WRITING BETTER UNIT TEST

BY JEREMIE AGAN

Scope and Disclaimer

- Patron saint of Unit Testing, Roy Osherove
  - www.artofunittesting.com
- Like you, I am also a student of Unit Testing
- This does not cover the whole TDD and/or Continuous Integration
- Will cover generic techniques in writing better tests
- My goal is to change your perspective.

Cars and Software

- Imagine we're going to build a car from scratch.
- Each will be assigned to a molding metal and forming the chassis and body, others will use the mold into creating the engine, others will be sewing the cloth for the interiors, etc.
- We are going to build the car in one location all at once

Cars and Software

- Once we complete the car, we will let testers try driving the car
- If the car is broken, then we will all gather to fix it
- Have the driver test the car again
- Repeat cycle until everything works

Cars and Software

- Replace the Car with software, these isn't much any difference
- FDD – Faith Driven Development

Cars and Software

- In the real world, car manufacturers order parts from different manufacturers
- Those manufacturers develop their parts to work 100% of the time when the assembled correctly into the car
- To do this, they do rigorous testing on each batch ensuring quality.
- They are built to be modular; meaning they can installed in a test chamber/adapters without the need of a full car

Integration v Unit v Black box Testing

- Black box testing
  - It concerns testing the entire business workflow
  - Expected output/outcome depends on the specific data/behavior the tester applies to the application
  - Tests from the UI or any equivalent ‘business-end’ of the application
  - Validates the application from the user’s perspective

Integration v Unit v Black box Testing

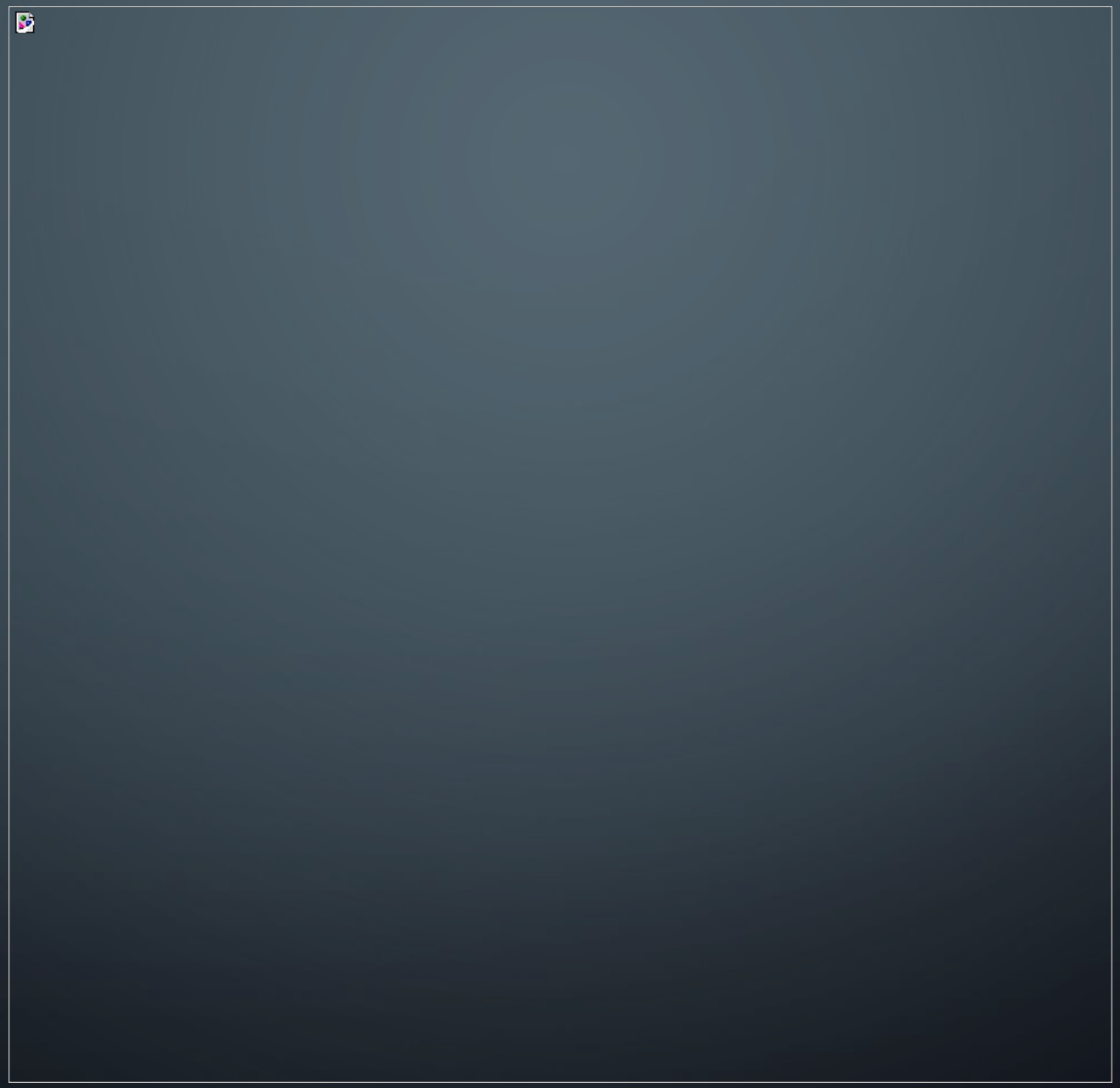
- Integration Testing
  - Full stack test of a particular method or service
  - Test the behavior of the methods and includes the supplementary methods and externals resources such as file systems, database, HTTP, xml, etc; running this is slow
  - Requires deep analysis in creating one
  - It validates a particular behavior of a method based on the **parameters/variables** set AND **the existing test data** generated prior to running the test
  - Basically, it just tells you if something is working.

