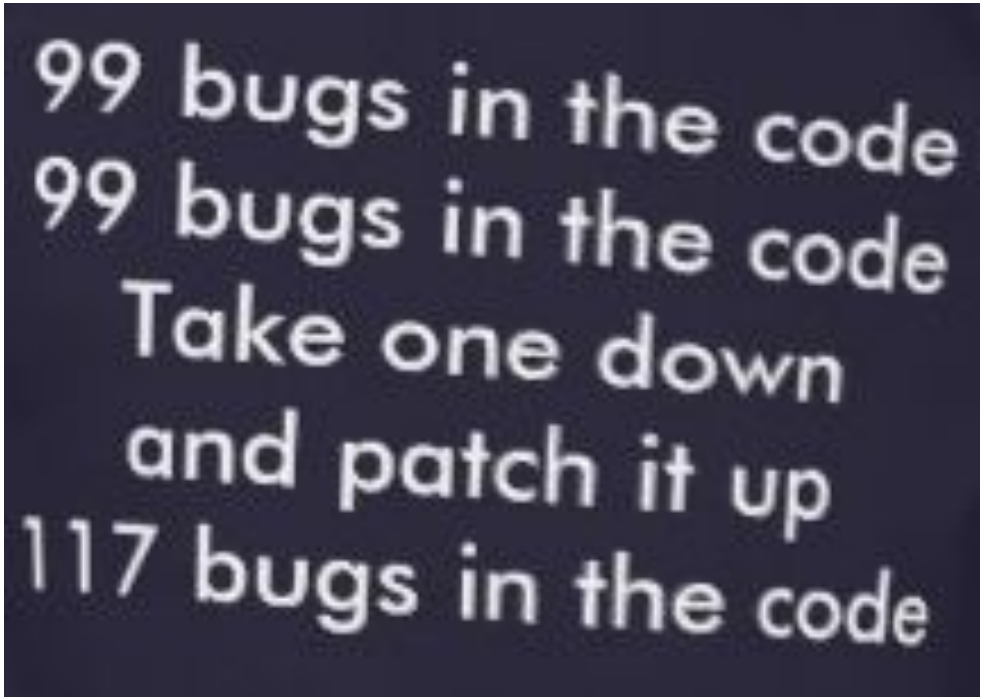
CS1632 Lecture 10

TDD

Bill Laboon/Dustin Iser

– – –


99 bugs in the code
99 bugs in the code
Take one down
and patch it up
117 bugs in the code

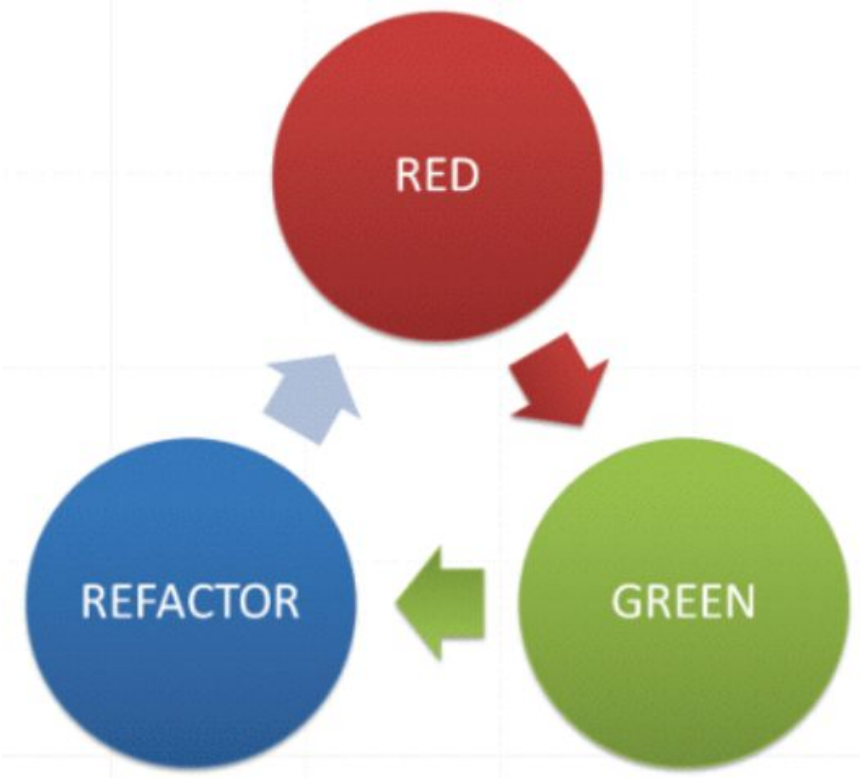
Test-Driven Development

A particular software development methodology which embraces test-first development along with several other tenets, such as continuous refactoring and expectation of change.

Tenets

- Writing tests before writing code
- Writing only code that is tested
- Writing only tests that test the code
- A very short turnaround cycle
- Refactoring early and often

Red-Green-Refactor Loop



Red-Green-Refactor Loop

- Red - Write a test for new functionality.
  - This test should fail immediately.
- Green
  - Write only enough code to make the test pass.
- Refactor
  - Review the code and make it better.

