Behaviour-Driven Development with D and Cucumber

@atilaneves

Átila Neves, PhD

Cisco Systems

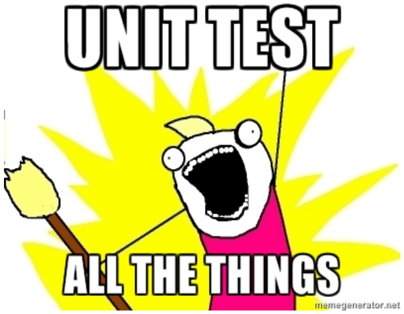
DConf 2015

Outline

- My Software Testing Journey
- TDD – what it is, what it’s for, how it’s done
- Cucumber: a BDD framework
- BDD – how it expands on TDD
- Short BDD example
- Writing command-line D programs in BDD fashion
- Using Cucumber to drive D code for integration / system / acceptance testing

My Software Testing Journey

- Manual testing. Once.
- Learned about JUnit and UTs in 2003
- Confusion about the different types of testing
- UTs for all production code
- TDD
- Automated defect discovery of **unit-testable** code, but other bugs still emerging



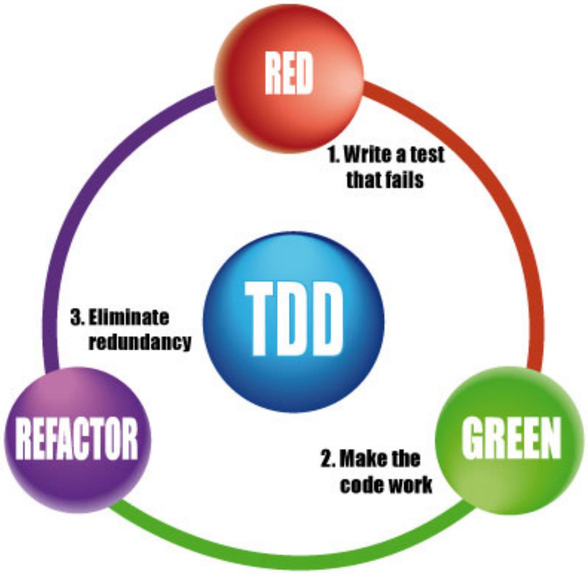
Unit Tests: my definition

- Unit tests are automated.
- Unit tests are small.
- Unit tests are independent of one another.
- Unit tests only use the CPU and RAM. No contact with the outside world.
- Unit tests are fast (<10ms).
- Unit tests are **repeatable**, **deterministic**, **fast** and **easy**.

- Compile-time?

TDD: a way to unit test

- Write the test before, not after, the code to be tested



Why TDD?

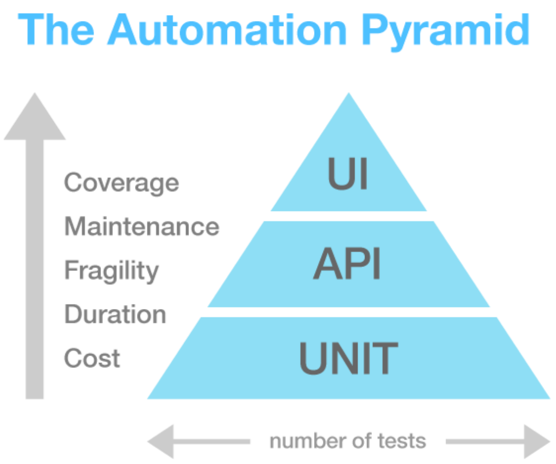
- Confidence that the production code works as intended
- Runnable documentation
- Lower coupling in the code under test
- It can often be easier to write a test than production code
- Can help with the design of a software system
- Reduces the possibility of bugs in the test code
- Good code coverage

TDD shortcomings

- A good fit for the mental model of certain people, but not everyone
- Not indicated when exploratory programming is desired or the only option
- Should however be mandatory for bug fixing
- The most important thing is to write the tests, whether before or after the production code

But not all code is unit-testable...

