Test-driven Development

Chapter 1

TDD is a simple procedure of writing tests **before the actual implementation**. It's an inversion of a traditional approach where testing is performed after the code is written.

Red-green-refactor: Test-driven development is a process that relies on the repetition of a very short development cycle. It is based on the test-first concept of extreme programming that encourages simple design with a high level of confidence. While writing tests we're in the red state, when the implementation of a test is finished, all tests should pass and then we'll be in the green state.

> When a test fails, the programmer should spend little time trying to fix it. So, if the change can't be done in a matter of minutes it's better to revert the changes and start over.

After all tests are passing, it's time to refactor the code, if the refactor breaks an existing functionality the tests will show and the changes must be reverted.

Speed is the key

The first implementation should not be focused on perfection, the idea is to keep switching between tests and implementation as fast as a game of ping-pong.

> The time between changing from tests to implementation should be measured in minutes or seconds.

TDD is a solution to approach the design in a different way, in such the programmer is forced to think about the implementation and about what the code needs to do before implementation. The main objective of TDD is testable code design with tests as a very useful side product.

> Tests that are defined by an already existing application are biased. They have a tendency to confirm what code does, and not to test whether client's expectations are met, or that the code is behaving as expected.

Mocking

In order for tests to run fast and provide constant feedback, code needs to be organized in such a way that the methods, functions, and classes **can be easily replaced with mocks and stubs**. With or without mocks, the code should be written in a way that we can easily replace one dependency with another.

Chapter 3 - Red-Green State

The following code extracts use this structure:

TicTacToe.java

```java
enum Piece {
    X, O
}

enum Result {
    DRAW, X_WIN, O_WIN
}

public class TicTacToe {

    private final int SIZE = 3;
    private final Piece[][] board;
    private int filledSpaces = 0;

    private Piece lastPlayed;


    public TicTacToe() {
        this.lastPlayed = null;
        this.board = new Piece[SIZE][SIZE];
        for (int i = 0; i < SIZE; i++)
            for (int j = 0; j < SIZE; j++)
                this.board[i][j] = null;

    }

}
```

Requirement 1

A piece can be placed on any empty space of a 3×3 board.

REQ1 - Red State

Tests for this requirement:

TicTacToeSpec.java

```java
    @Rule
    public ExpectedException exception = ExpectedException.none();
    private TicTacToe board;

    @Before
    public final void before() {
        board = new TicTacToe();
    }

    @Test
    public void whenPiecePlacedOutsideXAxisThrowRuntimeException() {
        exception.expect(RuntimeException.class);
        board.play(4, 2);
    }

    @Test
    public void whenPiecePlacedInNegativeXExpectRuntimeError() {
        exception.expect(RuntimeException.class);
        board.play(-1, 2);
    }

    @Test
    public void whenPiecePlacedInNegativeYExpectRuntimeError() {
        exception.expect(RuntimeException.class);
        board.play(1, -2);
    }

    @Test
    public void whenPiecePlacedOutsideYaxisThrowRuntimeException() {
        exception.expect(RuntimeException.class);
        board.play(2, 5);
    }

    @Test
    public void whenPiecePlacedInOccupiedFieldExpectRuntimeException() {
        board.play(1, 1);
        exception.expect(RuntimeException.class);
        board.play(1, 1);
    }
```

REQ1 - Green State

Implementation passing all tests and after refactoring.

TicTacToe.java

```java
    public void play(int x, int y) {
        checkCoordinate(x);
        checkCoordinate(y);

        checkField(x, y);

        Piece piece = getNextPlayer();
        this.board[x][y] = piece;
        this.lastPlayed = piece;
        this.filledSpaces++;
    }


    private void checkCoordinate(int value) {
        if (value < 0 || value >= SIZE) throw new RuntimeException("Value out of boundaries.");
    }

    private void checkField(int x, int y) {
        if (this.board[x][y] != null)
            throw new RuntimeException(String.format("Piece placed in occupied field (%d, %d).", x, y));
    }
```

Requirement 2

There should be a way to find out which player should play next.