Red-Green-Refactor Loop

---

1. Write a test for new functionality.
2. Run test suite. Only the new test should fail.
3. Write only enough code to make the failing test pass.
4. Run tests suite.
5. If any tests fail, go to step 3.
6. Refactor code.
7. Run test suite.
8. If any tests fail, go to step 6.
9. Go to step 1.

Step 1 - Write a test

This test should be a small unit of functionality. Something like one input value and an expected output for a method.

For pure TDD, you should not write multiple tests or tests which are very complex.

Step 2 - Run Test Suite

Run all the tests. Only the one you've just added should fail.

If it doesn't fail, you've already written the code for that test and the test is redundant.

Step 3 - Write the code

Write just enough code to make the test pass.

Avoid the temptation to over-engineer your solutions or add more functionality than the test covers.

Step 4 - Rerun the test suite

All the tests should pass this time.

Otherwise:

- If only your new test fails, you have not written your code (or possibly test) correctly.
- If other tests fail, you have created a regression failure.

Step 5 - Fix any failing tests

Step 6 - Refactor

Make it green, then make it clean.

Clean up your code to make it look nicer, perform better, or easier to understand.

- Poor algorithm choice
- Bad variable names
- Poor performance
- Needs comments
- Difficult to understand
- Bad design

Step 7 - Rerun test suite

Make sure that your refactoring did not cause any problems.

You should have the same functionality as before, just better code.

Step 8 - Fix any failing tests

Step 9 - Go back to step 1

If you have more features to implement, go back to step 1.

KISS

Keep it short and simple

Fake it ‘til you make it

Avoid slow-running tests

Each iteration requires at least three test suite runs.

Short feedback loops are critical with TDD.

Benefits of TDD

- Automatically create tests.
  - Research shows that more tests are correlated with fewer defects.
- Makes writing tests easy because it’s done often.
  - Anything you do often, you learn how to do better.
- Tests are relevant.
  - The tests are testing the exact functionality you are implementing.
- Developer is focused on the end result, not code.
- Ensures you take small steps.
  - Research shows that more senior engineers take smaller steps.
- Ensures the code is extensible and testable.
- Confidence in codebase

Drawbacks of TDD

- Extra time up front.
  - May be saved in the long run by fewer defects and less manual testing.
- Focus on unit tests may take away attention from other types of tests.
- Not appropriate for prototyping.
- Not appropriate for large architectural changes.
- Test code, just like application code, must be maintained.
- Easy to overtest.
- May be difficult to implement on legacy code that is not testable.

FizzBuzz with TDD

Print out the numbers from 1 to 100, each on a separate line. If a number is evenly divisible by 3, print “Fizz” instead. If a number is evenly divisible by 5, print “Buzz” instead. If a number is evenly divisible by 3 and 5, print “FizzBuzz” instead.

Write a test

```
@Test
public void testNumber() {
  assertEquals(_fb.value(1), "1");
 }
```

Write code until the test passes

```
public String value(int n) {
  return "1";
}
```

Add another test

```
@Test
public void testNumber2() {
  assertEquals(_fb.value(2), "2");
}
```

Write code until the test passes

```
    public String value(int n) {
     if (n == 1) {
      return "1";
     } else {
      return "2";
     }
   }
```

Refactor

```
public String value(int n) {
   return String.valueOf(n);
}
```

Add another test

```
@Test
public void testNumber3() {
  assertEquals(_fb.value(3), "Fizz");
}
```

Write code until the test passes

```
private boolean fizzy(int n) {
  return (n % 3 == 0);
}

public String value(int n) {
  if (fizzy(n)) {
   return "Fizz";
  } else {
   return String.valueOf(n);
  }
}
```

Add another test

```
@Test
public void testNumber5() {
  assertEquals(_fb.value(5), "Buzz");
}
```

Write code until the test passes

```
private boolean buzzy(int n) {
   return (n % 5 == 0);
}
public String value(int n) {
   if (fizzy(n)) {
     return "Fizz";
   } else if (buzzy(n)) {
     return "Buzz";
   } else {
     return String.valueOf(n);
   }
}
```

Add another test

```
  @Test
public void testNumber15() {
   assertEquals(_fb.value(15), "FizzBuzz");
}
```

Write code until the test passes

```
public String value(int n) {
  if (fizzy(n) && buzzy(n)) {
   return "FizzBuzz";
  } else if  (fizzy(n)) {
   return "Fizz";
  } else if (buzzy(n)) {
   return "Buzz";
  } else {
   return String.valueOf(n);
  }
}
```

TDD doesn’t have to be pure

Just like everything else, TDD is a tool to help you build great software.

If you feel comfortable doing so, it may be alright to write more than one test in an iteration.

TDD is just one example

There are other kinds of test-first development methodologies, such as ATDD and BDD.

TDD is not an industry accepted standard. Google “TDD is dead” for an argument against using TDD.