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 some of functionality on the roster class and the user class.

2.1 Roster test fixtures

We have made some small changes to the class and currently it looks as such:

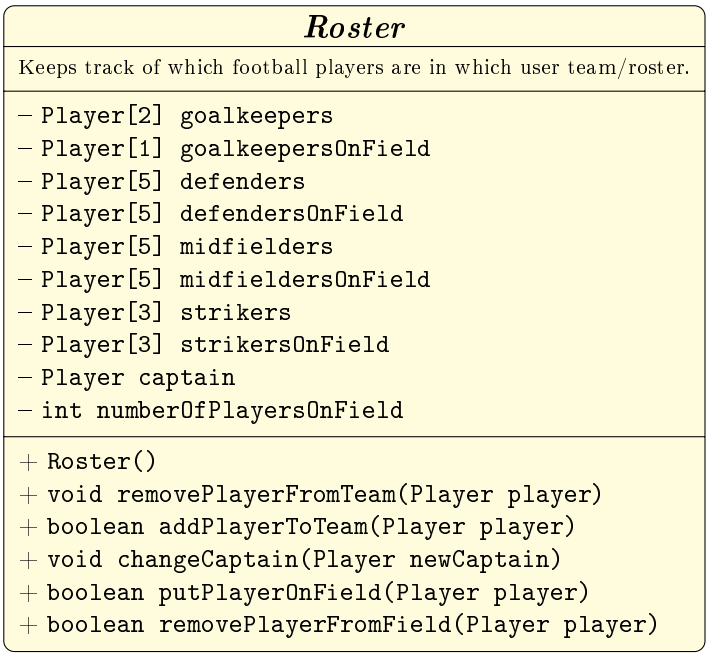

Figure 1: The modified Roster class

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.

2.1.1 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.

2.1.2 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 }
```

2.1.3 Testing the addPlayerToRoster method

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));
}
```

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));
}
```

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());
}
```

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));
}
```

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  }
```

2.1.4 Testing the addPlayerToField method

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  }
```

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"));
```

```
17      roster.addPlayerToRoster(players.get("Striker 3"));
18
19      boolean b;
20      roster.addPlayerToField(players.get("Goalkeeper 1"));
21      b = roster.addPlayerToField(players.get("Defender 1"));     assertTrue(b);
22      b = roster.addPlayerToField(players.get("Defender 2"));     assertTrue(b);
23      b = roster.addPlayerToField(players.get("Midfielder 1"));   assertTrue(b);
24      b = roster.addPlayerToField(players.get("Midfielder 2"));   assertTrue(b);
25      b = roster.addPlayerToField(players.get("Midfielder 3"));   assertTrue(b);
26      b = roster.addPlayerToField(players.get("Midfielder 4"));   assertTrue(b);
27      b = roster.addPlayerToField(players.get("Midfielder 5"));   assertTrue(b);
28      b = roster.addPlayerToField(players.get("Striker 1"));      assertTrue(b);
29      b = roster.addPlayerToField(players.get("Striker 2"));      assertTrue(b);
30      b = roster.addPlayerToField(players.get("Striker 3"));      assertTrue(b);
31
32      // Test if adding a player that is not in the roster will throw the
            InvalidPlayer exception
33      Throwable exception = null;
34      try{
35          roster.addPlayerToField(players.get("Football fan"));
36      } catch (Throwable e) {
37          exception = e;
38      }
39      assertNotNull(exception);
40      assertSame(InvalidPlayer.class,exception.getClass());
41
42      // Test if we're not able to add the 12th player to the field
43      b = roster.addPlayerToField(players.get("Defender 3"));
44      assertFalse(b);
45  }
```

2.1.5 Testing the removeFromRoster method

Test 8: Join us! Now leave us!

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

```
1   public void testRemoveFromRoster() throws InvalidPosition, InvalidPlayer{
2       roster.addPlayerToRoster(players.get("Goalkeeper 1"));
3       roster.removePlayer(players.get("Goalkeeper 1"), true);
4
5       // Check if the roster is empty
6       List<List<PlayerInterface>> actual = roster.getPlayersInRoster();
7       List<List<String>> excepted = new ArrayList<List<String>>(4)
            {{add(goalkeepers);add(defenders);add(midfielders);add(strikers);}};
8       assertEquals(0,compareListsOfLists(excepted, actual));
9   }
```

Test 9: Go away nobody

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

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

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

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());
}
```

2.2 User test fixtures

2.2.1 Set up and tear down

This test, tests the class that holds onto the users of the game. We setup the test with one new user which takes in "user1" as its name.

```
@BeforeClass
public static void setUp() throws Exception {
    user = new User("user1");
}
```

```
@AfterClass
public static void tearDown() throws Exception {
    user = null;
}
```

2.2.2 Testing the user classes

Test 1: Remember kids, null != empty

This test checks if the user has a Roster.

```
@Test
public void checkRoster() {
    assertNotNull(user.getRoster());
}
```

Test 2: Mannanafnanefnd

This test checks if a name change goes though.

