Software Engineering
Test-Driven Development

Erik Fredericks // frederer@gvsu.edu

Adapted from materials provided by Byron DeVries, Jagadeesh Nandigam

What is Test-Driven Development (TDD)?

TDD Process

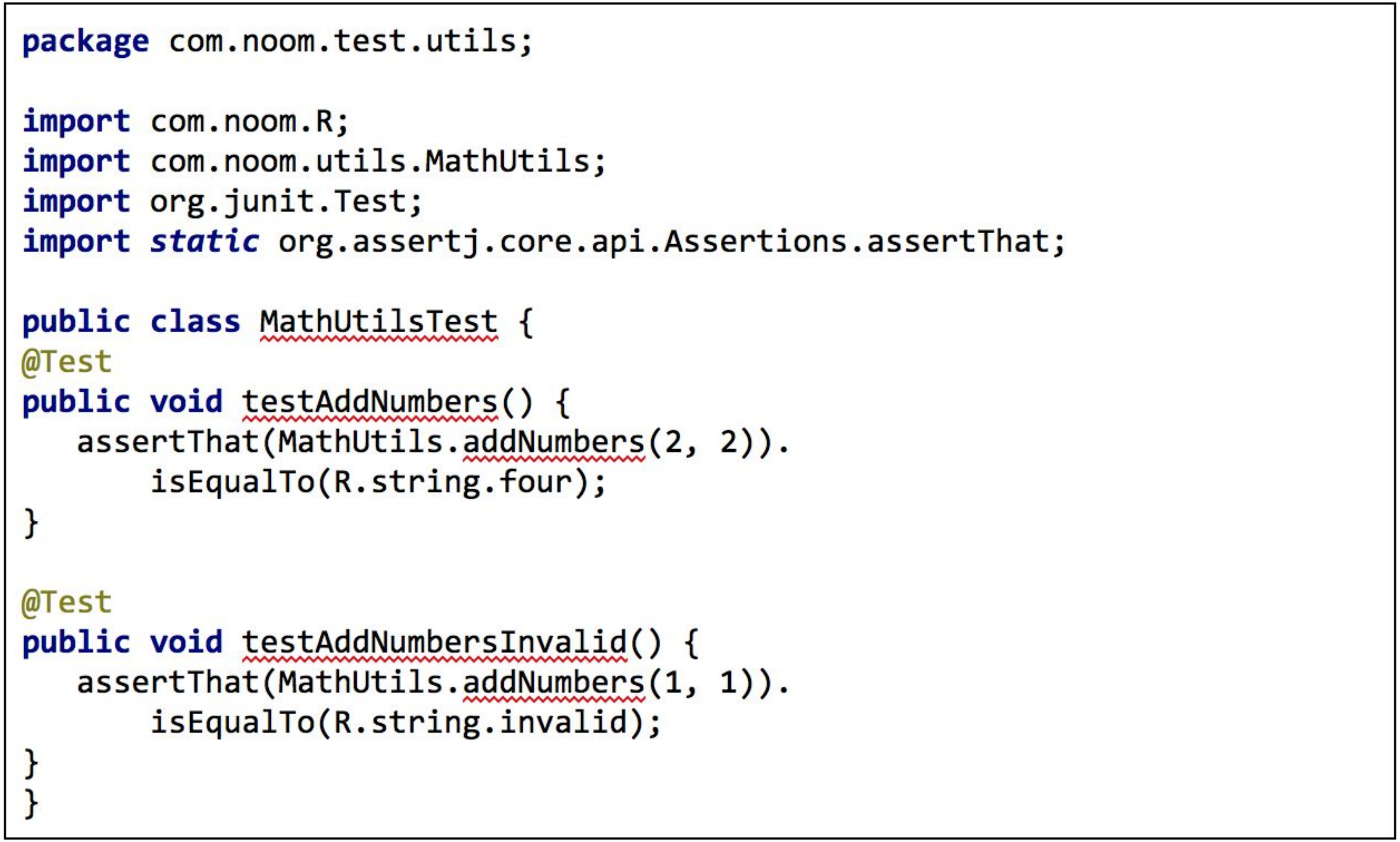
- Red/Green/Refactor Cycle
- Test-First Stoplight
- Step-by-Step Tasks in TDD

Benefits of TDD

Why do Developers Avoid Unit Tests?

Managing Unit Tests

What is a test?