Integration v Unit v Black box Testing

- Unit Testing
  - Only concerns the logic/algorithm of the current method being tested; smaller user-case
  - All supplementing methods or external resources (DB, xml, http, files system, state, etc.) will be disregarded to isolate the test;
  - External factors are either stubbed or mocked depending on the test type
  - Everything is run through memory; it is fast
  - Code coverage is measured
  - Basically, this is API Documentation
  - When all are combined, it provides check-and-balance
  - Each have their own limitations, but it is supplemented by the other two test.

Integration v Unit v Black box Testing

- When all are combined, it provides check-and-balance
- Each have their own limitations, but it is supplemented by the other two test.



The 3 main pillars of writing a good test

- **READ**able
- **MAINTAIN**able
- **TRUST**worthy

Code sample time!

```
[Test]
public void GetChangeSetForSettings_Test()
{
```

- This test failed
- Can anyone tell me why the test failed

What makes a test **READ**able?

1. Test method must be able to tell the following:
   - ➢ Name of the method being tested
   - ➢ Scenario
   - ➢ Expected Result
2. You should know at first glance where the **Arrange, Act**, and **Asserts** are.
3. Variables must be declared and named accordingly to their purpose
4. Ask yourself, “Can people still understand what the fuck I did there 4 months after I died?”

