBall(ball);

    // Remove the ball from the remaining balls set
    remainingBalls.remove(ball);
}
```

- The two indicated pieces of code were extracted to two separate methods, as shown in the second & third images

Turn Advancement

```
boolean correctFirstTouch;
if (firstBallTouched instanceof RegularBall3D) {
    RegularBall3D firstTouched = (RegularBall3D) firstBallTouched;
    correctFirstTouch = firstTouched.getType() == activePlayer.getBallType();
} else {
    correctFirstTouch = false;
}

// Check for four criteria:
// - Did the player touch the right type of ball first
// - Did the player not pot the cue ball
// - Did the player pot a ball of the wrong type
// - Did the player pot a ball of the correct type
// Special case: if any ball is potted during the break shot, keep the turn
if (!(turnCount == 0 && !allPottedBalls.isEmpty()) || this.cueBallPotted) {
    if (!correctFirstTouch
            || !getActivePlayer().getPottedCorrectBall()) {
        // Not all criteria were satisfied -> player loses the turn
        loseTurn();
    }
}
```

```
/**
 * Method to handle all logic with regards to gaining an extra turn.
 */
// Warnings are suppressed because the 'DU'-anomaly isn't actually applicable here,
// and it suddenly showed up. Very probable to be a bug in PMD.
/PMD.DataflowAnomalyAnalysis/
public void handleTurnAdvancement() {
    state = State.Idle;
    Player activePlayer = getActivePlayer();

    // Advance turn to the next Player if the current
    // player should lose their turn.
    if (doesPlayerLoseTurn()) {
        loseTurn();
    }

    // Reset temporary variable
    activePlayer.setPottedCorrectBall(false);

    // Increment the turn counter
    turnCount += 1;
}
```

```java
/**
 * Determines whether the active Player should lose their current turn.
 * Check for four criteria:
 * - Did the player touch the right type of ball first
 * - Did the player not pot the cue ball
 * - Did the player pot a ball of the wrong type
 * - Did the player pot a ball of the correct type
 *  Special case: if any ball is potted during the break shot, keep the turn
 * @return  True if the active Player should lose their turn.
 */
private boolean doesPlayerLoseTurn() {
    // Not all criteria were satisfied -> player loses the turn
    return !isPlayerRegularPottingValid() || cueBallPotted;
}

/**
 * Checks whether the Player satisfies all conditions for regular
 * ball potting to gain the next turn.
 * @return  True if the Player has not violated any potting rules
 *          for regular (non-cue) ball potting.
 */
private boolean isPlayerRegularPottingValid() {
    // If break shot, we only care if the Player potted
    // any balls at all
    if (turnCount == 0) {
        return !allPottedBalls.isEmpty();
    } else {
        // If not break shot, we have to verify
        // that the Player touched & potted the
        // correct ball type.
        return isBallContactValid();
    }
}
```

```java
/**
 * Checks whether the Player contacted the cue ball with the valid
 * pool balls to gain the next turn. This includes touching
 * and potting the correct ball first.
 * @return  True if the Player contacted the right pool balls to
 *          gain next turn.
 */
private boolean isBallContactValid() {
    // Check whether the first touched ball is correct
    boolean firstTouchCorrect = false;

    if (firstBallTouched instanceof RegularBall3D) {
        RegularBall3D firstTouched = (RegularBall3D) firstBallTouched;
        firstTouchCorrect = firstTouched.getType() == getActivePlayer().getBallType();
    }

    // Additional check to see whether the Player potted the correct ball
    return firstTouchCorrect && getActivePlayer().getPottedCorrectBall();
}
```

- The method that handles turn advancement has a part which handles the Player losing their turn on various conditions
- In order to reduce the complexity of the method, we can extract the loss of turn logic into its own method, and then further extract the if condition checking into their own methods, as shown in the subsequent images.

Cue Ball Placement

```
    for (Ball3D other : this.getPoolBalls()) {
        if (ball.equals(other)) {
            continue;
        }

        CollisionHandler handler = ball.getCollisionHandler();
        if (handler.checkHitBoxCollision(ball.getHitBox(), other.getHitBox())) {
            doesCollide = true;
            break;
        }
    }
```