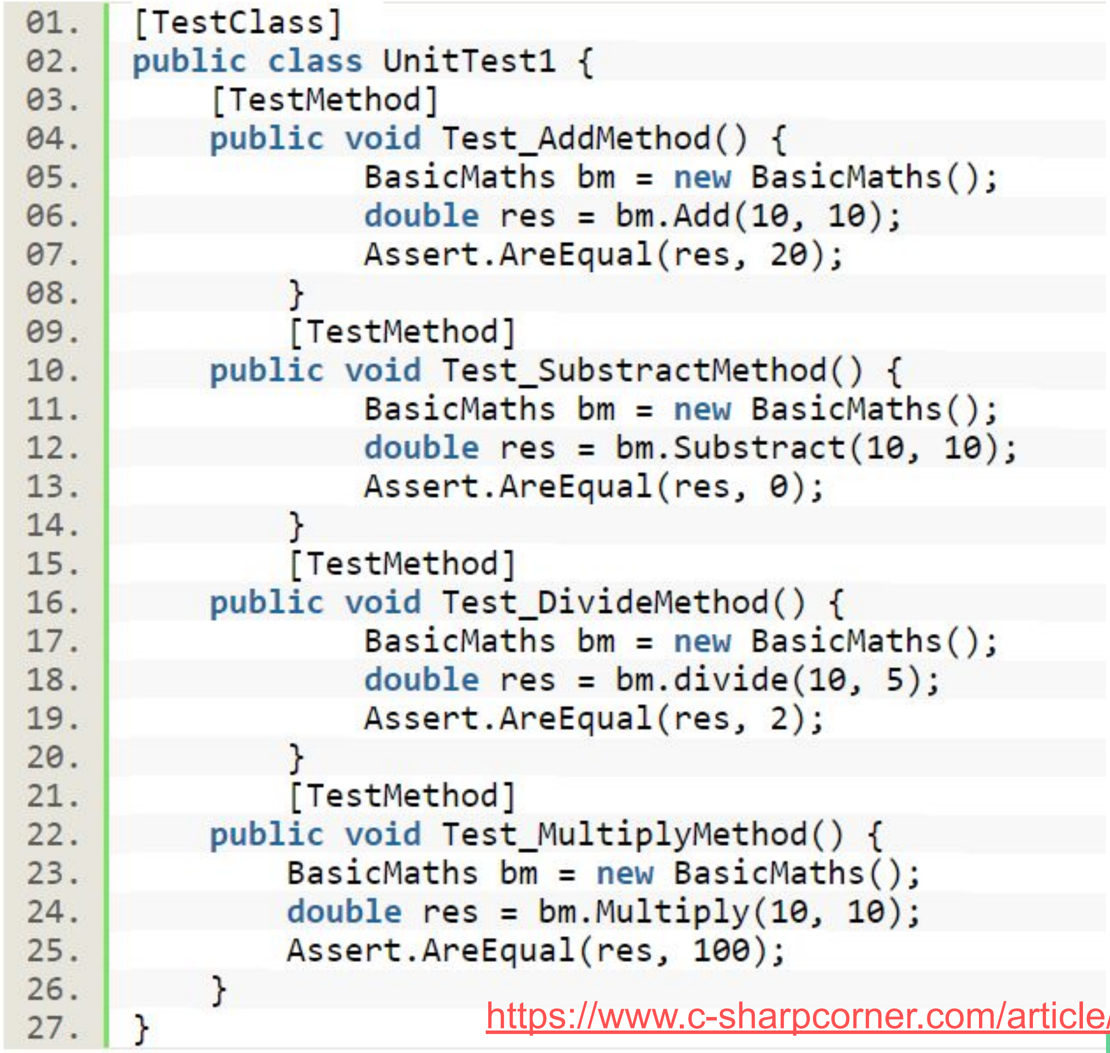


```java
package com.noom.test.utils;

import com.noom.R;
import com.noom.utils.MathUtils;
import org.junit.Test;
import static org.assertj.core.api.Assertions.assertThat;

public class MathUtilsTest {
@Test
public void testAddNumbers() {
   assertThat(MathUtils.addNumbers(2, 2)).
       isEqualTo(R.string.four);
}

@Test
public void testAddNumbersInvalid() {
   assertThat(MathUtils.addNumbers(1, 1)).
       isEqualTo(R.string.invalid);
}
}
```



```
[TestClass]
public class UnitTest1 {
    [TestMethod]
    public void Test_AddMethod() {
            BasicMaths bm = new BasicMaths();
            double res = bm.Add(10, 10);
            Assert.AreEqual(res, 20);
        }
        [TestMethod]
    public void Test_SubstractMethod() {
            BasicMaths bm = new BasicMaths();
            double res = bm.Substract(10, 10);
            Assert.AreEqual(res, 0);
        }
        [TestMethod]
    public void Test_DivideMethod() {
            BasicMaths bm = new BasicMaths();
            double res = bm.divide(10, 5);
            Assert.AreEqual(res, 2);
        }
        [TestMethod]
    public void Test_MultiplyMethod() {
        BasicMaths bm = new BasicMaths();
        double res = bm.Multiply(10, 10);
        Assert.AreEqual(res, 100);
    }
}
```

https://www.c-sharpcorner.com/article/a-basic-introduction-of-unit-test-for-beginners/

What...is...TDD?