- Production code tends to do pesky things like use the file system, send/ receive packets, talk to DBs...
- Real code deals with the real world, which is messy.
- Layered testing approach: lower-level tests before the higher-level ones: unit, integration, system, acceptance.
- D has built-in unit tests, as well as a few unit testing libraries
- What to use for higher-level tests?



cucumber

- BDD tool written in Ruby
- Uses its own DSL called Gherkin
- Features are written and described in plain text, then mapped to Ruby code blocks with regular expressions

Cucumber: feature example

```
Feature: Calculator
  As a calculator user
  I want to add, multiply and divide numbers
  So I can do simple maths quickly

  Scenario: Adding two numbers
    Given a calculator
    When the calculator adds 3 and 4
    Then the calculator returns 7
```

Cucumber: step definitions

```
Given(/a calculator/) do
  @calc = Calculator.new
end

When(/the calculator adds (\d+) and (\d+)/) do |x, y|
  @calc.add(x.to_i, y.to_i)
end

Then(/the calculator returns (\d+)/) do |x|
  expect(@calc.result).to eq(x.to_i)
end
```

Aruba: A Cucumber plugin

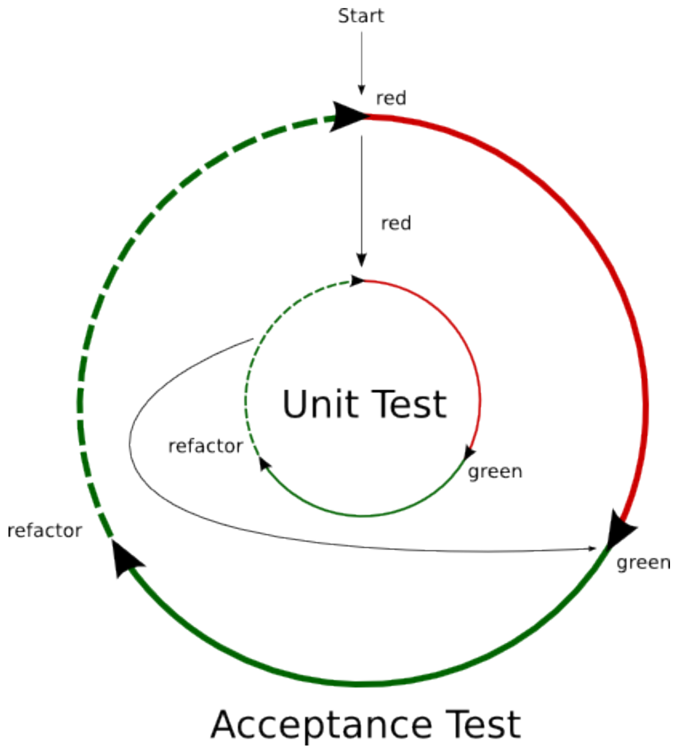
- Built-in step definitions for testing command-line programs
- Manipulation of filesystem state, reset after every test
- Creates and manipulates files in a sandbox

Sample Cucumber/Aruba feature

```
Feature: Adder

  Scenario: Correct sum
    Given a file named "adder.d" with:
      """
      import std.stdio, std.conv;
      void main(string[] args) {
          writeln(`The sum of `, args[1], ` and `, args[2], ` is `,
                  args[1].to!int + args[2].to!int);
      }
      """
    When I run `rdmd adder.d 2 3`
    Then the output should contain:
      """
      The sum of 2 and 3 is 5
      """
```

The BDD Cycle



BDD example: feature

```
Feature: Control request
  As a protocol client
  I want to get a response from my control request message
  So that I can initiate a probe

  Scenario: Handshake V2
    Given I have started the responder
    When I send a CONTROL REQUEST V2 message
    Then I should successfully receive a CONTROL RESPONSE V2 message
```

BDD: 1st feature pending

```
1 scenario (1 undefined)
3 steps (3 undefined)
0m0.003s

You can implement step definitions for undefined steps with these snippets:

Given(/^I have started the IPSLA responder$/) do
  pending # express the regexp above with the code you wish you had
end

When(/^I sent a CONTROL REQUEST message$/) do
  pending # express the regexp above with the code you wish you had
end

Then(/^I should receive a CONTROL RESPONSE message$/) do
  pending # express the regexp above with the code you wish you had
end
```

BDD: 1st feature failing

```
Scenario: Positive test                          # features/request.feature:6
  Given I have started the IPSLA responder        # features/step_definitions/steps.rb:27
    No such file or directory - bin/ipsla_responder (Errno::ENOENT)
    ./features/step_definitions/steps.rb:13:in `popen'
    ./features/step_definitions/steps.rb:13:in `run_responder'
    ./features/step_definitions/steps.rb:28:in `/^I have started the IPSLA responder$/'
    features/request.feature:7:in `Given I have started the IPSLA responder'
  When I send a CONTROL REQUEST message           # features/step_definitions/steps.rb:63
  Then I should receive a CONTROL RESPONSE message # features/step_definitions/steps.rb:67

Failing Scenarios:
cucumber features/request.feature:6 # Scenario: Positive test

1 scenario (1 failed)
3 steps (1 failed, 2 skipped)
```

