



---

## Group assignment 3: Tests

---

PRÓUN 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 Prastarson (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:

<i><b>Roster</b></i>
Keeps track of which football players are in which user team/roster.
<ul style="list-style-type: none"><li>- Player[2] goalkeepers</li><li>- Player[1] goalkeepersOnField</li><li>- Player[5] defenders</li><li>- Player[5] defendersOnField</li><li>- Player[5] midfielders</li><li>- Player[5] midfieldersOnField</li><li>- Player[3] strikers</li><li>- Player[3] strikersOnField</li><li>- Player captain</li><li>- int numberOfPlayersOnField</li></ul>
<ul style="list-style-type: none"><li>+ Roster()</li><li>+ void removePlayerFromTeam(Player player)</li><li>+ boolean addPlayerToTeam(Player player)</li><li>+ void changeCaptain(Player newCaptain)</li><li>+ boolean putPlayerOnField(Player player)</li><li>+ boolean removePlayerFromField(Player player)</li></ul>

**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 as 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,
10                                List<List<PlayerInterface>> actual) throws IllegalStateException{
11     // Count the number of matches
12     int matches = 0;
13     // If the two lists (of lists) have different number of elements, throw
14     // exception.
15     if (actual.size() != expected.size()){
16         throw new IllegalStateException("Sizes of lists containing lists not
17             the same: "+expected.size()+" and "+actual.size());
18     }
19     // Create an iterator for both lists (of lists)
20     Iterator<List<PlayerInterface>> playerlist_iterator = actual.iterator();
21     Iterator<List<String>> expected_playerlist_iterator = expected.iterator();
22     // Loop through the outer lists
23     while(playerlist_iterator.hasNext()){
24         List<PlayerInterface> playerlist = playerlist_iterator.next();
25         List<String> expected_playerlist = expected_playerlist_iterator.next();
26
27         if (playerlist.size() != expected_playerlist.size()){
28             throw new IllegalStateException("Sizes of lists not the same:
29                 "+expected_playerlist.size()+" and "+playerlist.size());
30         }
31
32         Iterator<PlayerInterface> player_iterator = playerlist.iterator();
33         Iterator<String> expected_player_iterator =
34             expected_playerlist.iterator();
35
36         // Loop through the inner lists
37         while(player_iterator.hasNext()){
38             String expected_player = expected_player_iterator.next();
39             if(player_iterator.next().getName() != expected_player){
40                 return -1;
41             }
42             else{
43                 matches++;
44             }
45         }
46     }
47     return matches;
48 }
```

### 2.1.3 Testing the addPlayerToRoster method

#### Test 1: List, art thou empty?

This test will check if a new roster is empty.

```
1 public void testIfEmpty() throws IllegalStateException, InvalidPosition {
2     List<List<PlayerInterface>> actual = roster.getPlayersInRoster();
3     List<List<String>> expected = new ArrayList<List<String>>(4)
4         {{add(goalkeepers);add(defenders);add(midfielders);add(strikers);}};
5     assertEquals(0,compareListsOfLists(expected, actual));
6 }
```

#### Test 2: Takes one to know one

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

```
1 public void testIfOnePlayer() throws IllegalStateException, InvalidPosition {
2     // Add the player "Goalkeeper 1" to the roster
3     boolean add = roster.addPlayerToRoster(players.get("Goalkeeper 1"));
4     assertTrue(add);
5
6     // Get the roster players
7     List<List<PlayerInterface>> actual = roster.getPlayersInRoster();
8
9     // Create the expected outcome of the test
10    goalkeepers.add("Goalkeeper 1");
11    List<List<String>> expected = new ArrayList<List<String>>(4)
12        {{add(goalkeepers);add(defenders);add(midfielders);add(strikers);}};
13
14    assertEquals(1,compareListsOfLists(expected, actual));
15 }
```

#### 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.

```
1 public void testIfInvalidPlayer() throws InvalidPosition {
2     Throwable exception = null;
3     // Add the player "Football fan" to the roster
4     try{
5         roster.addPlayerToRoster(players.get("Football fan"));
6     } catch (Throwable e) {
7         exception = e;
8     }
9     assertNotNull(exception);
10    assertEquals(InvalidPosition.class,exception.getClass());
11 }
```

#### 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.

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