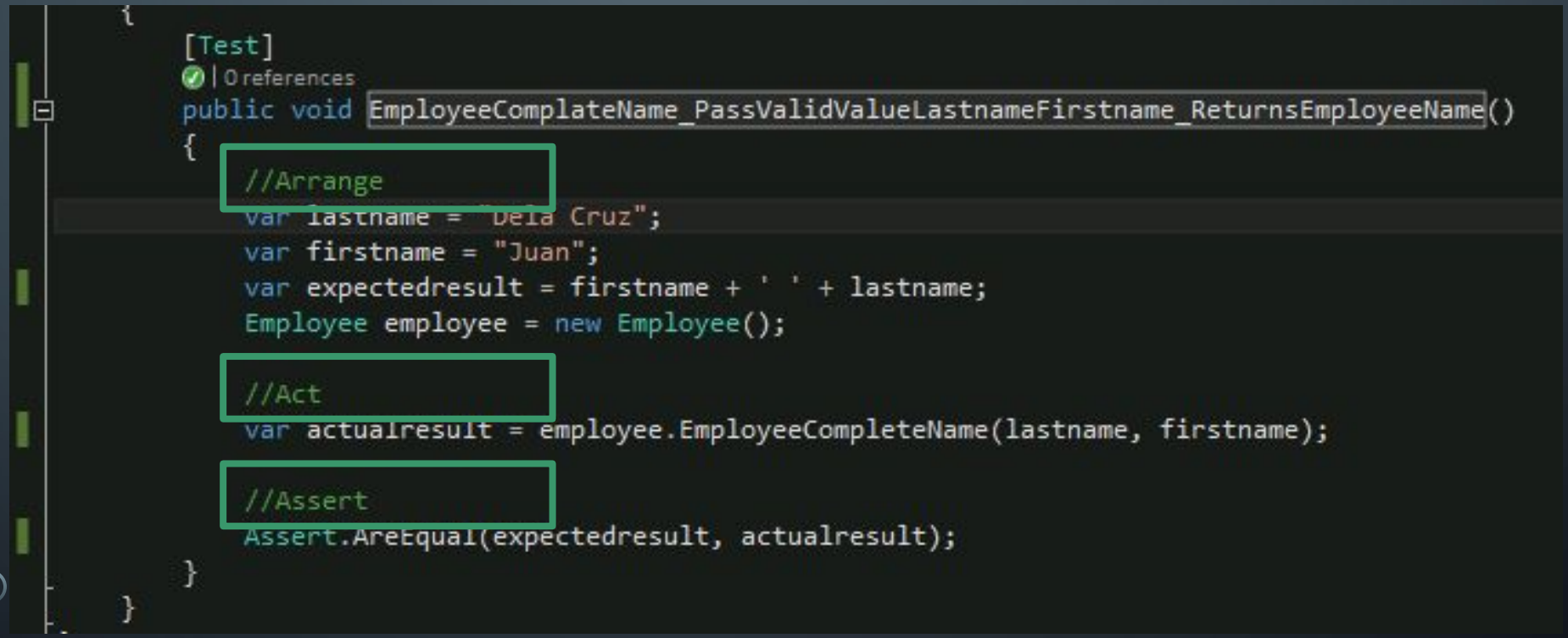
What makes a test **READ**able?

```
{
    [Test]
    0 references
    public void EmployeeComplateName_PassValidValueLastnameFirstname_ReturnsEmployeeName()
    {
        //Arrange
        var lastname = "Dela Cruz";
        var firstname = "Juan";
        var expectedresult = firstname + ' ' + lastname;
        Employee employee = new Employee();

        //Act
        var actualresult = employee.EmployeeCompleteName(lastname, firstname);

        //Assert
        Assert.AreEqual(expectedresult, actualresult);
    }
}
```

What makes a test **READ**able?

```
{
    [Test]
    0 references
    public void EmployeeComplateName_PassValidValueLastnameFirstname_ReturnsEmployeeName()
    {
        //Arrange
        var lastname = "Dela Cruz";
        var firstname = "Juan";
        var expectedresult = firstname + ' ' + lastname;
        Employee employee = new Employee();

        //Act
        var actualresult = employee.EmployeeCompleteName(lastname, firstname);

        //Assert
        Assert.AreEqual(expectedresult, actualresult);
    }
}
```

What makes a test **READ**able?



What makes a test **READ**able?

```
{
    [Test]
    0 references
    public void EmployeeComplateName_PassValidValueLastnameFirstname_ReturnsEmployeeName()
    {
        //Arrange
        var lastname = "Dela Cruz";
        var firstname = "Juan";
        var expectedresult = firstname + ' ' + lastname;
        Employee employee = new Employee();

        //Act
        var actualresult = employee.EmployeeCompleteName(lastname, firstname);

        //Assert
        Assert.AreEqual(expectedresult, actualresult);
    }
}
```

What makes a test **READ**able?

```
[Test]
public void ReadFromDataRecord_PassNonExistingCertProgIDRecord_ReturnsCertProgIDIsEmpty()
{
    //arrange
    Guid expectedResult = Guid.Empty;

    Mock<IDataRecord> _iDataRecord = new Mock<IDataRecord>();

    int fieldCount = 1;
    int fieldIdx = 0;
    string fieldName = "b_active";
    Boolean isActive = true;

    _iDataRecord.Setup(m => m.FieldCount).Returns(fieldCount);
    _iDataRecord.Setup(x => x.GetName(fieldIdx)).Returns(fieldName);
    _iDataRecord.Setup(d => d.GetBoolean(fieldIdx)).Returns(isActive);

    //act
    var actualResult = CertificationProgramUser.ReadFromDataRecord(_iDataRecord.Object);

    //assert
    Assert.AreEqual(expectedResult, actualResult.CertProgId);
}
```

