1 Introduction

This lab explores the process of test driven development using the Android JUnit test framework applied to the testing of a data model of the simplified RoboCup GameController.

1. This lab is for individuals working alone. The lab work will continue outside the lab session and is not due for submission for a few days (see moodle for details).
2. Submit your completed lab to moodle. Create a zip archive of your project using **File > export as zip** and submit this to moodle
3. Be ready to demonstrate your completed lab at the start of the following lab session.

2 Overview and background

In the previous lab you developed the UI layout for a simplified RoboCup GameController app, but this app had no real functionality. That version of the app could largely be considered to be the "View" of the *Model-View-Controller* (MVC) architecture.

In this lab we will be working on the "Model" of the MVC architecture.

Furthermore, you will be using the *test-driven development (TDD)* method to ensure that your code is unit tested as you develop it. We'll use a variation that creates a stub of the method to be tested before writing the first test, as this is better suited to development in AndroidStudio.

2.1 Test driven development

Specifically, you should use the following method to develop tests.

Step 1: implement a stub of the function you want to test. The stub will have the correct method signature but does almost nothing in the method body. If the method defines a return type, then just return some default value every time (no matter what parameters are passed in or what the state of the object is.) E.g.

```
public double convert(double celsius) {
    return 0.0; // stub - just return a default value every time
}
```

Step 2: Write some test cases. To create a new test case for a particular method, select the method in the code and then type Ctrl+Shift+T and "Create New Test…". Check the "setUp" and "tearDown" boxes and check the box corresponding to the method you want to test under "Member". Choose OK.

On the next dialog, if the class/methods you want to test do not depend on Android in any way, then choose the folder under "app/src/**test**/…". Otherwise, if you do depend on some Android specifics, then choose the folder under "app/src/**androidTest**/…"

Now your test class should be generated.

Now write your test code inside the method in the test class that (currently) has the same name as the method you wish to test.

Design your test code to provide inputs to the class/method and check outputs and side effects against the expected values.

Step 3: Run the test case—it should fail. If the test case does not fail at this point you've done something wrong, since the stub code should not be able to pass all tests.

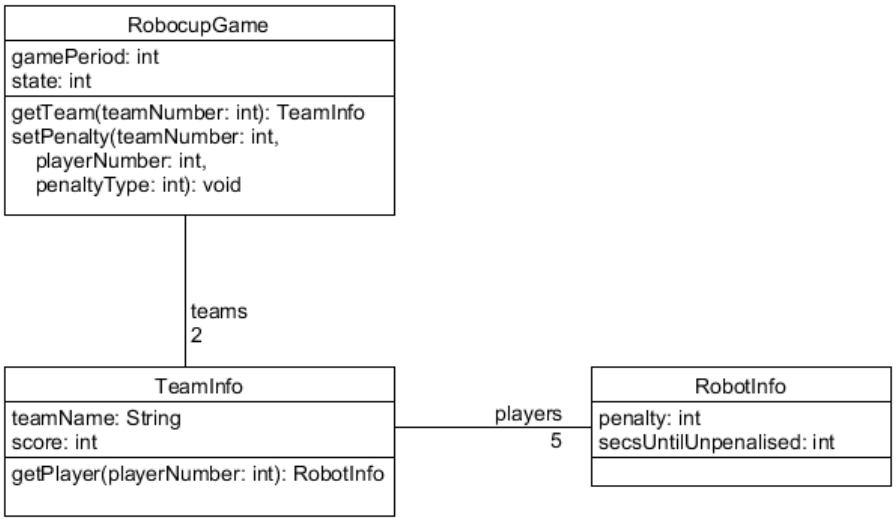
Step 4: Finally complete any missing implementation and fix bugs until the test case passes. Use black box and glass box techniques to add sufficient tests to be confident that the implementation is working. In the case of a method based on the internal state of an object it is a good idea to test the

transition between various possible states, in addition to checking that each state seems to work correctly on its own.

You should also refer to the Assignment 2 starter code which contains examples of test code for the CelsiusToFarenheitConverter that you can learn from.

