Test-Driven Development

Hui Chen [a]

[a]CUNY Brooklyn College, Brooklyn, NY, USA

Outline

Outline

Test-Driven Development (TDD)

Basic idea: write the tests before you write the code

- The tests should fail at the start
- As you complete the implementation the tests should succeed
- Testing (and the good test coverage provided by TDD) allows your team to be more flexible
- Confident in making changes to the system, when your tests pass

TDD Rule of Thumb

Another way of expressing the main rule:

- Only production code you write is for the purpose of fixing a failing test

What is production code?

Outline

TDD High Level Work Flow

RED-GREEN-REFACTORING

1. Red: Write some tests – Think about one thing the code should do; capture that thought in a test
2. Green: Write the simplest possible code that lets the test pass – Aim for "always have working code"
3. Refactor: Clean up the code you have just written – Aim for nice, clean structure, naming, etc.

What is refactoring?

The "Red"

First, we write a test

- This really amounts to design by example
- We make API decisions
- We're thinking hard about how code is used
- We're taking a client perspective
- We're working at a very small scale
- Example test for a Stack

```
Stack stack = new Stack();
stack.push(x);
y = stack.pop();
assertEquals(x,y);
```

The "Green"

Then, we just write enough code so that the test case can run

- We don't write more code
- All we want is to make the test pass
  - It should be a very small step
  - Implementation probably not optimal
  - We don't care about it (Yet!)
- Example the test case of the Stack
  - Write the default contructor, `push`, and `pop`, nothing more

The "Refactoring"

And then we fefactor

- TDD Without refactoring likely makes ugly code – thoroughly tested, but ugly
- There are a variety of transforms/refactoring to address this, but
- Developing in small increments
- The Code always runs and past the test cases!
  - Changes are small enough to fit in our heads
  - Timeframe is minutes to (maybe) hours
- Evolutionary design
  - Anticipated vs. unanticipated changes
  - Many "anticipated changes" turn out to be unnecessary

Refactoring

A disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior

- Keeping code healthy with refactoring (meaning?)
- Refactoring is disciplined – Wait for a problem before solving it
- Refactorings are transformations – Many refactorings are simply applications of patterns
- Refactorings alter internal structure
- Refactorings preserve behavior

Final Step

Making sure the software still works

- Protection with automated tests
  - Test harness is only thing that ensures software works
  - Rerun tests after each change (Regression Testing)
- Fast feedback
  - Sometimes, entire test suite is too slow – role of continuous integration servers to execute regression tests in background
- Management of multiple developers/multiple conflicting changes

Outline

Tools for TDD

- JUnit
- Also Continuous Integration Servers, e.g. Jenkins, Travis CI, Github Actions, etc

A demo ...

Let's fire up Android Studio and see how it works ...

Outline

Summary

TDD

TDD workflow

Demo

Outline

References

"Introduction to Software Testing" 2nd Edition. Ammann and Offutt

JUnit 5 User Guide

JUnit 4 Getting Strated Guide