REQ2 - Red State

Tests for this requirement:

TicTacToeSpec.java

```java
    @Test
    public void getNextPlayerAtStartShouldReturnX() {
        assertEquals(Piece.X, board.getNextPlayer());
    }

    @Test
    public void getNextPlayerAfterXShouldReturnO() {
        board.play(0, 0);
        assertEquals(Piece.O, board.getNextPlayer());
    }

    @Test
    public void getNextPlayerAfterOShouldReturnX() {
        board.play(0, 0);
        board.play(0, 1);
        assertEquals(Piece.X, board.getNextPlayer());
    }
```

REQ2 - Green State

Implementation passing all tests and after refactoring.

TicTacToe.java

```java
    public Piece getNextPlayer() {
        if (this.lastPlayed == null) return Piece.X;
        switch (this.lastPlayed) {
            case X:
                return Piece.O;
            case O:
                return Piece.X;
            default:
                return null;
            }
        }
```

Requirement 3

A player wins by being the first to connect a line of friendly pieces from one side or corner of the board to the other.

REQ3 - Red State

Tests for this requirement:

TicTacToeSpec.java

```java
        @Test
        public void assertWinnerForHorizontalLine() {
            board.play(0, 0); // X
            board.play(1, 2); // O
            board.play(0, 1); // X
            board.play(1, 1); // O
            board.play(0, 2); // X
            board.play(2, 1); // O
            assertEquals(Result.X_WIN, board.getWinner());
        }

        @Test
        public void assertWinnerForVerticalLine() {
            board.play(1, 0);
            board.play(0, 0);
            board.play(1, 2);
            board.play(0, 1);
            board.play(1, 1);
            board.play(0, 2);
            assertEquals(Result.O_WIN, board.getWinner());
```

```java
    }

    @Test
    public void assertWinnerForPrincipalDiagonal() {
        board.play(0, 0);
        board.play(0, 1);
        board.play(1, 1);
        board.play(0, 2);
        board.play(2, 2);
        board.play(1, 0);
        assertEquals(Result.X_WIN, board.getWinner());
    }

    @Test
    public void assertWinnerForSecondaryDiagonal() {
        board.play(2, 0); // X
        board.play(0, 0);
        board.play(1, 1); // X
        board.play(0, 1);
        board.play(0, 2); // X
        board.play(1, 0);
        assertEquals(Result.X_WIN, board.getWinner());
    }
```

REQ3 - Green State

Implementation passing all tests and after refactoring.

TicTacToe.java

```java
    public Result getWinner() {

        Piece winner;

        for (int row = 0; row < SIZE; row++)
```

```java
            if (this.board[row][0] == this.board[row][1] && this.board[row][1] == this.board[row][2]) {
                winner = this.board[row][0];
                if (winner != null) {
                    return mapPieceToResult(winner);
                }
            }

        for (int col = 0; col < SIZE; col++)
            if (this.board[0][col] == this.board[1][col] && this.board[1][col] == this.board[2][col]) {
                winner = this.board[0][col];
                if (winner != null) {
                    return mapPieceToResult(winner);
                }
            }

        if (this.board[0][0] == this.board[1][1] && this.board[1][1] == this.board[2][2]) {
            winner = this.board[0][0];
            if (winner != null) {
                return mapPieceToResult(winner);
            }
        }

        if (this.board[2][0] == this.board[1][1] && this.board[1][1] == this.board[0][2]) {
            winner = this.board[2][0];
            if (winner != null) {
                return mapPieceToResult(winner);
            }
        }

        if (this.filledSpaces != SIZE * SIZE)
            return null;

        return Result.DRAW;
    }

    private Result mapPieceToResult(Piece p) {
        if (p == null) return Result.DRAW;
```

```java
            return p == Piece.X ? Result.X_WIN : Result.O_WIN;
        }
```

Requirement 4

The result is a draw when all the boxes are filled.