2.2 Partial model

To get you started some model code is supplied as part of the lab. The model represents the data (and main behaviour) of the app that is independent of the UI. The class diagram for the model is as follows:



The model consists of the following main classes:

- **`RobotGame`**—this directly holds the data that is not specific to any team or player. It also provides a partial façade interface to the remainder of the model. Any modifications to the model (e.g. penalizing a robot) are achieved by calling methods on the RobotGame object (E.g. setState, setPenalty, etc.). It defines two nested classes (for teams and players).
- **`RobotGame.RobotInfo`**—nested class that holds the data specific to a robot player, particularly related to whether the robot is penalized or not.
- **`RobotGame.TeamInfo`**—nested class that holds the data specific to a team such as the team's name and score.

The implementation of setState and setPenalty in RobotGame are just stubs for now. You will be using test driven development to implement these methods correctly along with their accompanying test code.

The code for the partially implemented model follows. Don't copy this yet.

Listing 1—part of RobocupGame.java

```
/**
 * this class implements the model (as part of the MVC architecture)
 */
public class RobocupGame {
    public static final int MAX_NUM_PLAYERS = 5;

    public static final int PENALTY_NONE = 0;
    public static final int PENALTY_SPL_PUSHING = 2;
    public static final int PENALTY_SPL_LEAVING_THE_FIELD = 6;
    public static final int PENALTY_SPL_ILLEGAL_DEFENDER = 5;
```

```
    public static final int PERIOD_H1 = 0;
    public static final int PERIOD_H2 = 1;
    public static final int PERIOD_PENALTY_SHOOTOUT = 2;

    public static final int STATE_INITIAL = 0;
    public static final int STATE_READY = 1;
    public static final int STATE_SET = 2;
    public static final int STATE_PLAYING = 3;
    public static final int STATE_FINISHED = 4;


    // ================================================================================

    public class RobotInfo {
        private int penalty;
        private int secsUntilUnpenalised;

        RobotInfo() {
            penalty = PENALTY_NONE;
            secsUntilUnpenalised = 0;
        }

        public int getPenalty() {
            return penalty;
        }

        public int getSecsUntilUnpenalised() {
            return secsUntilUnpenalised;
        }
    }

    // ================================================================================

    public class TeamInfo {
        private String teamName;
        private int score;
        private RobotInfo players[];

        TeamInfo() {
            teamName = "not yet set";

            players = new RobotInfo[MAX_NUM_PLAYERS];
            for (int i=0; i<players.length; i++)
                players[i] = new RobotInfo();
        }

        RobotInfo getPlayer(int playerNum) {
            return players[playerNum-1];
        }
    }

    // ================================================================================

    private int gamePeriod;
    private int state;
    private TeamInfo teams[];

    /**
     * construct a new instance of the model
     */
    RobocupGame() {
        state = STATE_INITIAL;

        teams = new TeamInfo[2];
        teams[0] = new TeamInfo();
        teams[1] = new TeamInfo();
    }
```

```
    public int getGamePeriod() {
        return gamePeriod;
    }

    public int getState() {
        return state;
    }


    /**
     * get the TeamInfo for one of the two teams
     * @param teamNum - which team to get (1 or 2)
     */
    public TeamInfo getTeam(int teamNum) {
        return teams[teamNum-1];
    }

    /**
     * Set the game period.
     * This is only permitted when in the INITIAl state
     * @param gamePeriod - one of PERIOD_H1, PERIOD_H2, or PERIOD_PENALTY_SHOOTOUT
     */
    public void setGamePeriod(int gamePeriod) {
        if (state != STATE_INITIAL)
            throw new IllegalStateException("Can't change game period in this state");

        this.gamePeriod = gamePeriod;
    }

    /**
     * Change the game to a new state and unpenalize robots if needed
     * @param newState - one of state constants, i.e. STATE_INITIAL, STATE_READY,
     *                   STATE_SET, STATE_PLAY or STATE_FINISHED
     */
    public void setState(int newState) {
        // TODO: need to unpenalize robots in the INITIAL or FINISHED states

        // now change the state
        this.state = newState;
    }

    /**
     * Penalize (or unpenalize) a robot.
     * To unpenalize a robot, set the newPenalty to PENALTY_NONE
     * @param teamNum - which team the penalty applies to (1 or 2)
     * @param playerNum - which player the penalty applies to (1..5)
     * @param newPenalty - one of penalty constants, i.e. PENALTY_NONE,
     *                     PENALTY_SPL_PUSHING, PENALTY_SPL_LEAVING_THE_FIELD,
     *                     or PENALTY_SPL_ILLEGAL_DEFENDER
     */
    public void setPenalty(int teamNum, int playerNum, int newPenalty) {
        // TODO
    }
}
```