```
/**
 * Checks whether there exists a ball that collides with the
 * indacated ball.
 * @param ball  Ball to check for collision
 * @return  True if there is a ball that collides with the specified ball.
 */
private boolean existsCollidingBall(Ball3D ball) {
    for (Ball3D other : this.getPoolBalls()) {
        if (ball.equals(other)) {
            continue;
        }

        CollisionHandler handler = ball.getCollisionHandler();
        if (handler.checkHitBoxCollision(ball.getHitBox(), other.getHitBox())) {
            return true;
        }
    }

    return false;
}
```

- One of the reasons why the method has a complicated control flow graph is because it handles functionality that can be abstracted away into another method. Hence, we identified the part in the first method which we then **extracted** to another method.
- Then, we call this method instead of determining it via the loop like indicated above.

Game Loop Advancing

```
public void advanceGameLoop(float deltaTime) {
    if (state.isStarted()) {
        // Check if Game has a winning Player
        if (state.getWinningPlayer().isPresent()) {
            endGame(this.state.getWinningPlayer().get(), this.state.getPlayers());
        } else {
            // Check if any ball is in motion
            determineIsInMotion();

            if (state.isInMotion()) {
                moveBalls(deltaTime);
            } else if (state.isIdle()) {
                respondToInput();
            }
        }
    } // Do nothing if game is not started
}
```

```
/**
 * Performs an action in the game loop based on the
 * internal state of the game.
 * This could be moving the balls or responding
 * to user input.
 * @param deltaTime  deltaTime, time between current and last frame.
 */
private void performGameLoopAction(float deltaTime) {
    if (state.isInMotion()) {
        moveBalls(deltaTime);
    } else if (state.isIdle()) {
        respondToInput();
    }
}
```

- In the advancement of the Game loop, we have quite a bit of if statement nesting, which increases CC and the complexity of the method. Hence, we decided to move the 3rd nested if statement away to its own method instead.
- We then call the method that we moved instead.

Scene3D Construction

```
/**
 * Creates an instance of a 3D Pool Game scene from the specified
 * parameters of the scene.
 *
 * @param environment Environment settings of the scene (e.g. light)
 * @param camera      Camera used in the scene
 * @param poolBalls   List of pool balls part of the scene
 * @param table       The table to use for the scene
 * @param cue         The cue to use for the scene
 * @param batch       Model Batch to use for rendering
 */
public Scene3D(Environment environment, Camera camera, List<Ball3D> poolBalls,
               Table3D table, Cue3D cue, ModelBatch batch, SoundPlayer soundPlayer) {
    this.environment = environment;
```

```
/**
 * Creates an instance of a 3D Pool Game scene from the specified
 * parameters of the scene.
 *
 * @param batch         Model Batch to use for rendering
 * @param gameElements  Game element collection (Balls, Table and Cue)
 * @param sceneElements Scene element collection (Camera, Environment, Sound Player)
 */
public Scene3D(ModelBatch batch, GameElements gameElements, SceneElements sceneElements) {
    this.gameElements = gameElements;
    this.sceneElements = sceneElements;
    this.modelBatch = batch;
```

```
/**
 * Collection of 3D Scene elements that are part
 * of a Game of Pool.
 * This class groups the objects together in one cohesive unit.
 */
public final class GameElements {
    private final List<Ball3D> poolBalls;
    private final Table3D table;
    private final Cue3D cue;

    /**
     * Creates a new group of Game elements.
     * @param poolBalls  List of pool balls
     * @param table      3D Table object
     * @param cue        3D Cue object
     */
    public GameElements(List<Ball3D> poolBalls, Table3D table, Cue3D cue) {
        this.poolBalls = poolBalls;
        this.table = table;
        this.cue = cue;
    }
```

```java
/**
 * Collection of Scene elements that handle functionalities
 * and the environment of the Scene, such as the Camera or
 * lighting.
 * This class groups such objects together in one cohesive unit.
 */
public final class SceneElements {
    private final Environment environment;
    private final Camera camera;
    private final SoundPlayer soundPlayer;