What makes a test **READ**able?

```
[Test]
public void ReadFromDataRecord_PassNonExistingCertProgIDRecord_ReturnsCertProgIDIsEmpty()
{
    //arrange
    Guid expectedResult = Guid.Empty;

    Mock<IDataRecord> _iDataRecord = new Mock<IDataRecord>();

    int fieldCount = 1;
    int fieldIdx = 0;
    string fieldName = "b_active";
    Boolean isActive = true;

    _iDataRecord.Setup(m => m.FieldCount).Returns(fieldCount);
    _iDataRecord.Setup(x => x.GetName(fieldIdx)).Returns(fieldName);
    _iDataRecord.Setup(d => d.GetBoolean(fieldIdx)).Returns(isActive);

    //act
    var actualResult = CertificationProgramUser.ReadFromDataRecord(_iDataRecord.Object);

    //assert
    Assert.AreEqual(expectedResult, actualResult.CertProgId);
}
```

What makes a test **READ**able?

```
[Test]
public void ReadFromDataRecord_PassNonExistingCertProgIDRecord_ReturnsCertProgIDIsEmpty()
{
    //arrange
    Guid expectedResult = Guid.Empty;

    Mock<IDataRecord> _iDataRecord = new Mock<IDataRecord>();

    int fieldCount = 1;
    int fieldIdx = 0;
    string fieldName = "b_active";
    Boolean isActive = true;

    _iDataRecord.Setup(m => m.FieldCount).Returns(fieldCount);
    _iDataRecord.Setup(x => x.GetName(fieldIdx)).Returns(fieldName);
    _iDataRecord.Setup(d => d.GetBoolean(fieldIdx)).Returns(isActive);

    //act
    var actualResult = CertificationProgramUser.ReadFromDataRecord(_iDataRecord.Object);

    //assert
    Assert.AreEqual(expectedResult, actualResult.CertProgId);
}
```

What makes a test **READ**able?

```
[Test]
public void ReadFromDataRecord_PassNonExistingCertProgIDRecord_ReturnsCertProgIDIsEmpty()
{
    //arrange
    Guid expectedResult = Guid.Empty;

    Mock<IDataRecord> _iDataRecord = new Mock<IDataRecord>();

    int fieldCount = 1;
    int fieldIdx = 0;
    string fieldName = "b_active";
    Boolean isActive = true;

    _iDataRecord.Setup(m => m.FieldCount).Returns(fieldCount);
    _iDataRecord.Setup(x => x.GetName(fieldIdx)).Returns(fieldName);
    _iDataRecord.Setup(d => d.GetBoolean(fieldIdx)).Returns(isActive);

    //act
    var actualResult = CertificationProgramUser.ReadFromDataRecord(_iDataRecord.Object);

    //assert
    Assert.AreEqual(expectedResult, actualResult.CertProgId);
}
```

What makes a test **MAINTAIN**able?

1. A test must only test one thing; unless:
   - ➢ It is a utility method that will expect the same result but different inputs, use [TestCase] attribute
2. The dumber, the betterest
   - ➢ No Ifs/Else, Switch, loops, or any logic within a unit test!
   - ➢ What if there are bugs in your tests?
3. Know when to be or not be explicit with a test
   - ➢ Too brittle, it will be harder to refactor. Too loose, it's like there was no point to test it at all!
4. As much as possible, only test the logic of the method you are testing
   - ➢ If it is the only public modifier in the class, it could be a bad design

Two Asserts, One Cup

```
[Test]
public void DeserializeTest()
{
    var serializedData = SerializeLocalizedData.Serialize(_localizedDataList);
    var localizedDataList = SerializeLocalizedData.Deserialize(serializedData, null);

    DeserializeAssert(localizedDataList);

    var binarySerializedData = SerializeLocalizedData.BinarySerialize(_localizedDataList);
    localizedDataList = SerializeLocalizedData.Deserialize(null, binarySerializedData);

    DeserializeAssert(localizedDataList);
}
```

