Group assignment 3: Tests

Þróun hugbúnaðar
Spring 2015

Students: (Group F2a)
Einar Helgi Þrastarson
Hannes Pétur Eggertsson
Sigurður Birkir Sigurðsson

Teachers:
Matthias Book
Kristín Fjóla Tómasdóttir

1 Introduction

In this document there's information about the tests for group F2a. Group members are: Einar Helgi Þrastarson (personal ID number: 110287-2919), Hannes Pétur Eggertsson (240889-2939) and Sigurður Birkir Sigurðsson (120589-2539). Our project is to build an user interface for a fantasy football game.

The presenter on Wednesday, March 11th 2015, will be Sigurður Birkir Sigurðsson.

2 Test fixtures

We created test fixture using JUnit to constantly test our core functionality. The core functionality we decided to test was the roster class. We have made some small changes to the class and currently it looks as such:

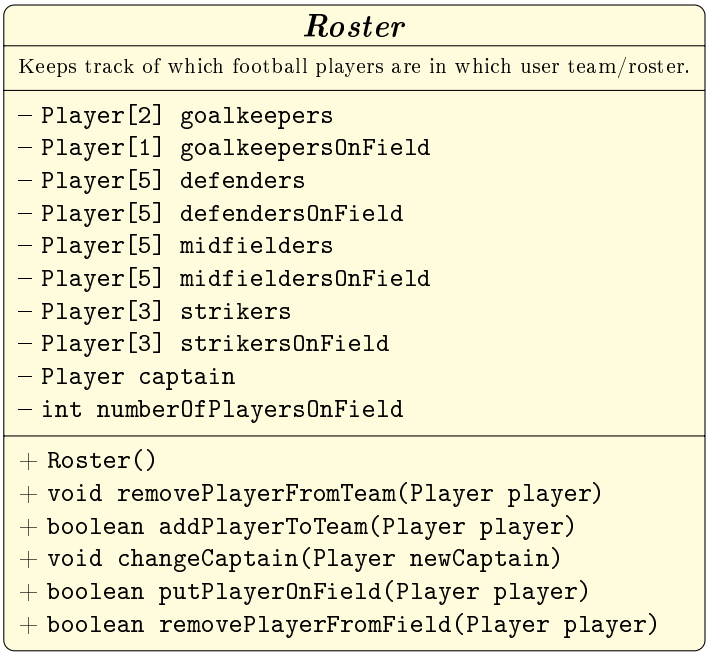

Figure 1: The modified Roster class

The class has been created and will be the target for our test fixtures. The full source code of the class can be seen in the appendix.

2.1 Set up and tear down

The source code of our set up and tear down is long and boring. To make a long story short we created multiple PlayerMock (more information on those in the next section) objects on the form:

```
position1 = new PlayerMock("Position 1","Position");
```

where position is a position of the player, e.g. goalkeeper or a midfielder. There is also one different PlayerMock object:

```
invalid_pos1 = new PlayerMock("Football fan","Couch potato");
```

This is test how the class will handle a player of invalid position. After all players have been created they are all added to a hash-table `players` with their name a the key. If you wish to view the full source code in can be found in the appendix.

2.2 Comparison of lists containing lists

To help us in the tests it made sense to create a function that will compare two lists containing lists. One containing lists of lists with strings with the expected player names and one lists of lists containing a class of type PlayerInterface (an interface PlayerMock implemented).

```
1  // Usage: m = compareListsOfLists(expected,actual)
2  // Before:expected is a list of lists with playernames in a String and actual
3  //        is a list of lists with players of type PlayerInterface.
4  //        From PlayerInterface we can retrieve the player's name with the
5  //        getName() method.
6  // After: If the list of lists contain the same players m will be the number of
7  //        players were matched. If the lists do not contain the same players
8  //        m will be returned as -1.
9  public int compareListsOfLists(List<List<String>> expected,
       List<List<PlayerInterface>> actual) throws IllegalStateException{
10     // Count the number of matches
11     int matches = 0;
12     // If the two lists (of lists) have different number of elements, throw
           exception.
13     if (actual.size() != expected.size()){
14         throw new IllegalStateException("Sizes of lists containing lists not
               the same: "+expected.size()+" and "+actual.size());
15     }
16     // Create an iterator for both lists (of lists)
17     Iterator<List<PlayerInterface>> playerlist_iterator = actual.iterator();
18     Iterator<List<String>> expected_playerlist_iterator = expected.iterator();
19     // Loop through the outer lists
20     while(playerlist_iterator.hasNext()){
21         List<PlayerInterface> playerlist = playerlist_iterator.next();
22         List<String> expected_playerlist = expected_playerlist_iterator.next();
23
24         if (playerlist.size() != expected_playerlist.size()){
25             throw new IllegalStateException("Sizes of lists not the same:
                   "+expected_playerlist.size()+" and "+playerlist.size());
26         }
27
28         Iterator<PlayerInterface> player_iterator = playerlist.iterator();
29         Iterator<String> expected_player_iterator =
               expected_playerlist.iterator();
30
31         // Loop through the inner lists
32         while(player_iterator.hasNext()){
33             String expected_player = expected_player_iterator.next();
34             if(player_iterator.next().getName() != expected_player){
35                 return -1;
36             }
37             else{
38                 matches++;
39             }
40         }
41     }
42     return matches;
43 }
```