TDD is a style of development where the following **two simple rules** are observed:

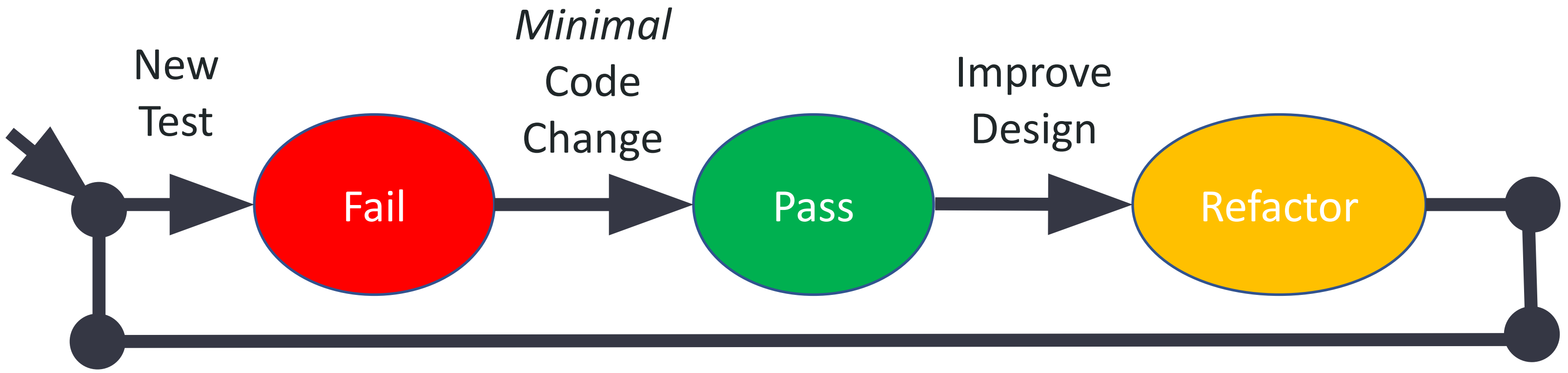
1. Write new code only if an automated test has failed, and
2. Eliminate duplication and other **code smells**.

Other Names:

- Test-Driven Design
- Test-First Development

TDD processes

- The Red/Green/Refactor cycle by Kent Beck:*



https://medium.com/@ellehallal/understanding-the-red-green-refactor-cycle-6495f995874d

*Creator of Extreme Programming

Red/Green/Refactor 🚦

[Red stage]

Don't write any code yet, just tests that will fail!

[Green stage]

Code written to pass test
(doesn't have to be perfect)

[Refactor]

Fix your ugly, spaghetti mess of code that you wrote to pass

TDD processes

The Test-First Stoplight cycle by William Wake:

Stop light progresses through Green, Yellow, and Red repeatedly.
https://xp123.com/articles/the-test-first-stoplight/

Test-first stoplight

Prepare a test list
Start the following process:

- Start (**Green Light**)
- Write a test
- Code may fail to compile (**Yellow Light**)
- Implement just enough (a stub) to compile
- Run the test and ensure it fails (**Red Light**)
- Implement just enough to make the test pass (**Green Light**)
- Improve design by refactoring
- Repeat all existing tests to ensure they are passing (**Green Light**)
- Repeat

The goal is to decrease the interval between writing tests and production code to a matter of a few minutes.

example!

Expect <name>MyName</name>

Green light

1) Create test

Yellow light

2) Method doesn't exist
    a) Add stub to solve

Red light

3) Method fails!

Green light

4) Fix method

```
public class Person {
    String name;
    int favorite = -1;

    public Person (String name, int favorite) {
        this.name = name;
        this.favorite = favorite;
    }
}
```

```
Person p = new Person("MyName", -1);
@Test
public void test_Person() {
  assertEquals("<name>MyName</name>", p.asXml());
}
```

```
(in Person)
public String asXml() {
  return null;
}
```

IntelliJ ➡ https://www.jetbrains.com/help/idea/junit.html

GREEN LIGHT GREEN LIGHT

```
public String asXml() {
  return "<name>" + name + "</name>";
}
```

Benefits of TDD

Tests actually get written!

