Test-Driven Development (TDD)
Some basic info

Torch Talk, 3 Nov 11

Marc Schröder, DFKI
Automated tests

- Idea of self-testing software
  - manual tests:
    - if I set up my system in this configuration,
    - and exercise it in this particular way,
    - is the result as I would expect?
  - automated tests:
    - set up a System Under Test (SUT) automatically
    - exercise the SUT automatically
    - verify automatically that the result matches the expectations
De-facto standard: jUnit

- invented by Kent Beck and Erich Gamma
  - ~12 years ago
    - first for Java, now for very many languages: xUnit
    - each “unit test” tests one “unit”

- Best practice
  - write many tests, one per tested functionality
  - if a test fails, it should inform us what went wrong
Benefits of automated tests

- Once written, automated tests can be run without manual effort
  - again, and again, and again
- Two main benefits
  - “executable specification”
  - “safety net”
Tests encode how the system should function

- when we work on the code, the tests help us verify that we don't break anything
  - when we add functionality
  - when we refactor to clean up our code
Tests as executable specification

- Specification documents tend to collect dust
  - you won't notice when they get outdated
- Tests encode how the system should function
  - they specify expected behaviour
  - tests of the “interface” encode expected interaction
  - if they get out of date, we will notice next time we run the tests (i.e. very soon)
Four-phase test

- Set up System Under Test (SUT)
- Exercise SUT
- Verify result
- (Teardown SUT, to avoid interactions between tests)
JUnit

public void testCanPlay() {
    MediaPlayer player = createMediaPlayer(); // setup
    player.start(); // exercise
    assertTrue(player.isPlaying()); // verify
    player.release(); // teardown
}
Django web framework (python):

class SearchTest(django.test.TestCase):

    def testBrowse(self):
        client = Client()  # setup
        response = client.get('/{0}browse/'.format(BASE))  # exercise
        self.assertEqual('repository/edit.html',
            response.templates[0].name)  # verify
TDD: Test-Driven Development

- Idea to write the test first
  - forces you to think through the interface before fleshing out the functionality
  - you write the API as you would like to use it

- Rhythm to TDD:
  - Red
  - Green
  - Refactor
TDD Rhythm: Red

- Write the test, which will not compile
- Add the minimal possible code required to compile the test
- Run the test: it will fail, giving you a red bar in the test runner GUI.
TDD Rhythm: Green

- Now add the minimum code needed to get the test passed. Commit sins as necessary, but be quick.
- Run the test again, see the green bar: test passes.
- So now we're done, quick and dirty, right?
  - NOOO!
Once your test passes, refactor it in order to convert ugly code into beautiful code.

- Most of all, DRY: Don't repeat yourself. Remove any duplication!
- Apply other refactoring techniques:
  - extract code into specialised methods, for reuse and readability;
  - extract a joint superclass;
  - create subclasses to avoid if-else chains or switch() blocks;
  - ...
- Run all tests again, your safety net: they still pass, everything's green.
In TDD, write the four-phase test backward to get the most elegant API.

- Example: round-trip XML import-export
In TDD, write the four-phase test backward to get the most elegant API.

- Example: round-trip XML import-export

```java
public void testExportIsEquivalent() {
    // verify
    assertTrue(inputXML.isFunctionallyEquivalentTo(outputXML));
}
```
In TDD, write the four-phase test backward to get the most elegant API.

- Example: round-trip XML import-export

```java
public void testExportIsEquivalent() {
    // exercise
    ObjectModel objectModel = ObjectModel.createFromXML(inputXML);
    XMLDoc outputXML = objectModel.exportToXML();
    // verify
    assertTrue(inputXML.isFunctionallyEquivalentTo(outputXML));
}
```
In TDD, write the four-phase test backward to get the most elegant API.

- Example: round-trip XML import-export

```java
public void testExportIsEquivalent() {
    // setup
    XMLDoc inputXML = createTestXML();
    // exercise
    ObjectModel objectModel = ObjectModel.createFromXML(inputXML);
    XMLDoc outputXML = objectModel.exportToXML();
    // verify
    assertTrue(inputXML.isFunctionallyEquivalentTo(outputXML));
}
```
Summary

- It takes a bit of getting used to
- the rhythm is different
- little rewards are frequent
- trust in your code grows

- ... try it out! :-)

18