Test First Development

From Extreme Programming "*When you create your tests first, before the code, you will find it much easier and faster to create your code. The combined time it takes to create a unit test and create some code to make it pass is about the same as just coding it up straight away. But, if you already have the unit tests you don't need to create them after the code saving you some time now and lots later*."

Test-driven development (TDD) is an evolutionary approach to development which combines test-first development where you write a test before you write just enough production code to fulfil that test and refactoring.

One school of thought considers the goal of TDD to be specification and not validation, where a developer must think through the requirements or design before writing functioning code. Others think of TDD as a programming technique to write clean code that works.

The steps in test first development are

- Start with a failing test
- Make sure that the error message is informative enough
- Fix the code
- After the test passes refactor the code

Exercise

Implement a FootballTeam class, so we can compare different teams and see who takes first place in the league. Each team keeps a record of the number of games won.
Based on the example in Chapter 4 of Practical Unit Testing with JUnit and Mockito.

In order to compare two teams, each of them has to remember its number of wins. For the sake of simplicity let us design a FootballTeam class that takes the number of games as a constructor parameter.

First things first: let us make sure that this constructor works.

```
public class FootballTeamTest {
    @Test
    public void constructorShouldSetGamesWon() {
        FootballTeam team = new FootballTeam(3);
        assertEquals(3, team.getGamesWon());
    }
}
```

Obviously, you need to create a FootballTeam class and its getGamesWon() method before proceeding any further. You can let your IDE create the class, its constructor and this one method for you, or you can write them yourself.

There are two things to remember when writing code that is necessary in order for a test to compile:

- All production code should be kept in a different directory from the tests (IntelliJ automatically does this if you select add Test Class)
- Do nothing more than the minimum required for the test to compile. Create the necessary classes and methods, but do not fit them out with any logic

It does not matter whether we created the required code ourselves or let IDE do it we will end up with an implementation of the FootballTeam class along similar lines to the following:

```
public class FootballTeam {
    public FootballTeam(int gamesWon) {
    }
    public int getGamesWon() {
        return 0;
    }
}
```

Resist the urge to fix the code right away. Instead, look at the error message. Does it say precisely what is wrong here? If the answer is "no", then add a custom error message. If you are happy with the current message, then proceed further. In JUnit 5 the optional error message is the final argument. An example custom error message is:

```
assertEquals(3, team.getGamesWon(),
            "3 games passed to constructor, but "
            + team.getGamesWon +  " were returned");
```

To fix the code is straightforward this time: all we need to do is store the value passed as the constructor parameter to some internal variable. The fixed FootballTeam class is presented below.

```
public class FootballTeam {
    private int gamesWon;
    public FootballTeam(int gamesWon) {
        this.gamesWon = gamesWon;
    }
    public int getGamesWon() {
        return gamesWon;
    }
}
```

There is no meaningful refactoring to be done. However, do not forget to refactor the test. The least we should do is to get rid of the hardcoded number 3 – for example, by introducing a `THREE_GAMES_WON` variable

```
public class FootballTeamTest {
    private static final int THREE_GAMES_WON = 3;
    @Test
    public void constructorShouldSetGamesWon() {
        FootballTeam team = new FootballTeam(THREE_GAMES_WON);
        assertEquals(THREE_GAMES_WON, team.getGamesWon(),
                     THREE_GAMES_WON + " games passed to
                     constructor, but "+ team.getGamesWon() + "
                     were returned",);
     }
}
```

The constructor appears to work for one value! We should add tests for more than one value. We will use a parameterised test as follows

```
@ParameterizedTest
@ValueSource(ints = {1, 2, 3})
public void constructorShouldSetGamesWon(int legalNumberOfGames) {
   FootballTeam team = new FootballTeam(legalNumberOfGames);
   assertEquals(legalNumberOfGames, team.getGamesWon());
}
```

To test if the constructor throws an exception if an invalid number of wins is used as an argument to the constructor

```
public void constructorShouldSThrowIAEForIllegalNumberOfGames() {
   assertThrows(IllegalArgumentException.class,
                () -> {
FootballTeam team = new FootballTeam(ILLEGAL_NUMBER_OF_WINS);
                    });
}
```