- Repeatable verification of working code

Flexibility

- Alleviates fear when cleaning and/or improving code
- High-coverage test suite enables flexible designs

Documentation

- Each unit test explains how some part of the overall system works
- Tests are unambiguous documents that are executable and won’t get out of sync with the application
- Tests are low-level design documents

Benefits of TDD

Minimal Debugging

- You know where the bug is, because you just added it
- Small tests covering minimal functional code = small area to search

Better Design

- Most of the code, by definition, is testable
- TDD enforces decoupling
- Result is a highly modular and decoupled software structure

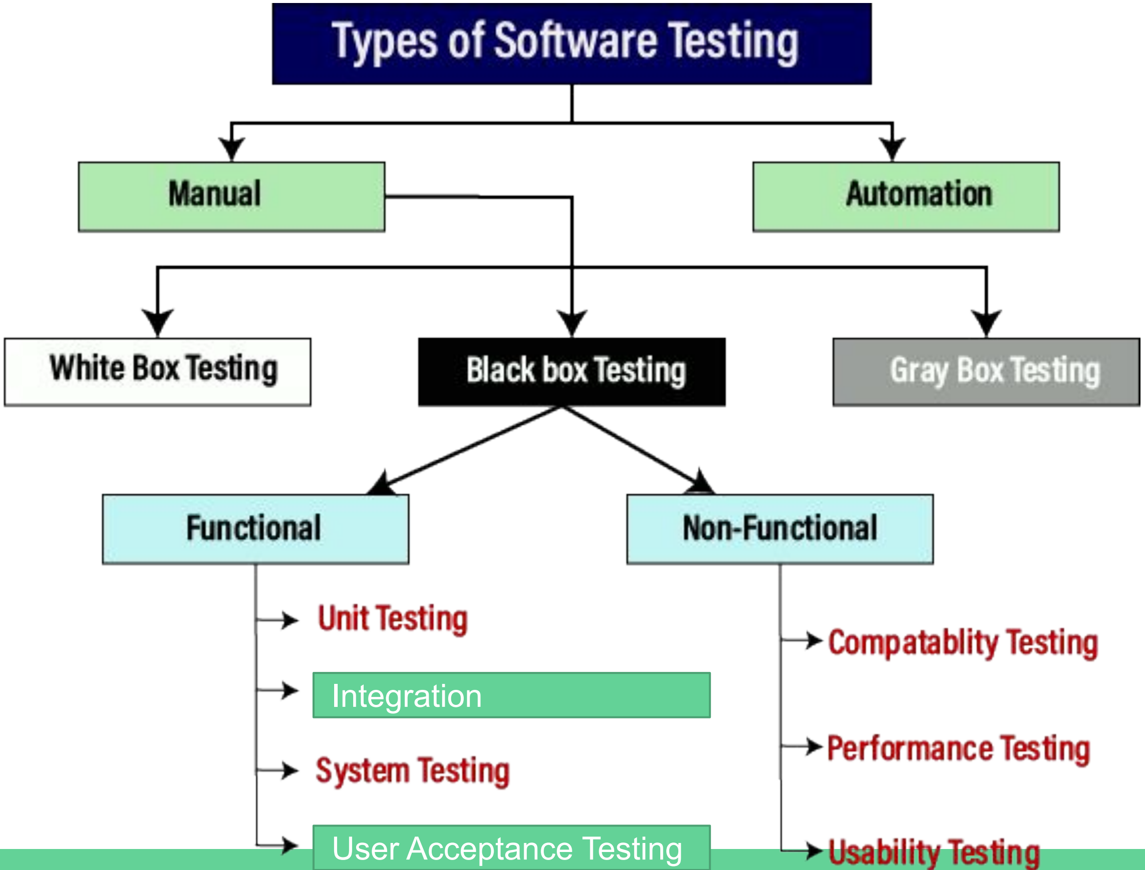
Professionalism

- Clean code
- Flexible code
- Code that works
- On time

WELCOME BACK IT HAS BEEN SO LONG I HAVE MISSED YOU ALL

What are the benefits of test-driven development?