2.3 Testing the addPlayerToRoster method

2.3.1 Test 1: List, art thou empty?

This test will check if a new roster is empty.

```
public void testIfEmpty() throws IllegalStateException, InvalidPosition {
    List<List<PlayerInterface>> actual = roster.getPlayersInRoster();
    List<List<String>> excepted = new ArrayList<List<String>>(4)
       {{add(goalkeepers);add(defenders);add(midfielders);add(strikers);}};
    assertEquals(0,compareListsOfLists(excepted, actual));
}
```

2.3.2 Test 2: Takes one to know one

This test will check if we can successfully add a single player to the roster.

```
public void testIfOnePlayer() throws IllegalStateException, InvalidPosition {
    // Add the player "Goalkeeper 1" to the roster
    boolean add = roster.addPlayerToRoster(players.get("Goalkeeper 1"));
    assertTrue(add);

    // Get the roster players
    List<List<PlayerInterface>> actual = roster.getPlayersInRoster();

    // Create the expected outcome of the test
    goalkeepers.add("Goalkeeper 1");
    List<List<String>> excepted = new ArrayList<List<String>>(4)
       {{add(goalkeepers);add(defenders);add(midfielders);add(strikers);}};

    assertEquals(1,compareListsOfLists(excepted, actual));
}
```

2.3.3 Test 3: Who invited you?

This test will check if we get an exception when adding a player with a invalid position. We expect to get a `InvalidPosition` exception in that case.

```
public void testIfInvalidPlayer() throws InvalidPosition {
    Throwable exception = null;
    // Add the player "Football fan" to the roster
    try{
        roster.addPlayerToRoster(players.get("Football fan"));
    } catch (Throwable e) {
        exception = e;
    }
    assertNotNull(exception);
    assertSame(InvalidPosition.class,exception.getClass());
}
```

2.3.4 Test 4: Only two can tango

This test will check if we will receive "false" from the `addPlayerToRoster()` method if we try to add too many players to the same position.

```
public void testIfThreePlayers() throws InvalidPosition {
    // Add the player "Goalkeeper 1" to the roster
```

```
    roster.addPlayerToRoster(players.get("Goalkeeper 1"));
    boolean add = roster.addPlayerToRoster(players.get("Goalkeeper 2"));
    assertTrue(add);
    add = roster.addPlayerToRoster(players.get("Goalkeeper 3"));
    assertFalse(add);

    // Get the roster players
    List<List<PlayerInterface>> actual = roster.getPlayersInRoster();

    // Create the expected outcome of the test
    goalkeepers.add("Goalkeeper 1");
    goalkeepers.add("Goalkeeper 2");
    List<List<String>> excepted = new ArrayList<List<String>>(4)
        {{add(goalkeepers);add(defenders);add(midfielders);add(strikers);}};

    assertEquals(2,compareListsOfLists(excepted, actual));
}
```