    /**
     * Creates a new group of Scene elements.
     * @param environment    Environment of the Scene
     * @param camera         Camera of the Scene
     * @param soundPlayer    Sound Player to use for the Scene
     */
    public SceneElements(Environment environment, Camera camera, SoundPlayer soundPlayer) {
        this.environment = environment;
        this.camera = camera;
        this.soundPlayer = soundPlayer;
    }
```

- The constructor of the Scene currently took in far too many parameters, most of which corresponded to the Scene's 3D elements and Scene's elements that manage the environment
- As such, these elements were grouped together into two separate immutable final class to make them more cohesive (as in images 3 and 4)
- The Scene's public API was left unchanged. Most of the getters were still left in there to prevent issues in changes to the code. The goal was to mainly reduce the complexity of construction rather than refactor the class as a whole in this case.

Classes

Scene3D Attributes

- As outlined previously in Scene3D Construction (for method refactoring), the refactoring operation that we performed not only improved method metrics, but also class metrics, because we grouped some attributes together into cohesive units instead. Thus, the refactor acted not only as a method refactoring for the constructor, but also as a class refactor.

Pool Lack of Cohesion

```
public class Pool implements Screen, GameObserver {

    @Override
    public void hide() {
    }

    @Override
    public void pause() {
    }

    @Override
    public void resume() {
    }

    @Override
    public void onGameStarted() {

    }

    @Override
    public void onBallPotted(Ball3D ball) {

    }

    @Override
    public void onMotion() {

    }

    @Override
    public void onMotionStop(Ball3D lastTouched) {

    }
```

```
public abstract class GameScreen implements Screen, GameObserver {
    @Override
    public void resize(int width, int height) {
        System.out.println("Resizing: " + width + ", " + height);
    }

    @Override
    public void hide() {
        System.out.println("Hiding game");
    }

    @Override
    public void onGameStarted() {
        System.out.println("Game started!");
    }

    @Override
    public void onBallPotted(Ball3D ball) {
        System.out.println(ball.getId() + " Potted!");
    }

    @Override
    public void onMotion() {
        System.out.println("Game is now in motion");
    }

    @Override
    public void onMotionStop(Ball3D lastTouched) {
        System.out.println("Game motion stopped");
    }

    @Override
    public void onGameEnded(Player winner, List<Player> players) {
        System.out.println(winner.getId() + " won!");
    }
}
```

- In our Pool class, which handles rendering and presenting the Game to the Player, we had low cohesion due to a lot of implemented methods not actually being used at all.
- Since this could occur more frequently, and we might want a stronger base for our game class (to make future extensions easier), we decided to group the two interface implementations under a new abstract class **GameScreen**
- This class implements both interfaces, and provides some default implementations which can serve to provide some debug information for functionality that is not directly visible.
- This no longer forces our Pool class to implement methods that do not need to be implemented at all.

Ball3D Length & Complexity

```
public boolean checkCollision(Ball3D other) {
    // balls placed below the table (when potted) should not collide.
    if (this.getCoordinates().y < 0 || other.getCoordinates().y < 0) {
        return false;
    }
    if (getCollisionHandler().checkHitBoxCollision(getHitBox(), other.getHitBox())) {
        // Create vector from ball to other
        Vector3 directionToOther = new Vector3(other.getCoordinates())
                .sub(new Vector3(getCoordinates()));

        // Calculate phi, the angle of the direction of collision.
        double phi = acos(directionToOther.nor().x);

        // Calculate theta1 and theta2, the angle of the respective ball's direction.
        double theta1 = acos(this.getDirection().x);

        double theta2 = acos(other.getDirection().x);
        theta2 = checkAngle(other.getDirection().x, theta2);
        phi = checkAngle(directionToOther.z, phi);
        theta1 = checkAngle(this.getDirection().z, theta1);

        // Declaration of the speed of both balls before the collision for calculation purposes
        double v1 = this.getSpeed();
        double v2 = other.getSpeed();

        // Calculate and set the speed and direction of ball 1
        // We do this using two methods now,
        // but keep the old code here in case something went wrong.
        double v1x = calculateVx(v1, v2, theta1, theta2, phi);
        //v2 * cos(theta2 - phi) * cos(phi) + v1 * sin(theta1 - phi)
        //* cos(phi + (PI / 2));
        double v1z = calculateVz(v1, v2, theta1, theta2, phi);
        // v2* cos(theta2 - phi) * sin(phi) + v1 * sin(theta1 - phi)
        //* sin(phi + (PI / 2));

        this.setSpeed(calculateSpeed(v1x, v1z));
        Vector3 newDirection = new Vector3(getDirection()).sub(new Vector3(directionToOther));
        setDirection(newDirection);
```

