CSC 440/540: Software Engineering

Test Driven Development and Refactoring
Topics

1. Bugs
2. Software Testing
3. Test Driven Development (TDD)
4. Behavior Driven Development (BDD)
5. Refactoring
6. Automating Acceptance Tests
Bugs

Photo # NH 96566-KN  First Computer "Bug", 1945

0800  Anchor started
1000  Should - anchor

13:00 (03:5) MP - MC
13:00 (03:5) PRO
13:00 (03:5) COW

Relays 6-2 in 033 failed special speed test
In Relay

15:25  Started cosine tape (sine check)

15:45  Started multi-adder test

First actual case of bug being found.

1940  Anchor started.
1700  Closed down.
Ariane 5 Flight 501 Bug

Ariane 5 spacecraft self-destructed

- June 4, 1996
- Due to overflow in conversion from a floating point to a signed integer.
- Spacecraft cost $1 billion to build.
Software Testing

Software testing is the process of evaluating software to find defects and assess its quality.
Test Granularity

1. **Unit Tests**
   Test specific section of code, typically a single function.

2. **Component Tests**
   Test interface of component with other components.

3. **System Tests**
   - End-to-end test of working system.
   - Also known as Acceptance Tests.
Regression Testing

**Regression testing** focuses on finding defects after a major code change has occurred. Regressions are defects such as

- Reappearance of a bug that was previously fixed.
- Features that no longer work correctly.
How to find test inputs

Random inputs
• Also known as fuzz testing.

Boundary values
• Test boundary conditions: smallest input, biggest, etc.
• Errors are likely to occur around boundaries.

Equivalence classes
• Divide input space into classes that should be handled in the same way by system.
How to determine if test is ok?
Test Driven Development

1. (Re)Write a test
2. Check if the test fails
   - Test fails
   - Write production code
   - Run all tests
3. Test(s) fail
   - All tests succeed
4. Clean up code
5. Repeat

Flowchart:
- (Re)Write a test
- Check if the test fails
- Write production code
- Run all tests
- Clean up code
- Repeat

CSC 440/540: Software Engineering
How to get a test to work

1. **Fake it**—return what the test needs, nothing more.
   1. If a test is checking whether `multiply(5,2)` is 10,
   2. You should write code that returns 10, nothing more.

2. **Obvious**—when the code to make the test pass is trivial.

3. **Triangulation**—write a second test that exposes the Fake It solution as insufficient.
   1. Test for `multiply(5,3)`
   2. Think about edge cases: zero, INT_MAX
Advantages of writing tests first

- Units tests are actually written.
- Programmer satisfaction.
- Clarification of detailed interface and behavior.
- Provable, repeatable, automated verification.
- Confidence to change code.
FizzBuzz TDD Example

Rules of FizzBuzz

- If a number is divisible by 3, return “Fizz”
- If a number is divisible by 5, return “Buzz”
- If a number is divisible by 15, return “FizzBuzz”
- Otherwise, return the number.

Initialization

- Make a directory under workspace named tdd.
- Create a Gemfile with
  - source “https://rubygems.org”
  - gem minitest
- Run bundle install.
Create fizzbuzz_test.rb

```ruby
require 'minitest/autorun'

class FizzBuzzTest < Minitest::Test
  def test_divisible_by_three
    myfb = FizzBuzz.new
    assert myfb.is_divisible_by_three?(3)
  end
end
```

$ ruby fizzbuzz_test.rb
Run options: --seed 19581

# Running:

E

Finished in 0.001046s, 955.8201 runs/s, 0.0000 assertions/s.

1) Error:
FizzBuzzTest#test_divisible_by_three:
NameError: uninitialized constant FizzBuzzTest::FizzBuzz
Did you mean?  FizzBuzzTest
   fizzbuzz_test.rb:5:in `test_divisible_by_three'

1 runs, 0 assertions, 0 failures, 1 errors, 0 skips
Create fizzbuzz.rb

Add require './fizzbuzz’ to fizzbuzz_test.rb.

fizzbuzz.rb contents:

class FizzBuzz
    def is_divisible_by_three?(num)
        true
    end
end
Step 2: Test Succeeds

$ ruby fizzbuzz_test.rb
Run options: --seed 53856

# Running:
.

Finished in 0.001028s, 973.1166 runs/s, 973.1166 assertions/s.