2.3.5 Test 5: No more room in heaven

```
public void testIfFullRoster() throws InvalidPosition {
    // Add the player "Goalkeeper 1" to the roster
    roster.addPlayerToRoster(players.get("Goalkeeper 1"));
    roster.addPlayerToRoster(players.get("Goalkeeper 2"));
    boolean add;
    add = roster.addPlayerToRoster(players.get("Defender 1"));  assertTrue(add);
    add = roster.addPlayerToRoster(players.get("Defender 2"));  assertTrue(add);
    add = roster.addPlayerToRoster(players.get("Defender 3"));  assertTrue(add);
    add = roster.addPlayerToRoster(players.get("Defender 4"));  assertTrue(add);
    add = roster.addPlayerToRoster(players.get("Defender 5"));  assertTrue(add);
    add = roster.addPlayerToRoster(players.get("Midfielder 1"));
        assertTrue(add);
    add = roster.addPlayerToRoster(players.get("Midfielder 2"));
        assertTrue(add);
    add = roster.addPlayerToRoster(players.get("Midfielder 3"));
        assertTrue(add);
    add = roster.addPlayerToRoster(players.get("Midfielder 4"));
        assertTrue(add);
    add = roster.addPlayerToRoster(players.get("Midfielder 5"));
        assertTrue(add);
    add = roster.addPlayerToRoster(players.get("Striker 1"));
        assertTrue(add);
    add = roster.addPlayerToRoster(players.get("Striker 2"));
        assertTrue(add);
    add = roster.addPlayerToRoster(players.get("Striker 3"));
        assertTrue(add);

    // Get the roster players
    List<List<PlayerInterface>> actual = roster.getPlayersInRoster();

    // Create the expected outcome of the test
    goalkeepers.add("Goalkeeper 1");
    goalkeepers.add("Goalkeeper 2");
```

```
26      defenders.add("Defender 1");
27      defenders.add("Defender 2");
28      defenders.add("Defender 3");
29      defenders.add("Defender 4");
30      defenders.add("Defender 5");
31      midfielders.add("Midfielder 1");
32      midfielders.add("Midfielder 2");
33      midfielders.add("Midfielder 3");
34      midfielders.add("Midfielder 4");
35      midfielders.add("Midfielder 5");
36      strikers.add("Striker 1");
37      strikers.add("Striker 2");
38      strikers.add("Striker 3");
39      List<List<String>> excepted = new ArrayList<List<String>>(4)
            {{add(goalkeepers);add(defenders);add(midfielders);add(strikers);}};
40
41      assertEquals(15,compareListsOfLists(excepted, actual));
42  }
```

2.4 Testing the addPlayerToField method

2.4.1 Test 6: We must follow the rules

This test will check if we can add too many players to the same position

```
1  public void testIfAddGoalkeepers() throws InvalidPlayer, InvalidPosition {
2      roster.addPlayerToRoster(players.get("Goalkeeper 1"));
3      roster.addPlayerToRoster(players.get("Goalkeeper 2"));
4      boolean b = roster.addPlayerToField(players.get("Goalkeeper 1"));
5      assertTrue(b);
6      b = roster.addPlayerToField(players.get("Goalkeeper 2"));
7      assertFalse(b);
8  }
```

2.4.2 Test 7: You can't play with us

This test will check if we can successfully add eleven players to the field and can't add the twelfth.

```
1  public void testIfAddElevenAndTwelveToField() throws InvalidPlayer,
       InvalidPosition {
2      // All 15 test players available in roster
3      roster.addPlayerToRoster(players.get("Goalkeeper 1"));
4      roster.addPlayerToRoster(players.get("Goalkeeper 2"));
5      roster.addPlayerToRoster(players.get("Defender 1"));
6      roster.addPlayerToRoster(players.get("Defender 2"));
7      roster.addPlayerToRoster(players.get("Defender 3"));
8      roster.addPlayerToRoster(players.get("Defender 4"));
9      roster.addPlayerToRoster(players.get("Defender 5"));
10     roster.addPlayerToRoster(players.get("Midfielder 1"));
11     roster.addPlayerToRoster(players.get("Midfielder 2"));
12     roster.addPlayerToRoster(players.get("Midfielder 3"));
13     roster.addPlayerToRoster(players.get("Midfielder 4"));
14     roster.addPlayerToRoster(players.get("Midfielder 5"));
15     roster.addPlayerToRoster(players.get("Striker 1"));
16     roster.addPlayerToRoster(players.get("Striker 2"));
```

```
    roster.addPlayerToRoster(players.get("Striker 3"));

    boolean b;
    roster.addPlayerToField(players.get("Goalkeeper 1"));
    b = roster.addPlayerToField(players.get("Defender 1"));     assertTrue(b);
    b = roster.addPlayerToField(players.get("Defender 2"));     assertTrue(b);
    b = roster.addPlayerToField(players.get("Midfielder 1"));   assertTrue(b);
    b = roster.addPlayerToField(players.get("Midfielder 2"));   assertTrue(b);
    b = roster.addPlayerToField(players.get("Midfielder 3"));   assertTrue(b);
    b = roster.addPlayerToField(players.get("Midfielder 4"));   assertTrue(b);
    b = roster.addPlayerToField(players.get("Midfielder 5"));   assertTrue(b);
    b = roster.addPlayerToField(players.get("Striker 1"));      assertTrue(b);
    b = roster.addPlayerToField(players.get("Striker 2"));      assertTrue(b);
    b = roster.addPlayerToField(players.get("Striker 3"));      assertTrue(b);

    // Test if adding a player that is not in the roster will throw the
       InvalidPlayer exception
    Throwable exception = null;
    try{
        roster.addPlayerToField(players.get("Football fan"));
    } catch (Throwable e) {
        exception = e;
    }
    assertNotNull(exception);
    assertSame(InvalidPlayer.class,exception.getClass());

    // Test if we're not able to add the 12th player to the field
    b = roster.addPlayerToField(players.get("Defender 3"));
    assertFalse(b);
}
```