REQ4 - Red State

Tests for this requirement:

TicTacToeSpec.java

```java
    @Test
    public void assertDrawBoardFilled() {
        board.play(0, 0);
        board.play(1, 1);
        board.play(2, 2);
        board.play(2, 1);
        board.play(0, 1);
        board.play(0, 2);
        board.play(2, 0);
        board.play(1, 0);
        board.play(1, 2);


        assertEquals(Result.DRAW, board.getWinner());
    }

    @Test
    public void assertWithoutResultBeforeFillingBoard() {
        board.play(0, 0);
        assertNull(board.getWinner());
    }
```

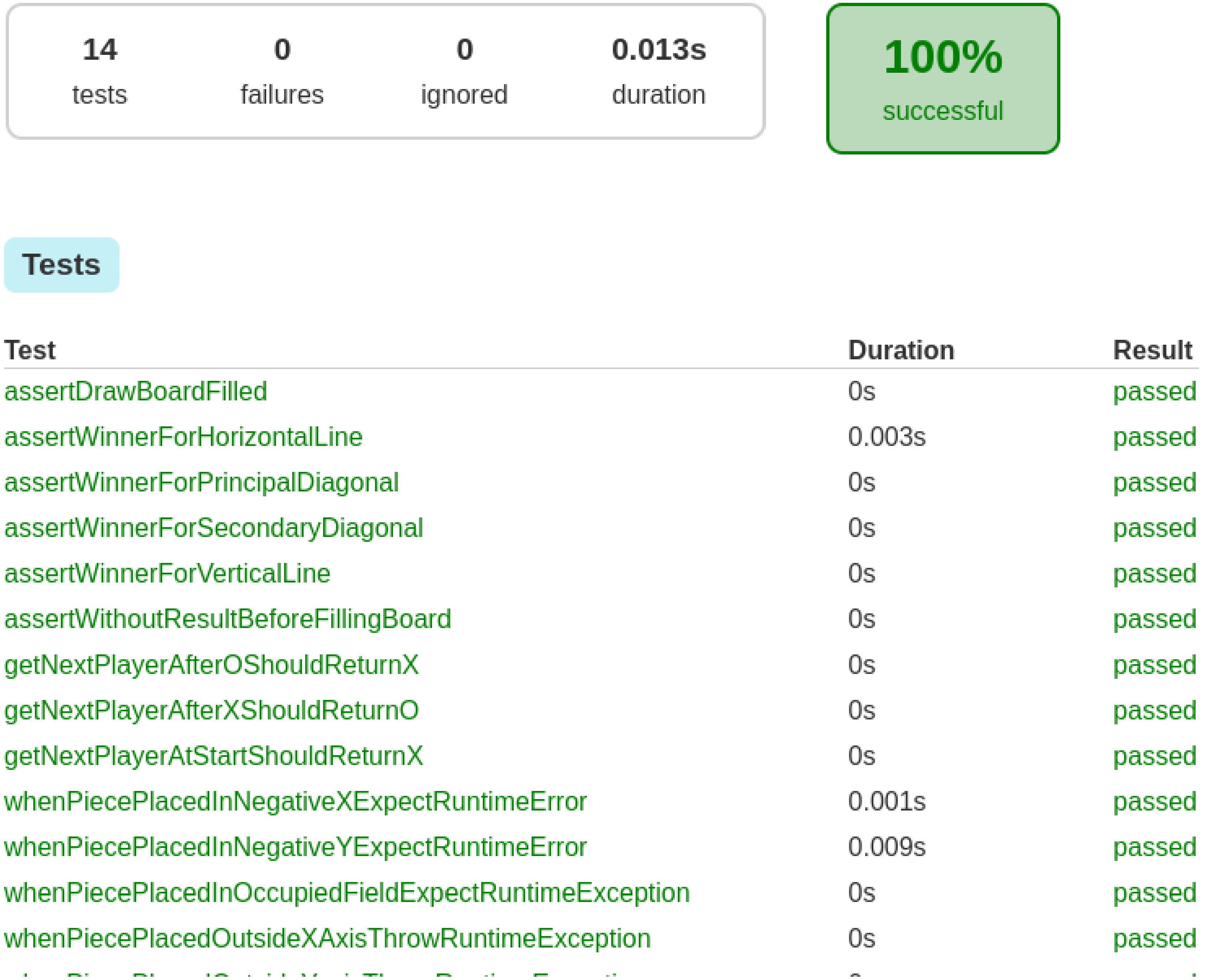
REQ4 - Green State

Implementation passing all tests and after refactoring.

