TEST-DRIVEN DEVELOPMENT

OVERVIEW

Testing Overview

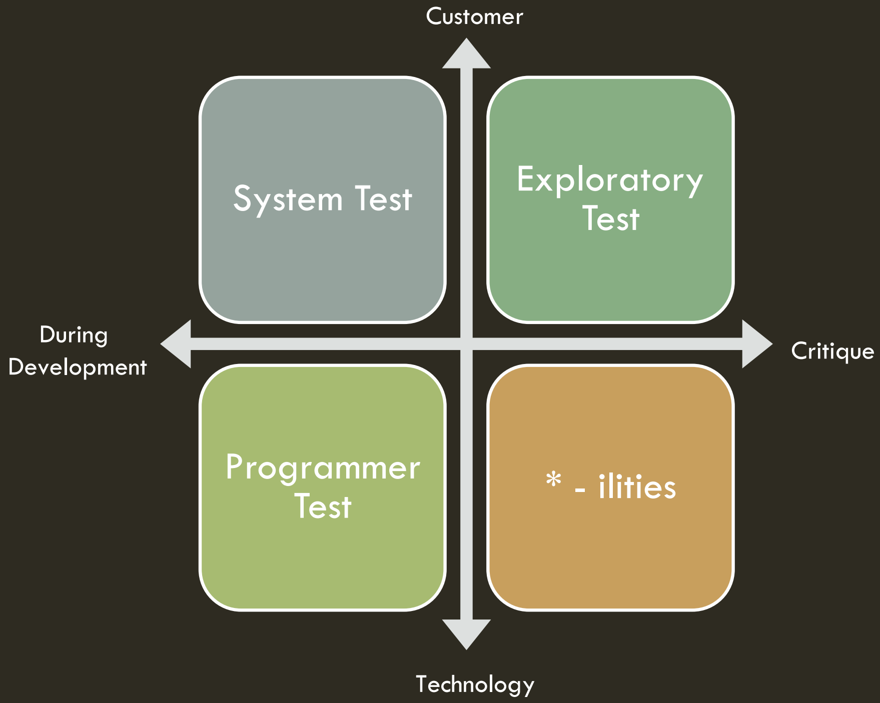
- Concepts
- Tools
- Patterns & Practices
- Writing Tests

Test-First Programming

Refactoring

TEST TYPES

Brian Marick – http://www.exampler.com



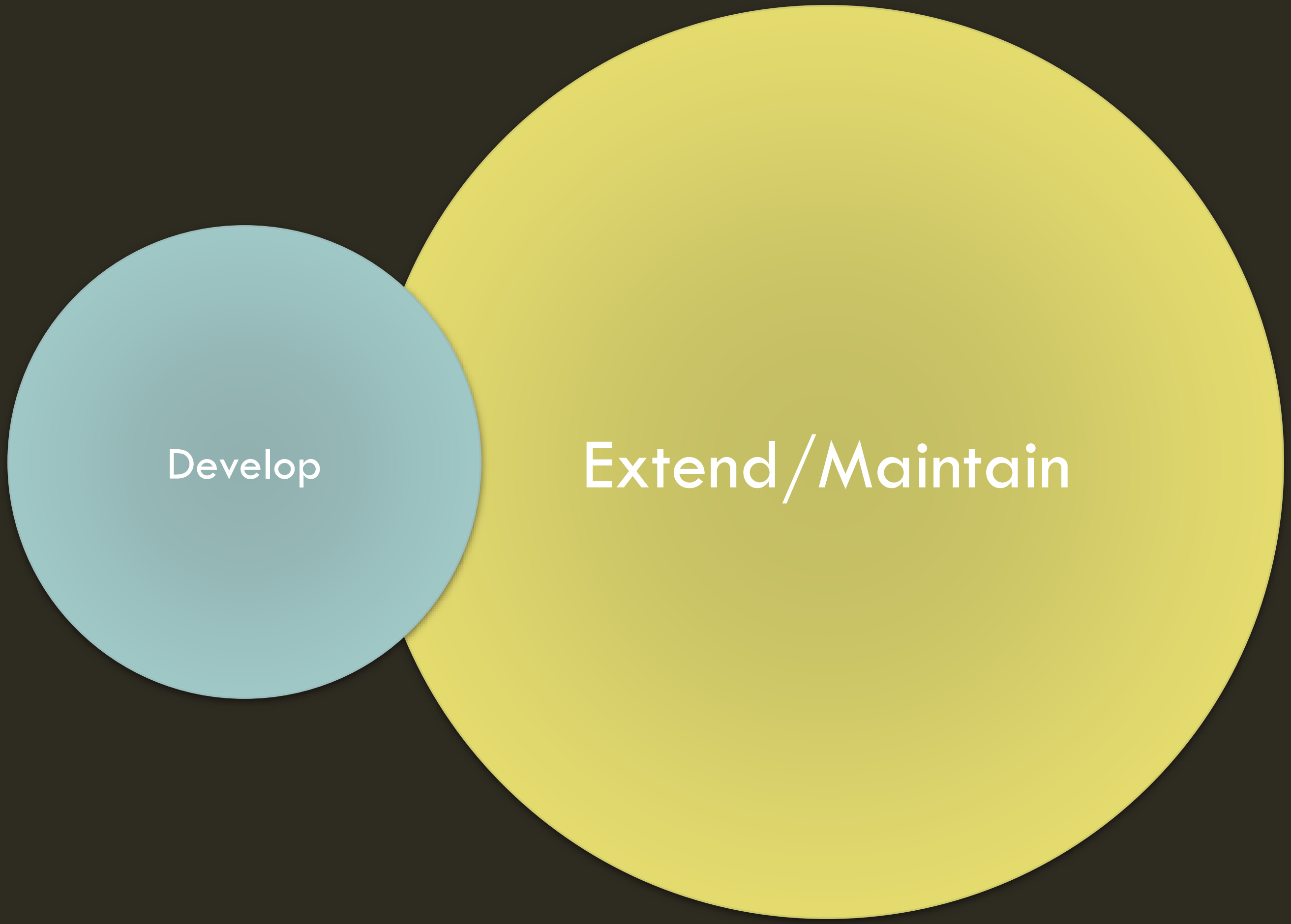
WHY DO PROGRAMMER TESTING?