Different types of testing



Levels of Testing

1

Unit Testing

By Developer

2

Integration Testing

By Developer & Tester

3

System Testing

By Tester

4

User Acceptance Testing

By End User / Customer

Different types of testing

many many many types of testing

Let's look at a few common ones (we're going to talk about some types more in-depth soon):

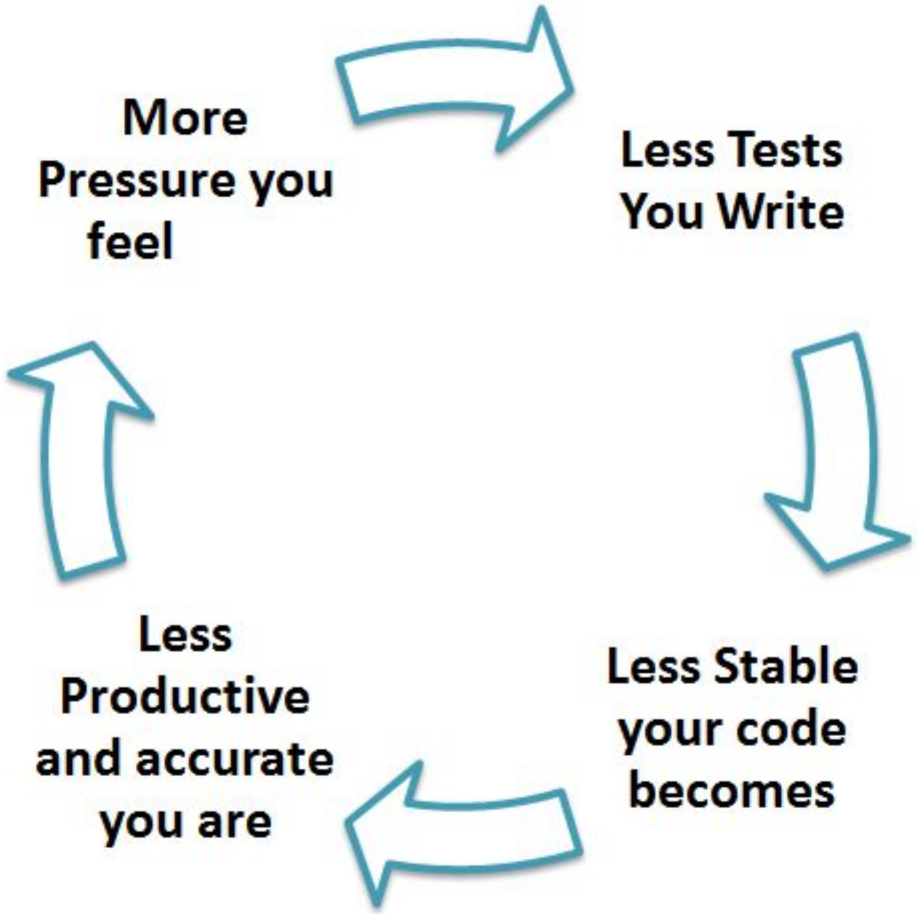
1) Unit testing
2) Integration testing
3) Regression testing
4) A/B testing
5) Stress testing

Unit testing

Test the **individual functions** / units

Basically, test the smallest parts of a program to make sure they function appropriately

- Check inputs
- Check outputs
- Check internal behavior

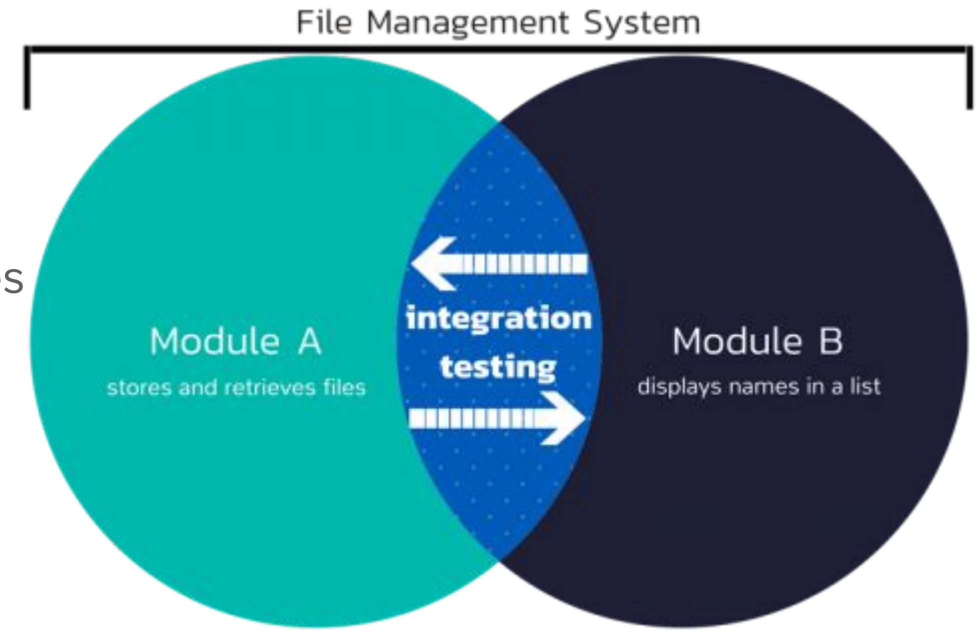

More Pressure you feel
Less Tests You Write
Less Stable your code becomes
Less Productive and accurate you are

