Unit-Testing Essentials

Leonhard Applis

September 29, 2019

What is a Unit-Test?

Definition

A Unit Test ...

1. tests a single piece of functionality
2. is written by the developer
3. should not depend on any other, external resource
4. should be run with every change of the code

Unit-Test Examples

These are Unit Tests ...

- testing the distance-function of a point to another point
- testing whether the sort() function of your list sorts the list
- testing whether a constructor fails for null-values

These are **not** Unit Tests...

- read and write to an database
- sending an http-request to an api
- checking if an config-file is successfully read

XUnit

- A common way of writing unit tests
- NUnit, JUnit, HUnit, ...
- tests are written as functions, which have an assert statement
- if the assert succeeds, the test is green
- if the assert fails, the test is red and the difference is shown
- as functions, tests can be debugged

Example 1: Point Distance Test

```
public class distanceTests{
    @Test
    public void testDistance_pointIs100Away_shouldBe100(){
        //Arrange
        Point a = new Point(0,0);
        Point b = new Point(100,0);
        //Act
        double distance = a.distanceTo(b);
        //Assert
        assertEquals(100,distance);
    }

    @Test
    public void testDistance_distanceToItself_shouldBe0(){
        //Arrange
        Point a = new Point(75,6);
        //Act
        double distance = a.distanceTo(a);
        //Assert
        assertEquals(0.0,distance);
    }
}
```

Example 1: Point Distance Code

```
public class Point{
    public double x,y;

    public Point(double x, double y){
        this.x=x;
        this.y=y;
    }

    public double distanceTo(Point other){
        double xdif = x - other.x;
        double ydif = y - other.y;
        double intermediate = xdif * xdif + ydif *
        return Math.sqrt( intermediate  );
    }
}
```

JUnit markup words

- *@Test* marks that the following function is a Test
- *assertEquals(expected,actual)* is used to check values - warning: for objects the *equals()* method is used
- *assertFalse(...),assertTrue(...),assertThrowsException(...)* are shortcuts to write smaller and more expressive tests
- *@Tag "xy"* gives the test a filterable tag, e.g. you could run only *player-related* tests

Best Practice - No Global Dependencies

Do not use (static) global variables in your tests if it's not absolutely necessary!
Reasons:

- Tests may fail because of their order
- if the dependency changes, every tests needs to change
- functions like *resetDB()* after every tests maybe fail because of parallel execution
- deactivating parallel-execution is bad
- the items you need are not written in your tests, but have to be looked up

Mitigation: **create the dependency in every test**. If you have a common pattern, you may want to use a *factory method* which returns a new, default instance of your dependency.

One 'Test' per Test

Only do *one Assert per test!*
If you are doing multiple asserts, you usually also have multiple tests.
If you try to assert two things in one statement, you are not sure if the later one would be right if the first one failed.
negative: A lot more tests, A lot more code in general
positive: Clearer tests, tests are more maintainable,
more test mean more green tests, which makes you happy.

Arrange - Act - Assert

Write your tests with a common pattern:

1. **Arrange:** Create every item required for the test
2. **Act:** Perform every Action required for the test
3. **Assert:** Perform the Check(s)

Split those points by a single line, and you'll have a nice, common pattern.
Don't do *Arrange-Act-Rearrange-Act-Assert!*
Never do *Arrange-Act-Assert-Act-Assert!*

Working with dependent Objects

For Inter-Object communication there are usually 2 cases:

1. To do something, I need another object
2. My actions change another object

No matter if these items are global (variables), static (System Args) or external (such as databases)

Stubs

A stub is a testclass which implements a given interface, which can be used as *fakeInput* for another test.

A stub is *input only*, so the stub will not be changed after generation. **A stub provides behaviour**.
e.g: a function *addPlayer(InterfacePlayer)* from the class *(*team) may use a stubPlayer

primary reason for stubs: stubs act the way you want, without having any logic. The tests of *team* would maybe fail if the *RealPlayer* would have failures.