“*There is no such thing as done. Much more investment will be spent modifying programs them developing them initially*” [Beck]

“*Programs are read more often than they are written*” [Beck]

“*Readers need to understand programs in detail and concept*” [Beck]

TOTAL DEVELOPMENT COST



TESTING STYLES

State-Based

Exercise the System Under Test (SUT)

Determine whether the exercised method worked correctly by examining the state of the System Under Test (SUT)

Stub Dependencies for Isolation

Interaction

Create/Setup expectations

Exercise the SUT

State-Based Verification and Behavior Based verification

Mock Dependencies for Isolation

TESTING STYLES

White-Box

A method of testing software that tests internal structures or workings of an application, as opposed to its functionality

An Internal perspective of the SUT is used to elaborate test cases

How: Choose inputs to exercise paths through the code and determine the appropriate outputs.

Black-Box

A method of testing that tests functionality of an application as opposed to it's internal structure

Knowledge of the application's code and programming knowledge in general is not required.

Test cases are built around specifications and requirements.

These tests can be functional or non-functional, usually functional.

PROGRAMMER TESTS SUMMARY

Technology Focused

Support Development

Primarily State-Based with some interaction testing in special cases

White-Box Testing with a caveat around how much of the implementation is tested directly

TOOLS

TOOLS

xUnit Frameworks

- Tests are written in the programming language of the System Under Test
- Self verifying tests (i.e. no human intervention is required to interpret the results)
- .NET Examples: xUnit.net, NUnit, MSTest (less likely)
- JavaScript: QUnit

Mocking Frameworks

- Moq
- NMock

PATTERNS AND PRACTICES

3A PATTERN

3A PATTERN

Attributed to Bill Wake (http://xp123.com)

- Arrange – Setup the test harness
- Act – Run the test
- Assert – Check the results

Let's look at an example!

A TYPICAL TEST

```
[Fact]
public void TopDoesNotChangeTheStateOfTheStack()
{
    Stack<string> stack = new Stack<string>();
    stack.Push("42");


    string element = stack.Top;


    Assert.False(stack.IsEmpty);
}
```

3A PATTERN

```
[Fact]
public void TopDoesNotChangeTheStateOfTheStack()
{
    Stack<string> stack = new Stack<string>();
    stack.Push("42");
```

Arrange

```
    string element = stack.Top;


    Assert.False(stack.IsEmpty);
}
```

3A PATTERN

```
[Fact]
public void TopDoesNotChangeTheStateOfTheStack()
{
    Stack<string> stack = new Stack<string>();
    stack.Push("42");


    string element = stack.Top;                        Act


    Assert.False(stack.IsEmpty);
}
```

3A PATTERN

```
[Fact]
public void TopDoesNotChangeTheStateOfTheStack()
{
    Stack<string> stack = new Stack<string>();
    stack.Push("42");


    string element = stack.Top;


    Assert.False(stack.IsEmpty);                     Assert
}
```

3A SUMMARY

Benefits

- Readability
- Consistency

Liabilities

- More Verbose
- Might need to introduce local variables

Related Issues

- One Assert per Test?