Integration testing

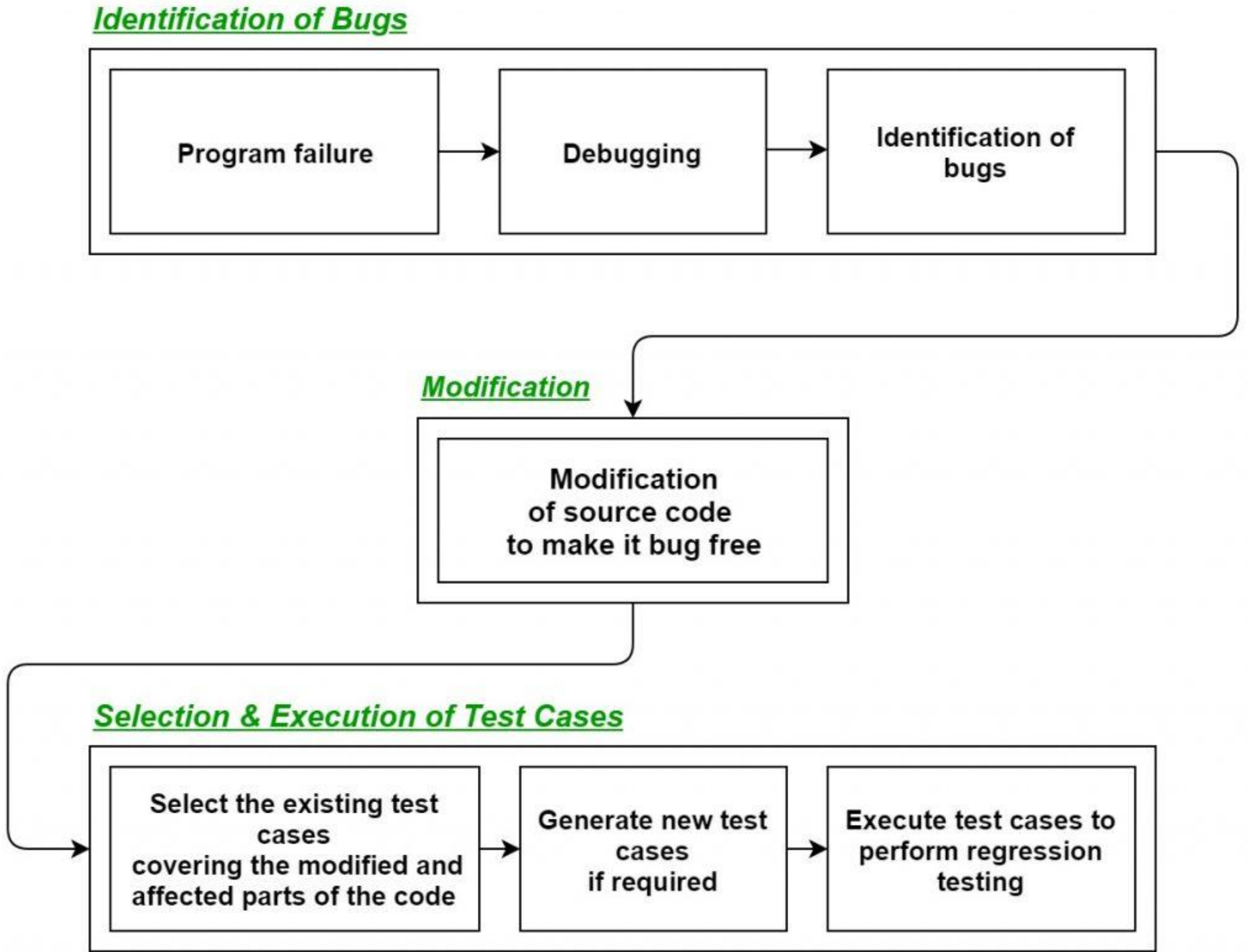
Testing to make sure two separate modules work together