```java
public boolean checkCollision(Ball3D other) {
    // balls placed below the table (when potted) should not collide.
    if (this.getCoordinates().y < 0 || other.getCoordinates().y < 0) {
        return false;
    }
    if (getCollisionHandler().checkHitBoxCollision(getHitBox(), other.getHitBox())) {
        // Create vector from ball to other
        Vector3 directionToOther = new Vector3(other.getCoordinates())
                .sub(new Vector3(getCoordinates()));

        // Calculate new speed for collision;
        // Result returned as float of 2 elements, where the first
        // is the new speed of this ball, and the second is the speed of the other ball.
        float[] newSpeed = PhysicsUtils.getSpeedOnCollision(this.direction,
                this.getSpeed(), directionToOther.nor(), other.direction, other.getSpeed());

        // Update speed & direction for this ball
        this.setSpeed(newSpeed[0]);
        Vector3 newDirection = new Vector3(getDirection()).sub(new Vector3(directionToOther));
        setDirection(newDirection);

        // Update speed & direction for other ball
        other.setSpeed(newSpeed[1]);
        other.setDirection(directionToOther);
        return true;
    }
    return false;
}
```

- As seen in the first image, right now a lot of physics calculations are done in the checkCollision method for the Ball.
- Since that is not the Ball's responsibility, and that functionality can be independent of the ball, we decided to create a new class that couples the physics calculations together. We will call this class PhysicsUtils
- Moreover, one of the previous classes, CollisionHandler, had a static method which reflects a vector. We thought it would be more suitable to put this under the class that we will create now.
- After refactoring (last image), the class becomes far less complicated, because the 4 methods that were only in use by the checkCollision method have been moved to PhysicsUtils.
- Also, our physics calculations are now more abstract and easier to modify (with regards to speed).

Game Ball State introduction

```
private transient Set<Ball3D> remainingBalls;
private transient List<Ball3D> currentPottedBalls; // Balls potted in current turn
private transient List<Ball3D> allPottedBalls; // All Balls potted in any turn.
private transient boolean cueBallPotted;
private transient boolean eightBallPotted;

// First ball touched in current turn
private transient Ball3D firstBallTouched;
```

```
/**
 * Class that holds the Pool Ball state during a Pool game.
 * State includes the remaining balls, the potted balls during
 * an iteration, flags whether the cue & eight ball were potted
 * and the first touched ball.
 */
public class GameBallState {
    private transient Set<Ball3D> remainingBalls; // Balls remaining to be potted
    private transient List<Ball3D> currentPottedBalls; // Balls potted in current turn
    private transient List<Ball3D> allPottedBalls; // All Balls potted in any turn.

    // Cue & eight ball potted flags
    private transient boolean cueBallPotted;
    private transient boolean eightBallPotted;

    // First ball touched in current turn
    private transient Ball3D firstBallTouched;
```

- The GameState class as of now held a lot of attributes for ball state (which balls were potted, which balls remain, etc.).
- As such, a decision was made to add another class, GameBallState, which keeps the internal state of the ball and provides an API to interact and update the state.
- This also benefited the code since if we ever want to replace the way the ball state is tracked, it is now easier to do so because we are relying on API calls rather than directly accessing the data structures in the Game State to maintain state.

Game Turn Handler

```java
/**
 * Class responsible for handling the turns in a Pool game.
 * Keeps track of the Players and the current turn, as well
 * as the overall turn count.
 */
public class TurnHandler {
    private transient List<Player> players;
    private transient int playerTurn;
    private transient int turnCount;

    /**
     * Creates a new Turn Handler with the specified players.
     * @param players  Players of the Game
     */
    public TurnHandler(List<Player> players) {
        this.players = players;
        this.turnCount = 0;
    }

    public List<Player> getPlayers() {
        return players;
    }

    public int getPlayerTurn() {
        return playerTurn;
    }

    /**
     * Gets the active player.
     * @return active player
     */
    public Player getActivePlayer() {
        return players.get(playerTurn);
    }
```

```
private transient State state;
private transient GameBallState gameBallState;
private transient TurnHandler turnHandler;