```

3 roster.addPlayerToRoster(players.get("Goalkeeper 1"));
4 boolean add = roster.addPlayerToRoster(players.get("Goalkeeper 2"));
5 assertTrue(add);
6 add = roster.addPlayerToRoster(players.get("Goalkeeper 3"));
7 assertFalse(add);
8
9 // Get the roster players
10 List<List<PlayerInterface>> actual = roster.getPlayersInRoster();
11
12 // Create the expected outcome of the test
13 goalkeepers.add("Goalkeeper 1");
14 goalkeepers.add("Goalkeeper 2");
15 List<List<String>> expected = new ArrayList<List<String>>(4)
16     {{add(goalkeepers);add(defenders);add(midfielders);add(strikers);}};
17
18 assertEquals(2,compareListsOfLists(expected, actual));
19 }

```

#### Test 5: No more room in heaven

```

1 public void testIfFullRoster() throws InvalidPosition {
2     // Add the player "Goalkeeper 1" to the roster
3     roster.addPlayerToRoster(players.get("Goalkeeper 1"));
4     roster.addPlayerToRoster(players.get("Goalkeeper 2"));
5     boolean add;
6     add = roster.addPlayerToRoster(players.get("Defender 1")); assertTrue(add);
7     add = roster.addPlayerToRoster(players.get("Defender 2")); assertTrue(add);
8     add = roster.addPlayerToRoster(players.get("Defender 3")); assertTrue(add);
9     add = roster.addPlayerToRoster(players.get("Defender 4")); assertTrue(add);
10    add = roster.addPlayerToRoster(players.get("Defender 5")); assertTrue(add);
11    add = roster.addPlayerToRoster(players.get("Midfielder 1"));
12        assertTrue(add);
13    add = roster.addPlayerToRoster(players.get("Midfielder 2"));
14        assertTrue(add);
15    add = roster.addPlayerToRoster(players.get("Midfielder 3"));
16        assertTrue(add);
17    add = roster.addPlayerToRoster(players.get("Midfielder 4"));
18        assertTrue(add);
19    add = roster.addPlayerToRoster(players.get("Midfielder 5"));
20        assertTrue(add);
21    add = roster.addPlayerToRoster(players.get("Striker 1"));
22        assertTrue(add);
23    add = roster.addPlayerToRoster(players.get("Striker 2"));
24        assertTrue(add);
25    add = roster.addPlayerToRoster(players.get("Striker 3"));
26        assertTrue(add);
27
28    // Get the roster players
29    List<List<PlayerInterface>> actual = roster.getPlayersInRoster();
30
31    // Create the expected outcome of the test
32    goalkeepers.add("Goalkeeper 1");
33    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>> expected = new ArrayList<List<String>>(4)
        {{add(goalkeepers);add(defenders);add(midfielders);add(strikers);}};
40
41     assertEquals(15,compareListsOfLists(expected, 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 assertEquals(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>> expected = new ArrayList<List<String>>(4)
        {{add(goalkeepers);add(defenders);add(midfielders);add(strikers);}};
8     assertEquals(0, compareListsOfLists(expected, 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{

```

```

4         roster.removePlayer(players.get("Goalkeeper 1"), true);
5     } catch (Throwable e) {
6         exception = e;
7     }
8     assertNotNull(exception);
9     assertSame(InvalidPlayer.class, exception.getClass());
10 }

```

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

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

```

1 public void testNumberOfPlayersOnField() throws InvalidPosition, InvalidPlayer {
2     assertEquals(0, roster.getNumberOfPlayersOnField());
3     roster.addPlayerToRoster(players.get("Goalkeeper 1"));
4     roster.addPlayerToRoster(players.get("Defender 2"));
5     assertEquals(0, roster.getNumberOfPlayersOnField());
6     roster.addPlayerToField(players.get("Goalkeeper 1"));
7     assertEquals(1, roster.getNumberOfPlayersOnField());
8     roster.addPlayerToField(players.get("Defender 2"));
9     assertEquals(2, roster.getNumberOfPlayersOnField());
10    roster.removePlayer(players.get("Goalkeeper 1"), false);
11    assertEquals(1, roster.getNumberOfPlayersOnField());
12    roster.removePlayer(players.get("Defender 2"), true);
13    assertEquals(0, roster.getNumberOfPlayersOnField());
14 }

```

## 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.

```

1 @BeforeClass
2 public static void setUp() throws Exception {
3     user = new User("user1");
4 }

```

```

1 @AfterClass
2 public static void tearDown() throws Exception {
3     user = null;
4 }

```

### 2.2.2 Testing the user classes

#### Test 1: Remember kids, null != empty

This test checks if the user has a Roster.

```

1 @Test
2 public void checkRoster() {
3     assertNotNull(user.getRoster());
4 }

```



## Test 2: Mannanafnanefnd

This test checks if a name change goes through.