Allow me add more bugs to your code by adding more bugs in your test…

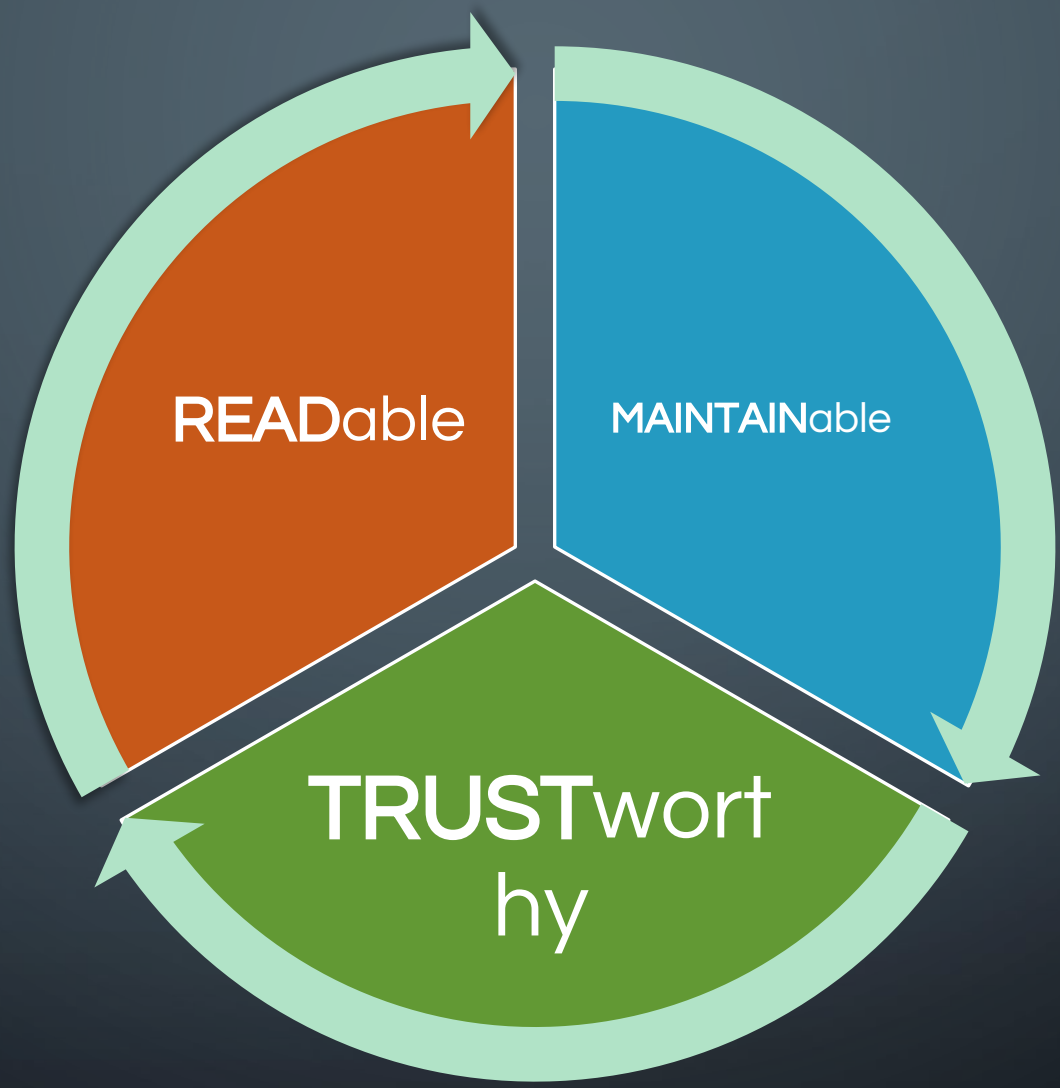
```
[Test]
public void GetCurrencySymbol_Test()
{
    var currencies = Currency.GetCurrencies().Cast<CurrencyListItem>();

    foreach(var currency in currencies)
    {
        var symbol = Currency.GetCurrencySymbol(currency.ID);
        Assert.AreEqual(currency.Symbol, symbol);
    }
}
```

```
[Test]
public void GetDefaultCurrency_Test()
{
    var expected = Currency.GetCurrencies()
        .Cast<CurrencyListItem>()
        .FirstOrDefault(x => x.Default);

    var actual = Currency.GetDefaultCurrency();

    Assert.AreEqual(expected.ID, actual.ID);
}
```

```
[TestCase(true, false, false, true)]
[TestCase(false, true, true, true)]
public void set_EventEnd_False_ChangedPropertyTrue(bool eventEndInitialValue, bool eventEnd, bool expectedEventEnd, bool expectedChange)
{
    Guid id = new Guid("cc244b24-71e7-4b48-b1fd-eef3aeb319d5");
    string filename = "filename";
    bool showInPortal = true;
    bool visibleWithoutRegistration = true;
    EventFile eventFile = new EventFile(id, filename, showInPortal, visibleWithoutRegistration, EventFileType.Empty);
    eventFile.EventEnd = eventEndInitialValue;
    eventFile.EventEnd = eventEnd;

    Assert.AreEqual(eventEnd, eventFile.EventEnd);
    Assert.AreEqual(expectedChange, eventFile.Changed);
}
```