The constructor works fine. Now we can move on to the main problem: that is, to the comparing of football teams. First of all, we have decided that we are going to use the Comparable interface. This is an important decision which will influence not only the implementation of this class but also its API and expected behaviour. If FootballTeam is comparable, then the client can expect that once there are a few teams in a collection, they could be sorted using Collections.sort(). If so, then there should be a test for this behaviour.

```
private static final int ANY_NUMBER = 123;
@Test
public void shouldBePossibleToCompareTeams() {
    FootballTeam team = new FootballTeam(ANY_NUMBER);
    assertTrue(team instanceof Comparable,
               "FootballTeam should implement Comparable");
}
```

The test will fail, but the IDE is capable of automatically fixing the code, generating the default implementation of the compareTo() method, to make the FootballTeam class look like this

```
public class FootballTeam implements Comparable<FootballTeam> {
    private int gamesWon;

    public FootballTeam(int gamesWon) {
        this.gamesWon = gamesWon;
    }
    public int getGamesWon() {
        return gamesWon;
    }
    @Override
    public int compareTo(FootballTeam o) {
          return 0;
    }
}
```

Comparison tests - fail

```
@Test
public void teamsWithMoreMatchesWonShouldBeGreater() {
    FootballTeam team_2 = new FootballTeam(2);
    FootballTeam team_3 = new FootballTeam(3);
    assertTrue(team_3.compareTo(team_2) > 0);
}

@Override
public int compareTo(FootballTeam o) {
    if (gamesWon > o.getGamesWon()) {
        return 1;
     }
     return 0;
}
```

After running all tests, we should:

- Refactor (e.g. change the o variable to otherTeam)
- Rerun the tests so we are sure nothing is broken
- Proceed with the next test

```
@Test
public void teamsWithLessMatchesWonShouldBeLesser() {
    FootballTeam team_2 = new FootballTeam(2);
    FootballTeam team_3 = new FootballTeam(3);
    assertTrue(team_2.compareTo(team_3) < 0,
              "team with " + team_2.getGamesWon()
              + " games won should be ranked after the team with "
              + team_3.getGamesWon() + " games won",);
}
```

Run, see it fail, then introduce changes to the FootballTeam class so the tests pass. The implementation which makes this test pass is, once again, straightforward to code

```
public int compareTo(FootballTeam otherTeam) {
    if (gamesWon > otherTeam.getGamesWon()) {
          return 1;
    }
    else if (gamesWon < otherTeam.getGamesWon()) {
        return -1;
    }
    return 0;
}
```

```
@Test
public void teamsWithSameNumberOfMatchesWonShouldBeEqual() {
    FootballTeam teamA = new FootballTeam(2);
    FootballTeam teamB = new FootballTeam(2);
    assertTrue(teamA.compareTo(teamB) == 0,
               "both teams have won the same number of games: "
             + teamA.getGamesWon() + " vs. " + teamB.getGamesWon()
             + " and should be ranked equal");
}
```

Well, this test passes instantly, because our implementation has already returned 0 in cases of equality. So, what should we do now? We have definitely skipped one step in the TDD rhythm. We have never seen this equality test failing, so we do not know why it passes. Is it because the FootballTeam class really implements the expected behaviour, or is it because our test has been badly written and would always pass?

Now that we have a safety net of tests we can could refactor the tested method. After having thought through the matter carefully, we have ended up with much simpler implementation:

```
public int compareTo(FootballTeam otherTeam) {
    return gamesWon - otherTeam.getGamesWon();
}
```

Tests That Pass By Default

If the test fails, we will be 100% sure it was executed. But if all we see is a green colour, then we have to be certain about whether it was executed at all. In cases of larger test suits it might be easy to overlook a test that was not run. So at least make sure your test was actually executed.

If a test happens to pass we will never see it fail. For example, the tests of the compareTo() method that we created for the FootballTeam class, the default return value (we chose or the IDE inserted) was 0. Given that, if we were to compare two football teams with equal numbers of wins, such a test would pass instantly. In these cases, the recommendation is break the code so you see the test fail.

```
public int compareTo(FootballTeam otherTeam) {
    if (gamesWon > otherTeam.getGamesWon()) {
        return 1; }
    else if (gamesWon < otherTeam.getGamesWon()) {
        return -1;
    }
    return 18723;
}
```

Now rerun the test, and see it fails Good. Revert the change of compareTo() method (so it returns 0 again). Rerun the test to see it pass. Good. This means that your test really verifies this behaviour.