```
1 @Test
2 public void nameChange() {
3     user.changeName("user2");
4     assertEquals("user2", user.getName());
5 }
```

## Test 3: Make sure to put it in the bank

```
1 @Test
2 public void testChangingMoney() {
3
4     // Money is 0
5     assertTrue(user.changeMoney(1000));
6     assertEquals(1000, user.getMoney());
7
8     // Money is 1000
9     int curr = user.getMoney();
10    assertTrue(user.changeMoney(-900));
11    assertEquals(curr-900, user.getMoney());
12
13    // Money is 100
14    curr = user.getMoney();
15    assertFalse(user.changeMoney(-200));
16    assertEquals(curr, user.getMoney());
17    // Money is still 100 cause you can't get negative money
18 }
```

### 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 Fla.

```
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") ||
24            pos.equals("Midfielder") || pos.equals("Striker")){
25            this.position = pos;
26        } else {
27            throw new InvalidPosition(pos+" is not a valid position. Only
28                Goalkeeper, Defender, Midfielder, and Striker are valid.");
29        }
30    }
31
32    @Override
33    public PositionMock getPosition() {
34        return position;
35    }
36 }
```

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 substring
Rooney	null	Striker	Wayne Rooney, Adam Rooney,...	All strikers having the Rooney substring
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

```
1 @Before
2 public void setUp() throws Exception {
3     roster = new Roster();
4     goalkeepers = new ArrayList<String>(2);
5     defenders = new ArrayList<String>(5);
6     midfielders = new ArrayList<String>(5);
7     strikers = new ArrayList<String>(3);
8 }
```

### After each test method

```
1 @After
2 public void tearDown() throws Exception {
3     roster = null;
4     goalkeepers = null;
5     defenders = null;
6     midfielders = null;
7     strikers = null;
8 }
```

### Roster.java

```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 import tests.*;
5
6 public class Roster {
7     private List<PlayerInterface> goalkeepers;
8     private List<PlayerInterface> goalkeeperOnField;
9     private List<PlayerInterface> defenders;
10    private List<PlayerInterface> defendersOnField;
11    private List<PlayerInterface> midfielders;
12    private List<PlayerInterface> midfieldersOnField;
13    private List<PlayerInterface> strikers;
14    private List<PlayerInterface> strikersOnField;
15    // private PlayerInterface captain;
16    private int numberOfPlayersOnField;
17
18    public Roster(){
19        this.numberOfPlayersOnField = 0;
20        this.goalkeepers = new ArrayList<PlayerInterface>(2);
21        this.goalkeeperOnField = new ArrayList<PlayerInterface>(1);
22        this.defenders = new ArrayList<PlayerInterface>(5);
23        this.defendersOnField = new ArrayList<PlayerInterface>(5);
24        this.midfielders = new ArrayList<PlayerInterface>(5);
25        this.midfieldersOnField = new ArrayList<PlayerInterface>(5);
26        this.strikers = new ArrayList<PlayerInterface>(3);
27        this.strikersOnField = new ArrayList<PlayerInterface>(3);
28    }
29
30    // 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
39 //         (true or false)
40 // After: If b is true then player will be removed both from the field and
41 //         the roster. If
42 //         b is false then the player will only be removed from the field.
43 //         If the player
44 //         provided is not in the roster then a InvalidPlayer exception will
45 //         be thrown.
46 public void removePlayer(PlayerInterface player, boolean removeFromRoster)
47     throws InvalidPlayer{
48     String posName = player.getPositionName();
49     if (posName.toLowerCase().equals("goalkeeper")){
50         if (removeFromRoster){
51             boolean b = goalkeepers.remove(player);
52             if (!b) throw new tests.InvalidPlayer("The player
53                 "+player.getName()+" isn't in this roster");
54         }
55         goalkeeperOnField.remove(player);
56         numberOfPlayersOnField--;
57     } else if (posName.toLowerCase().equals("defender")){
58         if (removeFromRoster){
59             boolean b = defenders.remove(player);
60             if (!b) throw new tests.InvalidPlayer("The player
61                 "+player.getName()+" isn't in this roster");
62         }
63         defendersOnField.remove(player);
64         numberOfPlayersOnField--;
65     } else if (posName.toLowerCase().equals("midfielder")){
66         if (removeFromRoster){
67             boolean b = midfielders.remove(player);
68             if (!b) throw new tests.InvalidPlayer("The player
69                 "+player.getName()+" isn't in this roster");
70         }
71         midfieldersOnField.remove(player);
72         numberOfPlayersOnField--;
73     }
74 }

```