```
@Test
public void nameChange() {
    user.changeName("user2");
    assertEquals("user2", user.getName());
}
```

Test 3: Make sure to put it in the bank

```
@Test
public void testChangingMoney() {

    // Money is 0
    assertTrue(user.changeMoney(1000));
    assertEquals(1000, user.getMoney());

    // Money is 1000
    int curr = user.getMoney();
    assertTrue(user.changeMoney(-900));
    assertEquals(curr-900, user.getMoney());

    // Money is 100
    curr = user.getMoney();
    assertFalse(user.changeMoney(-200));
    assertEquals(curr, user.getMoney());
    // Money is still 100 cause you can't get negative money
}
```

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  }
```

The class is made so it will work successfully on the test fixtures but doesn't include any other information or methods not used.

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

```
1  public static void setUpBeforeClass() {
2      goalkeeper1 = new PlayerMock("Goalkeeper 1","Goalkeeper");
3      goalkeeper2 = new PlayerMock("Goalkeeper 2","Goalkeeper");
4      goalkeeper3 = new PlayerMock("Goalkeeper 3","Goalkeeper");
5
6      invalid_pos1 = new PlayerMock("Football fan","Couch potato");
7
8      defender1 = new PlayerMock("Defender 1","Defender");
9      defender2 = new PlayerMock("Defender 2","Defender");
10     defender3 = new PlayerMock("Defender 3","Defender");
11     defender4 = new PlayerMock("Defender 4","Defender");
12     defender5 = new PlayerMock("Defender 5","Defender");
13
14     midfielder1 = new PlayerMock("Midfielder 1","Midfielder");
15     midfielder2 = new PlayerMock("Midfielder 2","Midfielder");
16     midfielder3 = new PlayerMock("Midfielder 3","Midfielder");
17     midfielder4 = new PlayerMock("Midfielder 4","Midfielder");
18     midfielder5 = new PlayerMock("Midfielder 5","Midfielder");
19
20     striker1 = new PlayerMock("Striker 1","Striker");
21     striker2 = new PlayerMock("Striker 2","Striker");
22     striker3 = new PlayerMock("Striker 3","Striker");
23
24     players = new HashMap<String, PlayerMock>();
25     players.put(goalkeeper1.getName(),goalkeeper1);
26     players.put(goalkeeper2.getName(),goalkeeper2);
27     players.put(goalkeeper3.getName(),goalkeeper3);
28
29     players.put(invalid_pos1.getName(),invalid_pos1);
30
31     players.put(defender1.getName(),defender1);
32     players.put(defender2.getName(),defender2);
33     players.put(defender3.getName(),defender3);
34     players.put(defender4.getName(),defender4);
35     players.put(defender5.getName(),defender5);
36
37     players.put(midfielder1.getName(),midfielder1);
38     players.put(midfielder2.getName(),midfielder2);
39     players.put(midfielder3.getName(),midfielder3);
40     players.put(midfielder4.getName(),midfielder4);
41     players.put(midfielder5.getName(),midfielder5);
42
43     players.put(striker1.getName(), striker1);
44     players.put(striker2.getName(), striker2);
45     players.put(striker3.getName(), striker3);
46 }
```

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;
}
```

Roster.java

```
import java.util.ArrayList;
import java.util.List;

import tests.*;

public class Roster {
    private List<PlayerInterface> goalkeepers;
    private List<PlayerInterface> goalkeeperOnField;
    private List<PlayerInterface> defenders;
    private List<PlayerInterface> defendersOnField;
    private List<PlayerInterface> midfielders;
    private List<PlayerInterface> midfieldersOnField;
    private List<PlayerInterface> strikers;
    private List<PlayerInterface> strikersOnField;
    // private PlayerInterface captain;
    private int numberOfPlayersOnField;

    public Roster(){
        this.numberOfPlayersOnField = 0;
        this.goalkeepers = new ArrayList<PlayerInterface>(2);
        this.goalkeeperOnField = new ArrayList<PlayerInterface>(1);
        this.defenders = new ArrayList<PlayerInterface>(5);
        this.defendersOnField = new ArrayList<PlayerInterface>(5);
        this.midfielders = new ArrayList<PlayerInterface>(5);
        this.midfieldersOnField = new ArrayList<PlayerInterface>(5);
        this.strikers = new ArrayList<PlayerInterface>(3);
        this.strikersOnField = new ArrayList<PlayerInterface>(3);
    }