3 Implementation/requirements:

3.1 Create the project and incorporate the partially implemented model

- Create a new project called Lab5_GameController with an empty activity. I suggest that you use the package name ie.mu.ee308 for your app.

- In the project pane, select your package (under app/java), right click and choose New > Java class. Name the class "RobocupGame" and click OK. The new class should appear in editor window and the project pane.
- Copy the code from Listing 1 into the RobocupGame.java file taking care not to overwrite the existing first line which specifies the package.

3.2 setGamePeriod tests

- The partially implemented model has already implemented the setGamePeriod method to implement the following behaviour/constraints:
    - In the INITIAL state, the new game period can be set
    - In any other state, an IllegalStateException is thrown and the game period is not changed
- If we were doing test driven development, we would have started with a stub or partial implementation of the setGamePeriod method and then we would have written some tests to check for the behaviour/constraints above.
- To start writing the tests, first create a new RobocupGameTest class and test code for the setGamePeriod method by selecting the setGamePeriod method and typing Ctrl+Shift+T to generate a new test as described in the background section. Ensure you tick the setUp and tearDown methods.
    - Declare a new member variable, game, to refer to the RobocupGame object.
    - In the setup method, create new RobocupGame object and make the game attribute refer to it. In the tearDown method set game to null;
    - At this point your test class should look something very similar (identical?) to the following (except maybe for the package name at the top of the file):

Listing 2—Initial RobocupGameTest.java

```
package ie.mu.ee308.lab4_gamecontrollermodel;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.*;

public class RobocupGameTest {

    RobocupGame game;

    @Before
    public void setUp() throws Exception {
        game = new RobocupGame();
    }

    @After
    public void tearDown() throws Exception {
        game = null;
    }

    @Test
    public void setGamePeriod() {
    }

}
```

- Now we can write some suitable tests based on the constraints/behaviour specified above for the setGamePeriod method. The following listing gives some examples (though it is not exhaustive). You can copy from this listing into the setGamePeriod test method of the RobocupGameTest.java file.

Listing 3—Example setGamePeriod test code in RobocupGameTest.java

```
@Test
public void setGamePeriod() {
    // at this point we are just after setUp so the game should be in the INITIAL state.
    // Let's confirm that is the case
    assertEquals(RobocupGame.STATE_INITIAL, game.getState());

    // two simple positive tests for setGamePeriod:
    // check that we can set the game period to something new while in this INITIAL state
    game.setGamePeriod(RobocupGame.PERIOD_H2);
    assertEquals(RobocupGame.PERIOD_H2, game.getGamePeriod());

    game.setGamePeriod(RobocupGame.PERIOD_PENALTY_SHOOTOUT);
    assertEquals(RobocupGame.PERIOD_PENALTY_SHOOTOUT, game.getGamePeriod());

    // one simple negative test for setGamePeriod
    // check that we cannot set the game period when not in the INITIAL state

    game.setState(RobocupGame.STATE_READY);
    int periodBefore = game.getGamePeriod();

    try {
        game.setGamePeriod(RobocupGame.PERIOD_H1);
        // an exception should be thrown which will jump us straight to the catch.
        // If no exception is thrown we mark the test a failure
        fail("An IllegalStateException should have been thrown");
    } catch (IllegalStateException e) {
        // if we get here the exception was thrown - verify that the period is unchanged
        assertEquals(periodBefore, game.getGamePeriod());
    }
}
```