TicTacToe.java

```java
    public Result getWinner() {

        Piece winner;

        for (int row = 0; row < SIZE; row++)
            if (this.board[row][0] == this.board[row][1] && this.board[row][1] == this.board[row][2]) {
                winner = this.board[row][0];
                if (winner != null) {
                    return mapPieceToResult(winner);
                }
            }

        for (int col = 0; col < SIZE; col++)
            if (this.board[0][col] == this.board[1][col] && this.board[1][col] == this.board[2][col]) {
                winner = this.board[0][col];
                if (winner != null) {
                    return mapPieceToResult(winner);
                }
            }

        if (this.board[0][0] == this.board[1][1] && this.board[1][1] == this.board[2][2]) {
            winner = this.board[0][0];
            if (winner != null) {
                return mapPieceToResult(winner);
            }
        }

        if (this.board[2][0] == this.board[1][1] && this.board[1][1] == this.board[0][2]) {
            winner = this.board[2][0];
            if (winner != null) {
                return mapPieceToResult(winner);
            }
        }

        if (this.filledSpaces != SIZE * SIZE)
            return null;

        return Result.DRAW;
    }

    private Result mapPieceToResult(Piece p) {
        if (p == null) return Result.DRAW;
        return p == Piece.X ? Result.X_WIN : Result.O_WIN;
    }
```

Jacoco Report



14	0	0	0.013s
tests	failures	ignored	duration

100%
successful

Tests

Test	Duration	Result
assertDrawBoardFilled	0s	passed
assertWinnerForHorizontalLine	0.003s	passed
assertWinnerForPrincipalDiagonal	0s	passed
assertWinnerForSecondaryDiagonal	0s	passed
assertWinnerForVerticalLine	0s	passed
assertWithoutResultBeforeFillingBoard	0s	passed
getNextPlayerAfterOShouldReturnX	0s	passed
getNextPlayerAfterXShouldReturnO	0s	passed
getNextPlayerAtStartShouldReturnX	0s	passed
whenPiecePlacedInNegativeXExpectRuntimeError	0.001s	passed
whenPiecePlacedInNegativeYExpectRuntimeError	0.009s	passed
whenPiecePlacedInOccupiedFieldExpectRuntimeException	0s	passed
whenPiecePlacedOutsideXAxisThrowRuntimeException	0s	passed

whenPiecePlacedOutsideYaxisThrowRuntimeException	0s	passed

Figura 1. Results after running `gradle test jacocoReport` (Fonte: elaborado pelo autor, 2023)

Chapter 6 - Mocking

While focused on units, we must try to remove all dependencies that a unit might use. Removal of those dependencies is accomplished through a combination of design and mocking.

Mocks are useful in situations such as:

- Objects generating nondeterministic results, such as `java.util.Date`;
- Objects that don't already exist, such as interface elements;
- Objects with slow processing, such as databases.

Mock objects simulate the behavior of real (often complex) objects. They allow us to create an object that will replace the real one used in the implementation code.

The following code extracts use this structure. The Bean represents the DB's Entity.