1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
Step 3: Refactor

- If our code is complex, we may want to refactor.
- If we don’t need to refactor, we return to Step 1.
  - Our code is faking it,
  - So let’s add a test for a number that’s not divisible by 3.
require 'minitest/autorun'
require './fizzbuzz'

class FizzBuzzTest < Minitest::Test
  def test_divisible_by_three
    myfb = FizzBuzz.new
    assert myfb.is_divisible_by_three?(3)
    assert ! myfb.is_divisible_by_three?(2)
  end
end
Step 2: fizzbuzz.rb

```ruby
class FizzBuzz
  def is_divisible_by_three?(num)
    num % 3 == 0
  end
end
end
```
Step 1: Add fizzbuzz() test

class FizzBuzzTest < Minitest::Test
  def test_fizzbuzz
    assert_equal(fizzbuzz(3), "Fizz")
  end
end
Continue TDD at c9.io

Find complete minitest documentation at https://devdocs.io/minitest/.
Behavior-Driven Development

Use behavioral specifications instead of tests.

- Write in an English-like DSL that describes what users want.
- Error messages become explanations of missing or wrong behavior due to English text in behavior specs.
- Frameworks: JBehave, Cucumber, RSpec

Example

**Scenario 1:** Refunded items should be returned to stock

**Given** that a customer previously bought a black sweater from me

**And** I have three black sweaters in stock.

**When** he returns the black sweater for a refund

**Then** I should have four black sweaters in stock.
describe FizzBuzz do
  before do
    @myfb = Fizzbuzz.new
  end

  describe "when asked if divisible by 3" do
    it "must be divisible by 3" do
      @myfb.is_divisible_by_three(3).must_equal true
    end
  end
end

csc 440/540: Software Engineering
Refactoring

Refactoring is a structured technique for changing the design and implementation of code without changing its functionality.

Goals of refactoring

- Reduce complexity
- Improve code readability
- Improve maintainability
- Improve flexibility
Refactoring Techniques

Techniques for breaking code apart into logical pieces

- **Extract Class** moves part of the code from an existing class into a new class.
- **Extract Method** turns part of a larger method into a new method, improving method cohesion.

Techniques for improving names and location of code

- **Move Method** or **Move Attribute** – move to a more appropriate class or source file.
- **Rename Method** or **Rename Attribute** – change name to a new one that better reveals purpose.
- **Pull Up** – move method or attribute to a superclass
- **Push Down** – move method or attribute to a subclass
Refactoring Techniques

Techniques that allow for more abstraction

- **Encapsulate Field** – force code to access the field with getter and setter methods, reducing coupling.
- **Generalize Type** – create more general types to allow for more code sharing
- Replace conditional with polymorphism.

There are many other refactorings.

- Or Martin Fowler’s book *Refactoring*
Acceptance Test Frameworks

Cucumber (Ruby)
- Uses declarative language to describe tests.
- Similar tools: Behat (PHP), Lettuce (Python), JBehave

Fit (Framework for Integrated Test)
- Customers provide examples formatted as tables
- Fit checks document and fills in table red/green.

Selenium (Selenese DSL)
- Software testing framework for web applications.
- Can use record/playback tool without scripting.
- Ports: C#, Java, Perl, PHP, Python, Ruby
Web Automation

**HtmlUnit**
- Java based headless browser with JS.

**Mechanize**
- Automates HTTP form interactions, but not JS.
- Available for Ruby, Python, and Perl.

**PhantomJS**
- Headless, scriptable WebKit browser with JS, HTML5
- CasperJS same interface with Firefox browser.
- TrifleJS same interface with IE browser.

**ScriptableBrowser**
- Headless browser that’s part of PHP SimpleTest.

**Spynner**
- Headless WebKit browser with AJAX support for Python.
References