2.5 Testing the removeFromRoster method

2.5.1 Test 8: Join us! Now leave us!

Test if we remove a player that's in the roster.

```
public void testRemoveFromRoster() throws InvalidPosition, InvalidPlayer{
    roster.addPlayerToRoster(players.get("Goalkeeper 1"));
    roster.removePlayer(players.get("Goalkeeper 1"), true);

    // Check if the roster is empty
    List<List<PlayerInterface>> actual = roster.getPlayersInRoster();
    List<List<String>> excepted = new ArrayList<List<String>>(4)
       {{add(goalkeepers);add(defenders);add(midfielders);add(strikers);}};
    assertEquals(0,compareListsOfLists(excepted, actual));
}
```

2.5.2 Test 9: Go away nobody

Test if we remove a player that's NOT in the roster. We expect it to thrown an exception.

```
public void testRemoveInvalidPlayer() {
    Throwable exception = null;
    try{
```

```
        roster.removePlayer(players.get("Goalkeeper 1"), true);
    } catch (Throwable e) {
        exception = e;
    }
    assertNotNull(exception);
    assertSame(InvalidPlayer.class,exception.getClass());
}
```

2.5.3 Test 10: 1,2,3,...

This test if check if the variable NumberOfPlayersOfField is changed correctly.

```
public void testNumberOfPlayersOnField() throws InvalidPosition, InvalidPlayer {
    assertEquals(0,roster.getNumberOfPlayersOnField());
    roster.addPlayerToRoster(players.get("Goalkeeper 1"));
    roster.addPlayerToRoster(players.get("Defender 2"));
    assertEquals(0,roster.getNumberOfPlayersOnField());
    roster.addPlayerToField(players.get("Goalkeeper 1"));
    assertEquals(1,roster.getNumberOfPlayersOnField());
    roster.addPlayerToField(players.get("Defender 2"));
    assertEquals(2,roster.getNumberOfPlayersOnField());
    roster.removePlayer(players.get("Goalkeeper 1"), false);
    assertEquals(1,roster.getNumberOfPlayersOnField());
    roster.removePlayer(players.get("Defender 2"), true);
    assertEquals(0,roster.getNumberOfPlayersOnField());
}
```

3 Mock objects

In order to have the test fixtures above we needed to create a mock up class for the player, we call PlayerMockup. Into this class we put the most basic information about the player and didn't create any unnecessary methods the real Player class will have when it's created by group F1a.

```
1  public class PlayerMock implements PlayerInterface {
2
3      private String name;
4      private PositionMock position;
5      private String positionName;
6
7      public PlayerMock(String name, String pos){
8          this.name = name;
9          this.positionName = pos;
10     }
11
12     public String getName(){
13         return this.name;
14     }
15
16     @Override
17     public String getPositionName() {
18         return this.positionName;
19     }
20
21     @Override
22     public void setPosition(PositionMock pos) throws InvalidPosition{
23         if (pos.equals("Goalkeeper") || pos.equals("Defender") ||
               pos.equals("Midfielder") || pos.equals("Striker")){
24             this.position = pos;
25         } else {
26             throw new InvalidPosition(pos+" is not a valid position. Only
                   Goalkeeper, Defender, Midfielder, and Striker are valid.");
27         }
28     }
29
30     @Override
31     public PositionMock getPosition() {
32         return position;
33     }
34
35 }
```

This class implements the PlayerInterface

```
1  public interface PlayerInterface {
2      public String getName();
3      public void setPosition(PositionMock pos) throws InvalidPosition;
4      public PositionMock getPosition();
5      public String getPositionName();
6  }
```