TicTacToeBean.java

```java
public class TicTacToeBean {
    @Id
    private final int turn;

    public int getTurn() {
        return turn;
    }

    private final int x;

    public int getX() {
        return x;
    }

    private final int y;

    public int getY() {
        return y;
    }

    private final Piece piece;

    public Piece getPiece() {
        return piece;
    }

    public TicTacToeBean(int turn, int x, int y, Piece piece) {
        this.turn = turn;
        this.x = x;
        this.y = y;
        this.piece = piece;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        TicTacToeBean that = (TicTacToeBean) o;
        return turn == that.turn && x == that.x && y == that.y && piece == that.piece;
    }

    @Override
    public int hashCode() {
        return Objects.hash(turn, x, y, piece);
    }

    @Override
    public String toString() {
        return "TicTacToeBean{" + "turn=" + turn + ", x=" + x + ", y=" + y + ", piece=" + piece + '}';
    }
}
```

Requirement 1

Implement an option to save a single move with the turn number, the X and Y axis positions, and the player (X or O).

REQ1 - Red State

Tests for this requirement:

TicTacToeCollectionSpec.java

```java
public class TicTacToeCollectionSpec {
    TicTacToeCollection collection;

    MongoCollection mongoCollection;
    private final Piece x = Piece.X;
    private final Piece o = Piece.O;

    private final TicTacToeBean bean = new TicTacToeBean(1, 0, 2, x);

    @Before
    public void before() {
        try {
            collection = spy(new TicTacToeCollection());
            mongoCollection = mock(MongoCollection.class);

        } catch (UnknownHostException e) {
            throw new RuntimeException(e);
        }
    }

    @Test
    public void whenInstantiatedThenMongoHasDbNameTicTacToe() {
        assertEquals("tic-tac-toe", collection.getMongoCollection().getDBCollection().getDB().getName());
    }

    @Test
    public void whenInstantiatedThenMongoDbHasCollectionNameGame() {
        assertEquals("game", collection.getMongoCollection().getName());
    }

    @Test
    public void whenSaveMoveThenInvokeMongoCollectionSave() {
        TicTacToeBean bean = new TicTacToeBean(1, 0, 2, x);
        doReturn(mongoCollection).when(collection).getMongoCollection();
        collection.saveMove(bean);
        verify(mongoCollection, times(1)).save(bean);
    }

    @Test
    public void assertSaveMoveReturnsTrueOnSuccess() {
        doReturn(mongoCollection).when(collection).getMongoCollection();
        assertTrue(collection.saveMove(bean));
    }

    @Test
    public void assertSaveMoveReturnsFalseOnException() {
        doThrow(new MongoException("Generic exception"))
                .when(mongoCollection)
                .save(any(TicTacToeBean.class));
        assertFalse(collection.saveMove(bean));
    }

    @Test
    public void testDropDatabase() {
        doReturn(mongoCollection).when(collection).getMongoCollection();
        collection.drop();
        verify(mongoCollection).drop();
    }

    @Test
    public void assertTrueOnDrop() {
        doReturn(mongoCollection).when(collection).getMongoCollection();
        assertTrue(collection.drop());
    }

    @Test
    public void assertFalseOnExceptionDuringDrop() {
        doThrow(new MongoException("Generic exception")).when(mongoCollection).drop();
        doReturn(mongoCollection).when(collection).getMongoCollection();
        assertFalse(collection.drop());
    }

}
```

REQ1 - Green State

Implementation passing all tests and after refactoring.

TicTacToeCollection.java

```java
public class TicTacToeCollection {
    private MongoCollection mongoCollection;

    protected MongoCollection getMongoCollection() {
        return mongoCollection;
    }

    public TicTacToeCollection() throws UnknownHostException {
        DB db = new MongoClient().getDB("tic-tac-toe");
        mongoCollection = new Jongo(db).getCollection("game");

    }

    public boolean saveMove(TicTacToeBean move) {
        try {
            getMongoCollection().save(move);
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    public boolean drop() {
        try {
            getMongoCollection().drop();
            return true;
        } catch (Exception e) {
            return false;
        }
    }

}
```

References

[1] Test-Driven Java Development, Second Edition: Invoke TDD Principles for End-To-end Application Development, 2nd Edition, by Viktor Farcic, and Alex Garcia.