1) Could mean that bringing together components still passes tests

2) Could be testing the interface between two components



Regression testing


Identification of Bugs
Program failure
Debugging
Identification of bugs
Modification
Modification of source code to make it bug free
Selection & Execution of Test Cases
Select the existing test cases covering the modified and affected parts of the code
Generate new test cases if required
Execute test cases to perform regression testing

Regression testing

Test to make sure new additions don't break prior test cases

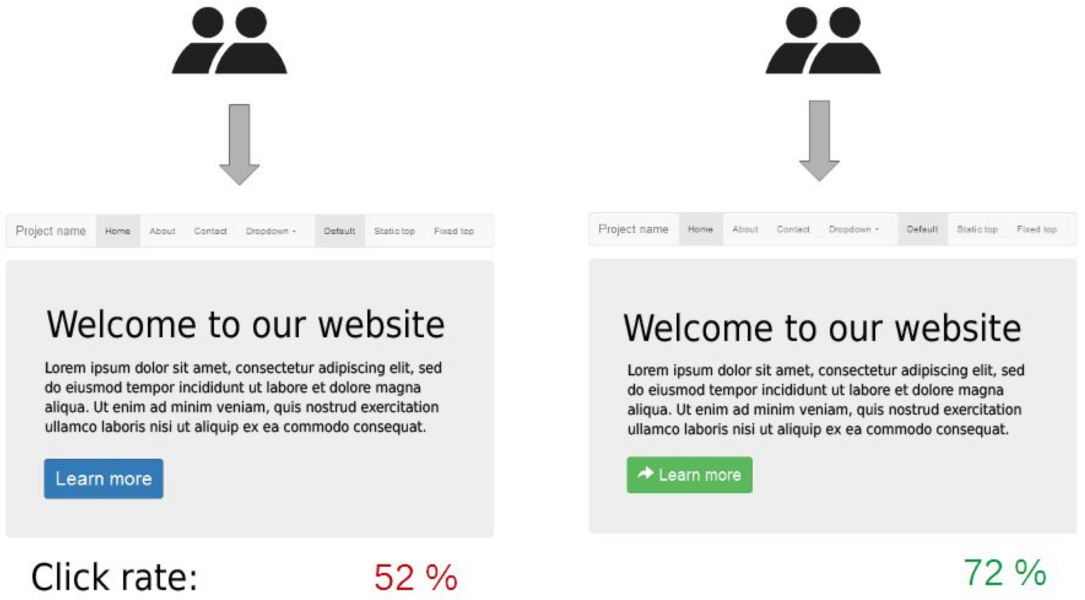
Basically, go back and re-run either all or a subset of your prior tests

What are some issues you might see here?

A/B testing

Present two variants to determine effectiveness

- Understand user engagement
- Experiment with a new version
  - (doesn't have to be UI-focused)



Welcome to our website

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Welcome to our website

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Stress testing

Well, just ... "stress" the system

Make it difficult to perform and see how it does

- i.e., check its performance under pressure

Examples:

- Overload a web server with traffic (e.g., simulate a (D)DoS attack)
- Present "load spikes" to see what happens during busy times
- Overwhelm local application with user inputs

Popular tools: Apache JMeter, LoadRunner, Locust

Why developers avoid unit tests

Common reasons (i.e., *excuses*) for not writing tests before code*:

1. Writing tests takes too much time
2. Low-level tests are useless and do not aid in integration of the whole system
3. Tests break the development flow
4. Testing is unnecessary
5. Testing is too complicated
6. I have legacy code
7. I have code that can’t be unit tested

Are any of these excuses reasonable?

*Dave Thomas & Andy Hunt

Managing unit tests

Automated unit tests will probably double the number of source files in a project