```

75 // Usage: b = addPlayerToRoster(player)
76 // Before: player is of type PlayerInterface 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 (this.midfielders.size() == 5) return false;
93         this.midfielders.add(player);
94         return true;
95     } else if (posName.toLowerCase().equals("striker")){
96         if (this.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) ||
119             this.numberOfPlayersOnField >= 11){
120             return false;
121         }
122         this.defendersOnField.add(player);
123         this.numberOfPlayersOnField++;
124         return true;
125     } else if (this.midfielders.contains(player)){
126         if (this.midfieldersOnField.contains(player) ||
127             this.numberOfPlayersOnField >= 11){
128             return false;
129         }
130         this.midfieldersOnField.add(player);
131         this.numberOfPlayersOnField++;
132         return true;
133     } else if (this.strikers.contains(player)){
134         if (this.strikersOnField.contains(player) ||
135             this.numberOfPlayersOnField >= 11){
136             return false;
137         }
138         this.strikersOnField.add(player);
139         this.numberOfPlayersOnField++;
140         return true;
141     } else {
142         throw new tests.InvalidPlayer(player.getName()+" is currently not
143             in the roster.");
144     }
145 }
146
147 // Usage: getPlayersInRoster()
148 // Before: Nothing
149 // After: List containing a list of all players in the current roster.
150 //         There will always be 4 inner
151 //         lists, the first for goalkeepers, second for defenders, third for
152 //         midfielders, and the
153 //         4th for strikers.
154 public List<List<PlayerInterface>> getPlayersInRoster(){
155     List<List<PlayerInterface>> names = new
156         ArrayList<List<PlayerInterface>>(4);
157     names.add(goalkeepers);
158     names.add(defenders);
159     names.add(midfielders);
160     names.add(strikers);
161     return names;
162 }
163 }

```



## User.java

```
1 public class User {
2     private int money;
3     private int[] score;
4     private String name;
5     private Roster roster;
6
7     public User(String name) {
8         this.name = name;
9         this.score = new int[10];
10        this.roster = new Roster();
11        this.money = 0;
12    }
13
14    public boolean changeMoney(int dMoney) {
15        if(this.money + dMoney < 0) return false;
16        this.money += dMoney;
17        return true;
18    }
19
20    public int getMoney(){
21        return this.money;
22    }
23
24    public Roster getRoster() {
25        return this.roster;
26    }
27
28    // Usage: i = getScore()
29    // Before: Nothing.
30    // After: i is an array of scores of this user
31    public int[] getScore() {
32        return this.score;
33    }
34
35    public void addScore(int round, int score) {
36        this.score[round] += score;
37        addScoreToStats();
38    }
39
40    public void addScoreToStats() {
41
42    }
43
44    public void changeName(String newname) {
45        this.name = newname;
46    }
47
48    public String getName() {
49        return name;
50    }
51 }
```

## UserTest.java

```
1 import static org.junit.Assert.*;
2
3 import org.junit.AfterClass;
4 import org.junit.BeforeClass;
5 import org.junit.Test;
6
7 import backend.User;
8
9 public class UserTest {
10
11     private static User user;
12
13     // This test, tests the class that holds onto the users of the game.
14     // We setup the test with one new user which takes in "user1" as its name.
15
16     @BeforeClass
17     public static void setUp() throws Exception {
18         user = new User("user1");
19     }
20
21     @AfterClass
22     public static void tearDown() throws Exception {
23         user = null;
24     }
25
26     // This test checks if the user has a Roster
27     @Test
28     public void checkRoster() {
29         assertNotNull(user.getRoster());
30     }
31
32     // This test checks if a name change goes though
33     @Test
34     public void nameChange() {
35         user.changeName("user2");
36         assertEquals("user2", user.getName());
37     }
38
39     // This test checks if the score from round 0 goes to index 0
40     @Test
41     public void scoreTest() {
42         user.addScore(0, 100);
43         assertEquals(100, user.getScore()[0]);
44     }
45
46     // This test checks if the changes to the money of the user behave as
47     // intended
48     // where the user can't have negative money and no money is subtracted if
49     // that subtraction
50     // would set the users money below zero
51     @Test
```

```
50 public void testChangingMoney() {
51
52     // Money is 0
53     assertTrue(user.changeMoney(1000));
54     assertEquals(1000, user.getMoney());
55
56     // Money is 1000
57     int curr = user.getMoney();
58     assertTrue(user.changeMoney(-900));
59     assertEquals(curr-900, user.getMoney());
60
61     // Money is 100
62     curr = user.getMoney();
63     assertFalse(user.changeMoney(-200));
64     assertEquals(curr, user.getMoney());
65     // Money is still 100 cause you can't get negative money
66 }
67 }
```