What makes a test **TRUST**worthy?

1. Have you said/heard the following:
   - "Let's debug so we can figure out what's wrong in the unit test"
   - "No, it's meant to break sometimes.."
   - "There's a particular order that's why it failed"
   - "What is a Test Review?"
   - "If failed in Team City but it is okay in QA"
   - "Tang ina, unit test na naman!"
- If you've heard on of these, then our test do not achieve the goal.

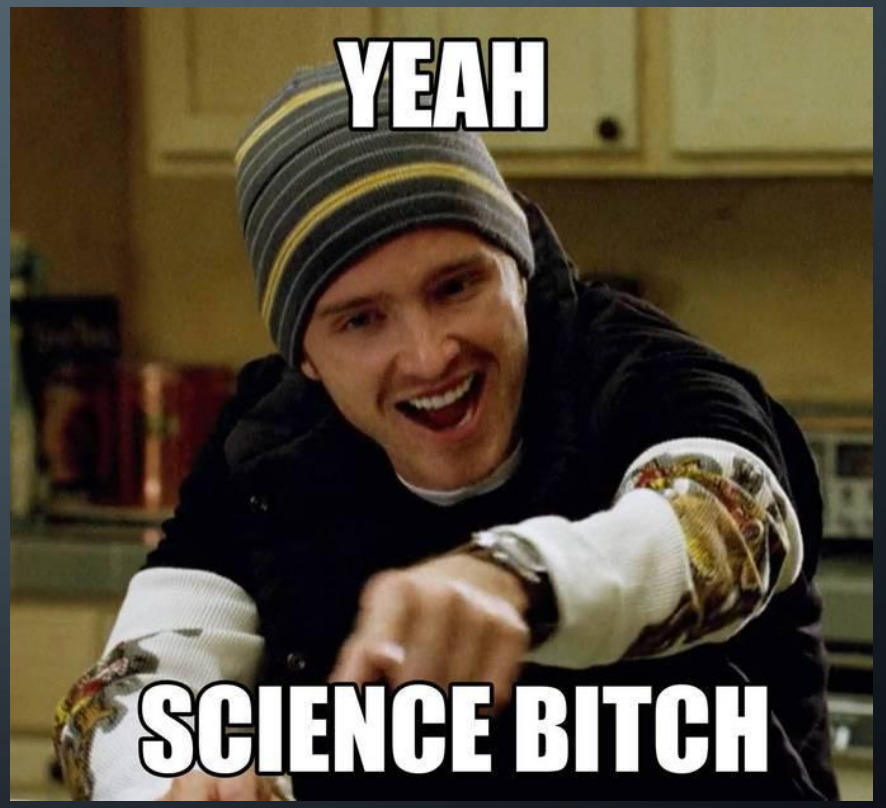

READable
MAINTAINable
TRUSTwort
hy

Common pitfalls in writing Unit Tests

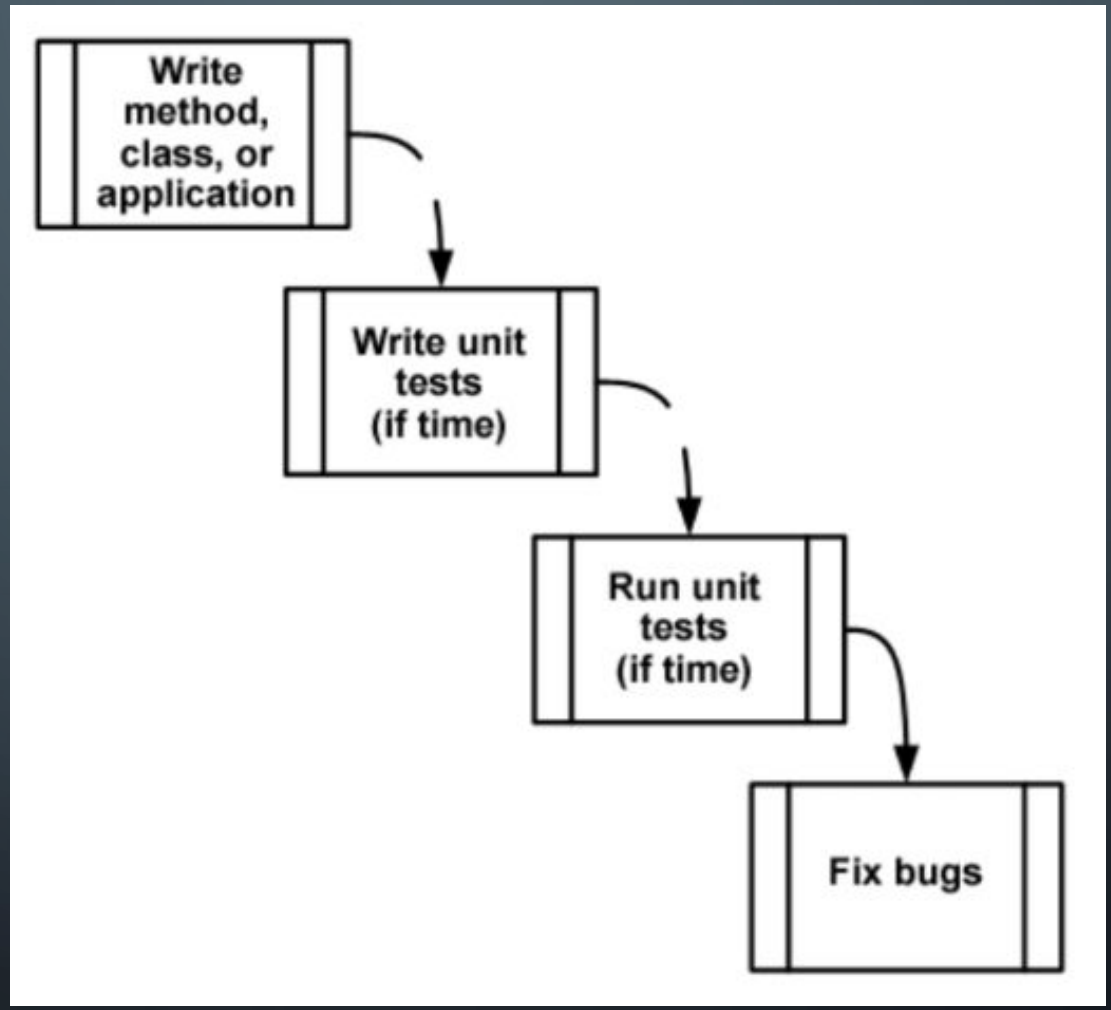
- Inability to differentiate Unit from Integration Tests
- Over complicate unit tests
- Unit test become a form of self expression
- Writing Unit tests at the end of coding
- Not trying at all.

COMPUTER SCIENCE NEEDS MORE SCIENTISTS!