    // Usage: i = getNumberOfPlayersOnField()
```

```
31      // Before:Nothing.
32      // After: i is the number of players currently on the field.
33      public int getNumberOfPlayersOnField(){
34          return this.numberOfPlayersOnField;
35      }
36
37      // Usage: removePlayer(player,b)
38      // Before:player is of type PlayerInterface and b is a boolean variable
            (true or false)
39      // After: If b is true then player will be removed both from the field and
            the roster. If
40      //        b is false then the player will only be removed from the field.
            If the player
41      //        provided is not in the roster then a InvalidPlayer exception will
            be thrown.
42      public void removePlayer(PlayerInterface player, boolean removeFromRoster)
            throws InvalidPlayer{
43          String posName = player.getPositionName();
44          if (posName.toLowerCase().equals("goalkeeper")){
45              if (removeFromRoster){
46                  boolean b = goalkeepers.remove(player);
47                  if (!b) throw new tests.InvalidPlayer("The player
                        "+player.getName()+" isn't in this roster");
48              }
49              goalkeeperOnField.remove(player);
50              numberOfPlayersOnField--;
51          } else if (posName.toLowerCase().equals("defender")){
52              if (removeFromRoster){
53                  boolean b = defenders.remove(player);
54                  if (!b) throw new tests.InvalidPlayer("The player
                        "+player.getName()+" isn't in this roster");
55              }
56              defendersOnField.remove(player);
57              numberOfPlayersOnField--;
58          } else if (posName.toLowerCase().equals("midfielder")){
59              if (removeFromRoster){
60                  boolean b = midfielders.remove(player);
61                  if (!b) throw new tests.InvalidPlayer("The player
                        "+player.getName()+" isn't in this roster");
62              }
63              midfieldersOnField.remove(player);
64              numberOfPlayersOnField--;
65          } else if (posName.toLowerCase().equals("striker")){
66              if (removeFromRoster){
67                  boolean b = strikers.remove(player);
68                  if (!b) throw new tests.InvalidPlayer("The player
                        "+player.getName()+" isn't in this roster");
69              }
70              strikersOnField.remove(player);
71              numberOfPlayersOnField--;
72          }
73      }
74
```

```
75      // Usage: b = addPlayerToRoster(player)
76      // Before:player is of type PlayerInferface and is not null
77      // After: If there is room for the player in roster in the position that he
            plays then he's
78      //        added to the roster and b is returned as true. If there is no
            room him in his
79      //        position then b is returned as false. If the player's position is
            not "Goalkeeper",
80      //        "Defender", "Midfielder", or "Striker" then InvalidPosition
            exception is thrown.
81      public boolean addPlayerToRoster(PlayerInterface player) throws
            InvalidPosition{
82          String posName = player.getPositionName();
83          if (posName.toLowerCase().equals("goalkeeper")){
84              if (this.goalkeepers.size() == 2) return false;
85              this.goalkeepers.add(player);
86              return true;
87          } else if (posName.toLowerCase().equals("defender")){
88              if (this.defenders.size() == 5) return false;
89              this.defenders.add(player);
90              return true;
91          } else if (posName.toLowerCase().equals("midfielder")){
92              if (midfielders.size() == 5) return false;
93              this.midfielders.add(player);
94              return true;
95          } else if (posName.toLowerCase().equals("striker")){
96              if (strikers.size() == 3) return false;
97              this.strikers.add(player);
98              return true;
99          } else {
100             throw new InvalidPosition(posName+" is not a valid position. Only
                    Goalkeeper, Defender, Midfielder, and Striker are valid.");
101         }
102     }
103
104     // Usage: b = addPlayerToField(player)
105     // Before:player is of type PlayerInterface
106     // After: If the player isn't in the roster then an InvalidPlayer exception
            is thrown. Otherwise, and
107     //        if the player's position and roster on field are not full, the
            player will be added to the field
108     //        and b will be returned as true. Otherwise b will be returned as
            false.
109     public boolean addPlayerToField(PlayerInterface player) throws
            InvalidPlayer{
110         if (this.goalkeepers.contains(player)){
111             if (this.goalkeeperOnField.size() >= 1 ||
                    this.numberOfPlayersOnField >= 11){
112                 return false;
113             }
114             this.goalkeeperOnField.add(player);
115             this.numberOfPlayersOnField++;
116             return true;
```

```
117         } else if (this.defenders.contains(player)){
118             if (this.defendersOnField.contains(player) ||
                    this.numberOfPlayersOnField >= 11){
119                 return false;
120             }
121             this.defendersOnField.add(player);
122             this.numberOfPlayersOnField++;
123             return true;
124         } else if (this.midfielders.contains(player)){
125             if (this.midfieldersOnField.contains(player) ||
                    this.numberOfPlayersOnField >= 11){
126                 return false;
127             }
128             this.midfieldersOnField.add(player);
129             this.numberOfPlayersOnField++;
130             return true;
131         } else if (this.strikers.contains(player)){
132             if (this.strikersOnField.contains(player) ||
                    this.numberOfPlayersOnField >= 11){
133                 return false;
134             }
135             this.strikersOnField.add(player);
136             this.numberOfPlayersOnField++;
137             return true;
138         } else {
139             throw new tests.InvalidPlayer(player.getName()+" is currently not
                    in the roster.");
140         }
141     }
142
143     // Usage: getPlayersInRoster()
144     // Before:Nothing
145     // After: List containing a list of all players in the current roster.
            There will always be 4 inner
146     //        lists, the first for goalkeepers, second for defenders, third for
            midfielders, and the
147     //        4th for strikers.
148     public List<List<PlayerInterface>> getPlayersInRoster(){
149         List<List<PlayerInterface>> names = new
                ArrayList<List<PlayerInterface>>(4);
150         names.add(goalkeepers);
151         names.add(defenders);
152         names.add(midfielders);
153         names.add(strikers);
154         return names;
155     }
156 }
```