- You should run the test method and verify that it passes (which means all the tests within it pass). Normally at this point in test driven development, we wouldn't expect the tests to pass because we would expect that the setGamePeriod method of the RobocupGame class would only be a stub and we would still have to fill in the real implementation. However, in this case the real implementation has already been done so the tests should pass.
- Review the test code to ensure you understand how the test code relates to the behaviour and constraints that were specified. Also note how we specify the inputs to tests as the input parameter to setGamePeriod but since this method has no return value, we check the outcome of the test by inspecting the result of the getGamePeriod method.

3.3 setPenalty tests and method

- The setPenalty method must allow a robot to be penalized or unpenalized. It must implement the following constraints:
    - In the INITIAL, SET, or FINISHED states it is only permitted to unpenalize a robot (i.e. to set PENALTY_NONE) – attempting to set any other penalty should cause an IllegalStateException to be thrown
    - Any penalty (including PENALTY_NONE) can be set on a robot that is currently unpenalized.

  - If a robot is currently penalized (i.e. the robot's current penalty is not PENALTY_NONE) then no other penalty can be set for this robot except PENALTY_NONE to unpenalize it. Any attempt to penalize an already-penalized robot should throw an IllegalStateException.
  - When a robot is penalized, its secsToUnpenalized attribute is set to 45 seconds. When a robot is unpenalized, this value is set to zero.

- To get access to a PlayerInfo object in order to read or change the penalty values we need to remember that teamNum and playerNum both start from 1 whereas the teams and players arrays use zero-based indexes as normal. Hence use code like the following:
  `RobotInfo player = teams[teamNum-1].players[playerNum-1];`
- Create test code for the setPenalty method by selecting it and typing Ctrl+Shift+T as before. Check the setPenalty method in the dialog. You'll be prompted to update the existing test class and you can click OK. I suggest renaming the generated test method to be setPenaltyPositiveTests and adding a similar setPenaltyNegativeTests method to your test class also. (You can just copy and paste setPenaltyPositiveTests and rename the copy.)
- Implement a comprehensive set of suitable tests for the setPenalty method based on the constraints described above. For each test, decide upon some input values for which you expect particular outcomes. (In this case, the outcome of interest will usually be the penalty value and secsToUnpenalized value of the player after the setPenalty function returns.) Check that the outcomes match what you expect using either `assertEquals` to check that 2 values are equal or `assertTrue` to check that some condition is true. Refer to the tests for setGamePeriod to see how to handle tests that should result in an exception being thrown.
- Verify that all tests pass.

3.4 setState tests and method

The existing setState method just sets the state to the requested value and implements no constraints. It would not be necessary to test such a method. However we need to modify/extend the method to implement some constraints and additional behaviour so you'll have to add both test code and the modified implementation of the method as follows.

- The setState method must only permit certain state changes and must implement some additional behaviour as follows:
  - Only the following state transitions are permitted. An attempt to set any other state transition should throw an IllegalStateException:
    - From INITIAL to READY or PLAYING
    - From READY to SET or FINISHED
    - From SET to PLAYING or FINISHED
    - From PLAYING to INITIAL, READY, or FINISHED
    - From FINISHED to INITIAL

    Hint: You'll need to check the current state (before you change it) and the new state to make sure the planned transition is permitted.
  - On transitioning to the INITIAL or FINISHED states all currently penalized robots must be unpenalized

    Hint: You'll need to iterate over all teams and then over all players to unpenalize where necessary.

- Create a new test method for the setState method using the same approach as used for setPenalty. (Alternatively, you can manually add a method to the RobocupGameTest class.) It will be part of the same test class as before, i.e. RobocupGameTest.java
- Implement a comprehensive set of suitable tests for the modified setState method based on the constraints and extra behaviour described above. For each test, decide upon some input values for which you expect particular outcomes. Check that the outcomes match what you expect. You'll need to modify/extend the implementation of the setState method in the RobocupGame class to make all the tests pass.
- Verify that all tests pass.