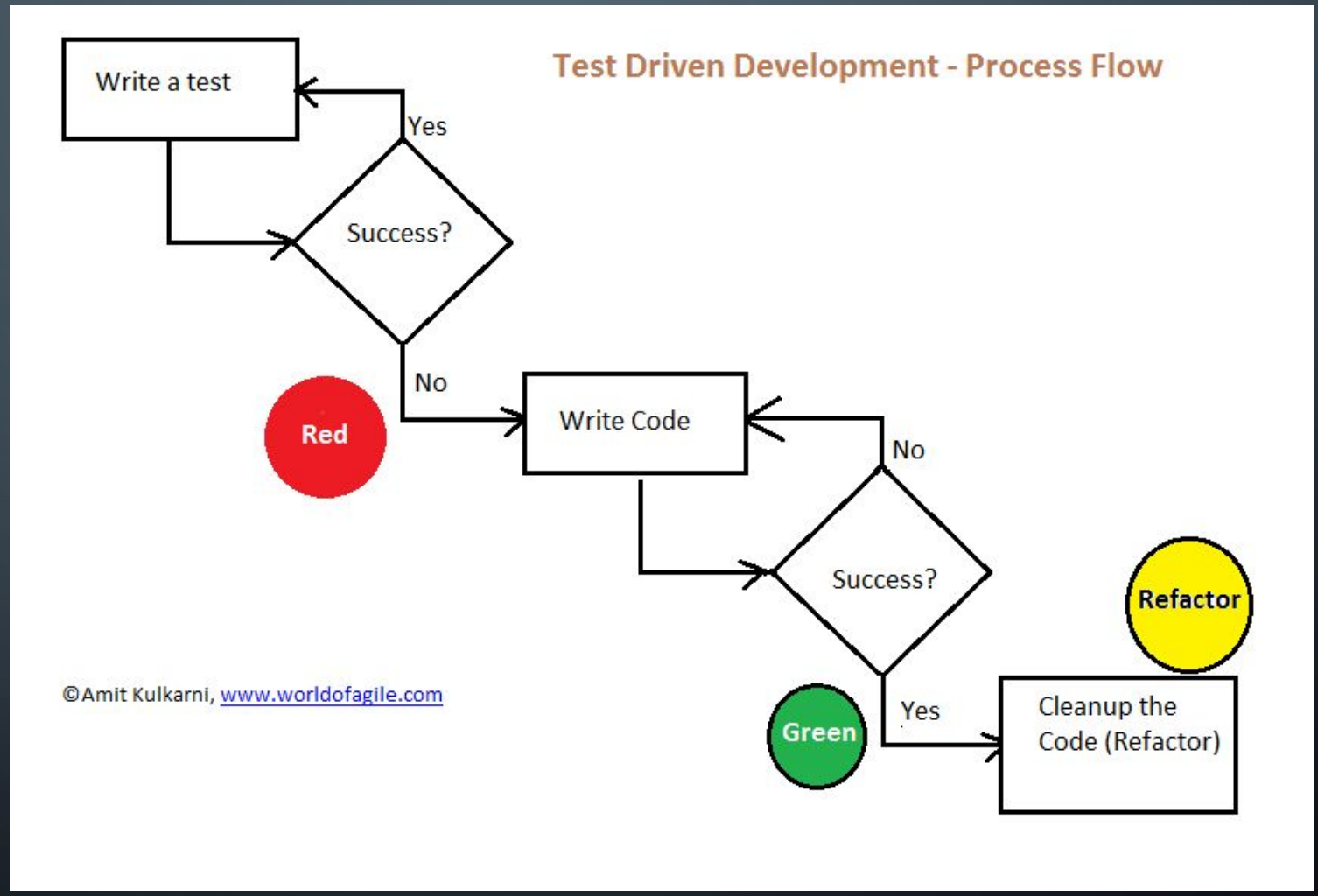
- Question, Hypothesis, Experimentation, Data gathering Conclusions.
- In any science experiment, in order to get consistent and observable results, we need to remove external factors to create consistent base lines.
- This is called “isolating a variable”



What usually happens



Correct Test Driven Development Flows



Let’s write a simple unit test

- Write a simple unit test using Nunit
- Understand the different Attributes in Nunit
- Apply Best Practices learned.