/**
 * Creates a new game state with the specified Players and
 * the specified Pool balls.
 * @param players     List of Players for the game
 * @param poolBalls   List of pool balls to use for the game
 */
public GameState(List<Player> players, List<Ball3D> poolBalls) {
    this.state = State.Stopped;
    this.typesAssigned = false;
    this.gameBallState = new GameBallState(poolBalls);
    this.turnHandler = new TurnHandler(players);
}
```

- Another distinguishable responsibility that was noted in the Game State class was turn handling, such as keeping track of turn count, advancing turn, getting the active player, and so on.
- In order to make the Game Turn handler more flexible and more abstract to the Game State, we decided to move the turn functionality to a new class called TurnHandler.
- In that class, we put all the functionality related to turn handling, thus resulting in the Game State class becoming more simplified and using an API instead.

Ball Potting Handler

- Another observation made was that Game State held all of the functionality for ball potting logic, including turn advancement and so on. In other words, it had very low cohesion in that regard.
- As such, to make Game State even more modular, we split off the functionality for ball potting into its own class named BallPottingHandler. Then, we made sure to link it with the Game State and replace method calls to the Ball Potting Handler.
- With the change in place, we can be more flexible with our ball potting handling approaches in the future. If we ever want to swap out the functionality, it would now be far easier, since API calls are used instead for potting rather than all of the functionality being cluttered in the Game State class.

Screen Base Class

```
public abstract class UiScreen implements Screen {
    protected transient MainGame game;
    protected transient Stage stage;
    protected transient Skin skin;
    protected transient TextureAtlas atlas;
    protected transient Table table;

    /**
     * Create the new screen.
     * @param game Game state to pull from.
     */
    public UiScreen(MainGame game) { this.game = game; }

    /**
     * Show the screen.
     * This actually renders the screen.
     */
    @Override
    public void show() {
        // Set up the screen.
        this.stage = new Stage(new FitViewport(Gdx.graphics.getWidth(), Gdx.graphics.getHeight()));
        Gdx.input.setInputProcessor(this.stage);
        this.atlas = new TextureAtlas( internalPackFile: "uiskin.atlas");
        this.skin = new Skin(Gdx.files.internal( path: "config/skin/uiskin.json"), this.atlas);

        // Render the elements.
        this.table = new Table();
        this.table.setFillParent(true);
        this.table.setPosition( x: 0,  y: 0);
        this.table.defaults().spaceBottom(10);
        this.table.row().fill().expandX().row();
    }
```

- With the introduction of the Leaderboard feature, we now had more screens (selection of opponent, leaderboard, logging in) which all started having some shared functionality.
- To address this, we introduced a base UiScreen class, which contained some shared functionality relevant for all the screens
- The previously defined GameScreen class was now made to extend the new UiScreen class
- This refactoring allowed us to reach consistency with regards to screens and reduce code redundancy significantly

Improved Metrics

For previous metrics, see the write-up at the beginning of the report.
Some methods are now in different classes due to class refactoring, so metrics for those methods are shown instead (since the original methods were moved there)
With the performed refactoring, we have significantly reduced the complexity of methods & classes primarily in terms of LOC (Lines of Code), cyclomatic complexity (v(G)), lack of cohesion of methods (LCOM) and new operations added (NOAC) among others.

Methods

method	CONTROL	IF_NEST	LOC	NP	RETURN	v(G)
com.sem.pool.game.BallPottingHandler.handleBallPotting(TurnHandler)	2	1	30	1	2	3
com.sem.pool.game.TurnHandler.advanceTurn(GameBallState)	1	1	17	1	1	2
com.sem.pool.scene.Scene3D.Scene3D(ModelBatch,GameElements,Scen	1	0	22	3	1	2
com.sem.pool.game.Game.advanceGameLoop(float)	2	1	22	1	2	3
com.sem.pool.game.Game.performGameLoopAction(float)	2	1	14	1	1	3
com.sem.pool.scene.Scene3D.recenterCueBall(CueBall3D)	3	1	27	1	1	3

Classes

class	LCOM	WMC	CSA	NOAC	LOC
com.sem.pool.game.GameState	2	20	6	17	132
com.sem.pool.game.TurnHandler	1	13	3	10	104
com.sem.pool.game.BallPottingHandler	1	24	2	10	156
com.sem.pool.scene.Ball3D	2	29	8	19	195
com.sem.pool.screens.Pool	2	17	12	6	149
com.sem.pool.screens.GameScreen	1	8	5	5	40
com.sem.pool.screens.UiScreen	4	8	5	7	85
com.sem.pool.scene.Scene3D	1	33	5	18	229