WORK INSIDE OUT

WHAT DO YOU WANT TO TEST?

```
[Fact]
public void TopDoesNotChangeTheStateOfTheStack()
{
    Stack<string> stack = new Stack<string>();
    stack.Push("42");


    string element = stack.Top;


    Assert.False(stack.IsEmpty);
}
```

WHAT SETUP IS NEEDED?

```
[Fact]
public void TopDoesNotChangeTheStateOfTheStack()
{
    Stack<string> stack = new Stack<string>();
    stack.Push("42");


    string element = stack.Top;


    Assert.False(stack.IsEmpty);
}
```

HOW DO I VERIFY CORRECTNESS?

```
[Fact]
public void TopDoesNotChangeTheStateOfTheStack()
{
    Stack<string> stack = new Stack<string>();
    stack.Push("42");


    string element = stack.Top;


    Assert.False(stack.IsEmpty);
}
```

SINGLE RESPONSIBILITY (!SINGLE ASSERT)

SINGLE RESPONSIBILITY

Derived from Single Responsibility Principle - "***There should never be more than one reason for a class to change.***"

A test should focus on the test of a single concept

Multiple asserts are acceptable if required to verify the concept

SINGLE RESPONSIBILITY EXAMPLE

```
[Fact]
public void ToArray()
{
    List<string> list = new List<string>()
                 {"aa", "bb", "cc"};

    string[] array = list.ToArray();

    Assert.Equal(3, array.Count());
    Assert.Contains("aa", array);
    Assert.Contains("bb", array);
    Assert.Contains("cc", array);
}
```

YOU AREN'T GOING TO NEED IT!

Defer Decisions until the last responsible moment

Implement things when you actually need them, never when you just foresee that you need them

LEAVE NO TRACE

The System Under Test should be returned to the state that existed prior to running the tests

Simple for in-memory objects

Difficult for interacting with external systems

KEEP YOUR TESTS CLOSE

KEEP YOUR TESTS CLOSE

Tests are equivalent to production code

Solves visibility problems

Caveats

- Should you ship your tests?
- Various Tool Issues
- If No, how do you separate the tests from the code when you release?

Be Self-Contained

BE SELF-CONTAINED

Readability

Test isolation

Caveats

- Duplicated initialization code

BE SELF-CONTAINED – EXAMPLE

```
[Fact]
public void ToArray()
{
    List<string> list = new List<string>()
                 {"aa", "bb", "cc"};

    string[] array = list.ToArray();

    Assert.Equal(3, array.Count());
    Assert.Contains("aa", array);
    Assert.Contains("bb", array);
    Assert.Contains("cc", array);
}
```

STATE-BASED TESTING

Test a constructor

Test a method that returns a value

Test a method that throws an exception

Test a method that does not return a value

TEST A CONSTRUCTOR

Don’t test the platform…

Constructors don’t return values - Look for changes to the state to verify the object is constructed correctly

Test Invalid Input Parameters

EXERCISE

Write tests for the following two constructors:

List<T> - the default constructor for the List<T> class

List<T>(int initialCapacity) - this constructor sets the initial capacity of the list

Test a method that throws an Exception

TEST A METHOD THAT THROWS AN EXCEPTION

Boundary Conditions

Watch/Test for Cleanup issues

Be specific

RECORD THE EXCEPTION

```
[Fact]
public void PopEmptyStack()
{
    Stack<string> stack = new Stack<string>();

    Exception ex = Record.Exception(() => stack.Pop());

    Assert.IsType<InvalidOperationException>(ex);
}
```

BE SPECIFIC

```
[Fact]
public void PopEmptyStack()
{
    Stack<string> stack = new Stack<string>();

    Exception ex = Record.Exception(() => stack.Pop());

    Assert.IsType<InvalidOperationException>(ex);
    Assert.Equal("Stack empty.", ex.Message);
}
```

TEST A METHOD THAT RETURNS A VALUE

Simplest due to cause and effect

Look for boundary conditions that cause exceptions

Watch/Test for side effects

EXERCISE

Write facts for Int32.Parse(string s)

Exceptions

- ArgumentNullException – when the passed in string is a null reference
- FormatException is thrown when the passed in string is not in the correct format
- OverflowException is thrown when the passed in string is less than MinValue or greater than MaxValue

PROGRAMMER TEST SUMMARY

Test modules in isolation

Use Stubs or Mock Objects to break dependencies

Fast and can run frequently without human intervention

Be Self-Contained

Leave no trace…

Easy to reproduce infrequent errors

Can achieve high code coverage