Need to manage the extra code

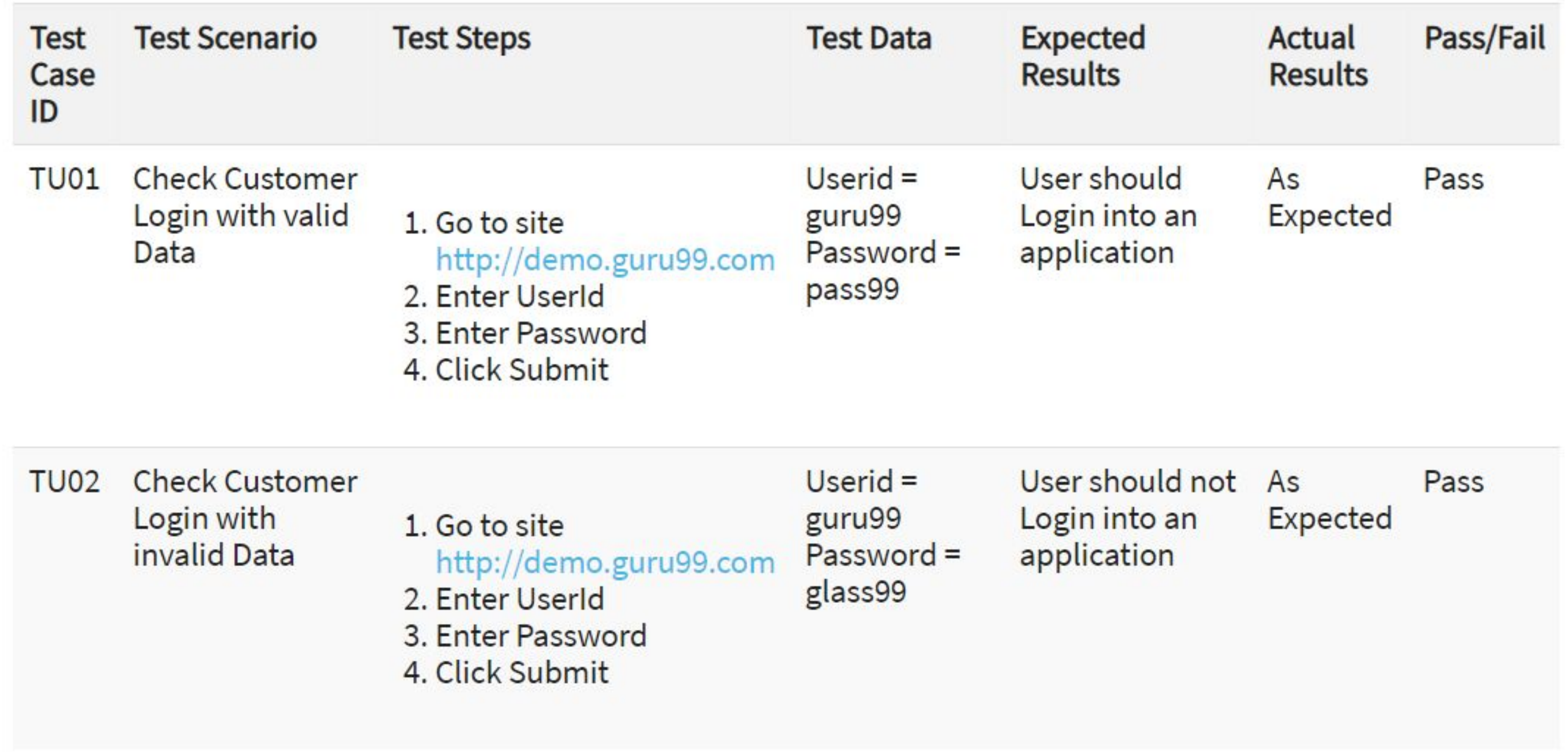
It is suggested* that:

- A “test” subdirectory is created
- A separate test file is written for each source file
- Organize tests so you can
    - Run individual tests
    - Run all the tests in a file
    - Run all the tests in a directory
    - Run all the tests in a system
- Both the building and running of the tests is automated

Some related testing stuff

What comprises a "test case" (not just a unit test)

- An ID
- Steps to execute
- Input data
- Expected output vs. actual output
- Does it pass or fail?
- What are its cross-references?



Test Case ID	Test Scenario	Test Steps	Test Data	Expected Results	Actual Results	Pass/Fail
TU01	Check Customer Login with valid Data	1. Go to site http://demo.guru99.com 2. Enter UserId 3. Enter Password 4. Click Submit	Userid = guru99 Password = pass99	User should Login into an application	As Expected	Pass
TU02	Check Customer Login with invalid Data	1. Go to site http://demo.guru99.com 2. Enter UserId 3. Enter Password 4. Click Submit	Userid = guru99 Password = glass99	User should not Login into an application	As Expected	Pass

https://www.guru99.com/test-case.html

Simple

Keep End user in Mind

Avoid **Repetition**

Do not Assume

Ensure 100% Coverage

Identifiable

Use **Testing Techniques**

Self-cleaning

Repeatable

Peer Review

examples.py.jar

JUnit v4 for me (I don't have the JUnit v5 lib locally setup)
(Project ➔ Properties ➔ Libraries ➔ JUnit 4)

(swapped to IntelliJ ➔ don't have this issue anymore!)

Java
https://www.tutorialspoint.com/junit/junit_writing_tests.htm

Python
https://docs.python.org/3/library/unittest.html