4 Test cases

We decided to create test cases for the search feature on the market panel. The goal of the feature is to provide a accurate search of all players and possible have some useful filters. The filters we chose to include in these test cases were "Teams", and "Position", i.e. you can choose a team and/or position to filter out players. The description of the function:

```
1 // Usage: matches = searchPlayers(String search_term, Team filtered_team,
2 //        Position filtered_position)
3 // Before:search_term is the term currently being searched for, filtered_team
4 //        is the filtered team (can be null if not specified), and
5 //        filtered_position is a position for a player (can also be null if
6 //        not specified).
7 // After: Search results for the the search_term using (if not null) the two
8 //        filters. Match is when the search_term is a substring of the
9 //        player's name. All matches are displayed on the screen.
```

A table of some test cases we would use to test this feature would be:

Before			**After**	
Search term	**Team**	**Position**	**Expected results**	**Explanation**
""	null	null	Aaron Lennon, Aaron Ramsey,...	All players
Rooney	null	null	Wayne Rooney, John Rooney,...	All players with Rooney as a subsrting
Rooney	null	Striker	Wayne Rooney, Adam Rooney,...	All strikers having the Rooney subsrtring
Rooney	null	Goalkeeper	""	All Rooney's in that are Goalkeepers
Rooney	Chelsea	null	""	All Rooney's in Chelsea
Rooney	Man. Utd.	null	Wayne Rooney	All Rooney's in Man. Utd.
Rooney	Man. Utd.	Striker	Wayne Rooney	All Rooney's in Man. Utd. and are strikers
Wayne Rooney	null	null	Wayne Rooney	All players with the Wayne Rooney substring

Appendix

RosterTest.java

Set up before the class method

```
public static void setUpBeforeClass() {
    goalkeeper1 = new PlayerMock("Goalkeeper 1","Goalkeeper");
    goalkeeper2 = new PlayerMock("Goalkeeper 2","Goalkeeper");
    goalkeeper3 = new PlayerMock("Goalkeeper 3","Goalkeeper");

    invalid_pos1 = new PlayerMock("Football fan","Couch potato");

    defender1 = new PlayerMock("Defender 1","Defender");
    defender2 = new PlayerMock("Defender 2","Defender");
    defender3 = new PlayerMock("Defender 3","Defender");
    defender4 = new PlayerMock("Defender 4","Defender");
    defender5 = new PlayerMock("Defender 5","Defender");

    midfielder1 = new PlayerMock("Midfielder 1","Midfielder");
    midfielder2 = new PlayerMock("Midfielder 2","Midfielder");
    midfielder3 = new PlayerMock("Midfielder 3","Midfielder");
    midfielder4 = new PlayerMock("Midfielder 4","Midfielder");
    midfielder5 = new PlayerMock("Midfielder 5","Midfielder");

    striker1 = new PlayerMock("Striker 1","Striker");
    striker2 = new PlayerMock("Striker 2","Striker");
    striker3 = new PlayerMock("Striker 3","Striker");

    players = new HashMap<String, PlayerMock>();
    players.put(goalkeeper1.getName(),goalkeeper1);
    players.put(goalkeeper2.getName(),goalkeeper2);
    players.put(goalkeeper3.getName(),goalkeeper3);

    players.put(invalid_pos1.getName(),invalid_pos1);

    players.put(defender1.getName(),defender1);
    players.put(defender2.getName(),defender2);
    players.put(defender3.getName(),defender3);
    players.put(defender4.getName(),defender4);
    players.put(defender5.getName(),defender5);

    players.put(midfielder1.getName(),midfielder1);
    players.put(midfielder2.getName(),midfielder2);
    players.put(midfielder3.getName(),midfielder3);
    players.put(midfielder4.getName(),midfielder4);
    players.put(midfielder5.getName(),midfielder5);

    players.put(striker1.getName(), striker1);
    players.put(striker2.getName(), striker2);
    players.put(striker3.getName(), striker3);
}
```

Before each test method

```
@Before
public void setUp() throws Exception {
    roster = new Roster();
    goalkeepers = new ArrayList<String>(2);
    defenders = new ArrayList<String>(5);
    midfielders = new ArrayList<String>(5);
    strikers = new ArrayList<String>(3);
}
```

After each test method

```
@After
public void tearDown() throws Exception {
    roster = null;
    goalkeepers = null;
    defenders = null;
    midfielders = null;
    strikers = null;
}
```