BDD: The first unit test

```
const(ubyte)[] bytes(ubyte ctrlVersion = 2, ushort status = 0) {
    ubyte status1 = status >> 8;
    ubyte status0 = cast(ubyte)(status & 0xff);
    return
        [ctrlVersion, 0, status1, status0] ~ // ver8, reserved8, status16
        [0, 0, 0, 0] ~ // seq no
        ...;
}
void testVersion() {
    IpslaControlV2(bytes).ctrlVersion.shouldEqual(2);
    IpslaControlV2(bytes(3)).ctrlVersion.shouldEqual(3);
}
```

Advantages of BDD

- Fully (mostly) tested code
- When a feature is green, it’s implemented
- Forces the code to do “real work” early
- Code tends to be less crufty: YAGNI is enforced by the process

Disadvantages of BDD

- It takes longer to write code
- More complicated than TDD
- Has the same problem TDD has with exploratory coding
- Like TDD, also isn’t for everyone

How to implement step definitions in D?

- Cucumber defines a JSON wire protocol to interface with other languages
  - Asks the server to tell it which steps exist
  - Asks the server to execute certain steps and report results
- The wire protocol is defined... using Cucumber!
- Unencumbered is a Cucumber wire protocol implementation in D
  - https://github.com/atilaneves/unencumbered
- Uses UDAs and compile-time reflection to link steps with code
  - Similar to the Python and Java implementations

Sample from the “definition”of the wire protocol

```
Scenario: Invoke a step definition which passes
  Given there is a wire server running on port 54321 which understands the following protocol:
   | request                                    | response                    |
   | ["step_matches",{"name_to_match":"we're all wired"}] | ["success",[{"id":"1", "args":[]}]] |
   | ["begin_scenario"]                         | ["success"]                 |
   | ["invoke",{"id":"1","args":[]}]            | ["success"]                 |
   | ["end_scenario"]                           | ["success"]                 |
  When I run `cucumber -f progress`
  And it should pass with:
    """
    .

    1 scenario (1 passed)
    1 step (1 passed)

    """
```

Unencumbered: Write Cucumber step definitions in D

- Unencumbered is a Cucumber wire protocol implementation in D
  - https://github.com/atilaneves/unencumbered
- Uses UDAs and compile-time reflection to link steps with code
  - Similar to the Python and Java implementations

```
Calculator calc;
@Given(r"^a calculator$") void initCalculator()  {  calc = Calculator(); }

@And(r"^the calculator adds up ([0-9.]+) and ([0-9.]+)$")
void andAddsUp(double a, double b) { calc.add(a, b); }

@Then(`^the calculator returns "(.+)"`)
void thenReturns(double a) { assert(closeEnough(calc.result, a)); }
```

How does the server know about the steps?

```
import cucumber.server;


shared static this() {
    runCucumberServer!"tests.calculator.steps"(54321, Yes.details);
}
```

How are the found functions stored?

- Several functions with different types and arity, what’s the common type?
- Easy solution: void function(string[])[] steps;

```
@And(...)
void andAddsUp(string[] args) {
    calc.add(args[1].to!double, args[1].to!double);
}
```

- Can’t the compiler write the boilerplate for me? (it’s D, so umm.. yeah)
  - For each step, count the number of capturing parentheses
  - Statically reflect on the arity and types of the input parameters
  - mixin(`steps ~= Step((cs) { andAddsUp(cs[0].to!double, cs[1].to!double) }, ...`);
  - Profit!

D Goodies

- Compile-time checks
  - If the capturing parentheses don’t match the function arity:
    - Error: static assert "Arity of andAddsUp (2) does not match the number of capturing parens (3) in ^the calculator adds up ([0-9.]+) and ([0-9.]+)()$“
  - If the regex is not valid:
    - Error: uncaught CTFE exception std.regex.internal.ir.RegexException("Unmatched ')'\x0aPattern with error: `^the calculator adds up ([0-9.]+) and ([0-9.]+))` <--HERE-- `$`"c)
- D exceptions
  - I'm an exception (tests.calculator.steps.MyCustomException from localhost:54321)

Further work

- Unencumbered could be a D-only alternative implementation
  - Pull requests welcome
- Lambdas?
  - Having to name the step functions is tedious, as is the return type
  - Java’s solution doesn’t work in D: UDAs must apply to **something**