Example 2: StubPlayer

```
public interface Player {
    void addToTeam(Team t);
}

public class StubPlayer implements Player{

    public void addToTeam(Team t) {
        // Do nothing
    }

}
```

Example 2: SimpleTeam

```
public class SimpleTeam implements Team {
    private List<Player> players =
        new LinkedList<>();

    public List<Player> getPlayers() {
        return  players;
    }

    public void addPlayer(Player p){
        players.add(p);
    }
}
```

Example 2: Team-Tests

```
public class SimpleTeamTest{
    @Test
    void addPlayer_WithstubPlayer_ShouldHaveOnePlayer(){
        //Arrange
        Team testObject = new SimpleTeam();
        Player stub = new StubPlayer();
        //Act
        testObject.addPlayer(stub);
        //Assert
        assertEquals(1,testObject.getPlayers().size());
    }
    @Test
    void addPlayer_WithStubPlayer_shouldContainStubPlayer(){
        //Arrange
        Team testObject = new SimpleTeam();
        Player stub = new StubPlayer();
        //Act
        testObject.addPlayer(stub);
        //Assert
        assertTrue(testObject.getPlayers().contains(stub));
    }
}
```

Stubs: Why Interfaces are your friends

To do a real nice set of stubs, you need interfaces. e.g. *Player*.

Interfaces *clean your code*, analyze and generalize behavior.
You can tell what a class should be doing by the implemented interfaces (atleast with good names used).
In object oriented languages (such as C-Sharp and Java) it is considered best practice to only use interfaces as inputs (especially in interfaces) which is called *programming against interfaces*.

Mocks

A mock is somewhat the opposite of a stub: The stub provides behavior, the mock *takes* behavior.

A mock is considered *write only*, except for the asserts.

Example: The function *addToTeam(Team)* of the class *Player* would need a *MockTeam*. The mock team *safes* that the class was touched in the expected way.

warning: in some languages, such as JS, everything is called a mock.

Example 3: MockTeam

```
public interface Team{
    List<Player> getPlayers();

    void addPlayer(Player p);
}

public class MockTeam implements Team{

    public List<Player> fakePlayerList =
        new LinkedList<>();
    public boolean invokedGetPlayers=false;

    public List<Player> getPlayers(){
        invokedGetPlayers=true;
        return fakePlayerList;
    }

}
```

Example 3: SimplePlayer

```
public class SimplePlayer implements Player{
    String name;
    public SimplePlayer(String name){
        this.name=name;
    }
    public void addToTeam(Team){
        Team.getPlayers().add(this);
    }
}
```

Example 3: Team-Tests

```
public class SimplePlayerTest{
    @Test
    void addToTeam_withMockTeam_MockTeamShouldHaveInvokedGetPlayer
        //Arrange
        Player testObject = new SimplePlayer("Diego");
        MockTeam mock = new MockTeam();
        //Act
        testObject.addToTeam(mock);
        //Assert
        assertTrue(mock.invokedGetPlayers);
    }

    @Test
    void addToTeam_withMockTeam_MockTeamShouldContainPlayer(){
        //Arrange
        Player testObject = new SimplePlayer("Diego");
        MockTeam mock = new MockTeam();
        //Act
        testObject.addToTeam(mock);
        //Assert
        assertTrue(mock.fakePlayerList.contains(testObject));
    }
}
```

Stubs vs. Mocks vs. Fakes

- **A Stub** provides behavior and is *read only*.
- **A Mock** accepts behavior and actions and is **write only**
- **A Fake** is a combination of a stub and a mock.

You may consider calling everything a fake, as the term *Mock* is overloaded (and misused) in some languages and frameworks, while the term stub is unknown to a lot of people.

Naming - Good vs. Bad

Names of the tests are very important for readability of reports. example bad-names: *testCalculate*, *testCalculateFails*, *testCollide*, *testCollide2 ...*
the name of the test should include:

1. the function under test
2. a summary of the setup
3. the expected result

simple test-name-pattern for beginners:
testFunction-WithBadInput-shouldThrowError
testConstructor-withValidAttributes-shouldBeBuild

Test First

The primary idea of *test first* is to write unit-tests for the expected behavior without writing any line of the *real code*.
Normal workflow: *write code -> extract interface -> write tests*.
Test First worklow: *write interface -> write tests -> write code*

benefits: The requirements and behavior must be defined beforehand, the code is properly tested, the code is using interfaces.
downsides: If the class is complex, the design and testing may take very long.
The tests may be red for a long time if the code is complex.

Test-Driven-Development

RED-GREEN-REFACTOR

TDD is a specified subclass of *test-first*.

1. pick an item from the todo list
2. write a non-passing test
3. write code to pass the test
4. clean up the class until it looks nice, but keep the tests as-is
5. repeat

Where you should never have more than one red test!

downsides: refactoring is sometimes left out, not appliable for complex problems - only for *commodity code*.

TDD-Schools

1. **London-School:** *Mock Everything*
   Mocking forces nice interfaces and code that uses nice interfaces.
2. **Chicago-School:** *Mock Nothing*
   Mocking bloats the code, and makes sometimes no sense.
3. **Munich-School:***Mock what makes sense*
   dont focus on a school - do what goes best and respect the language