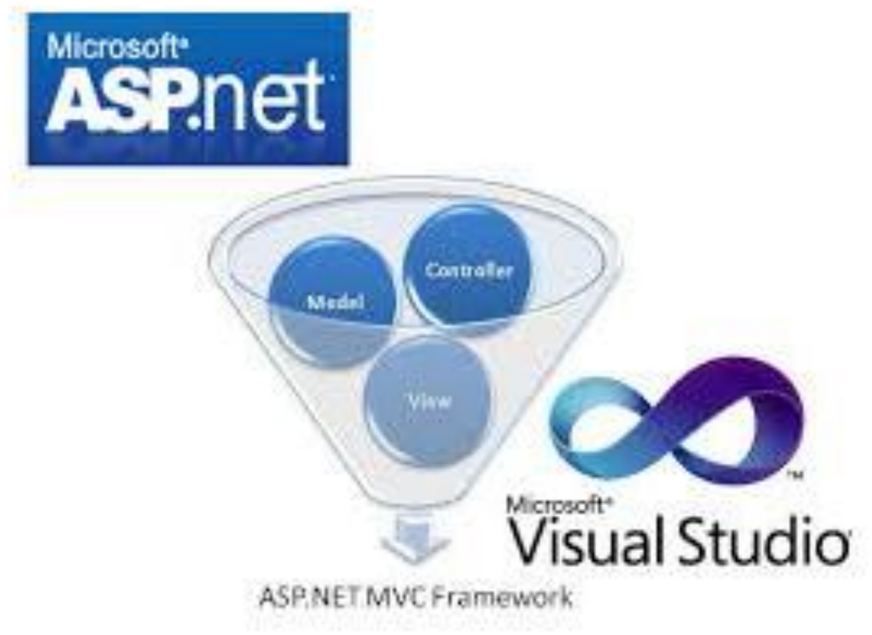


C#

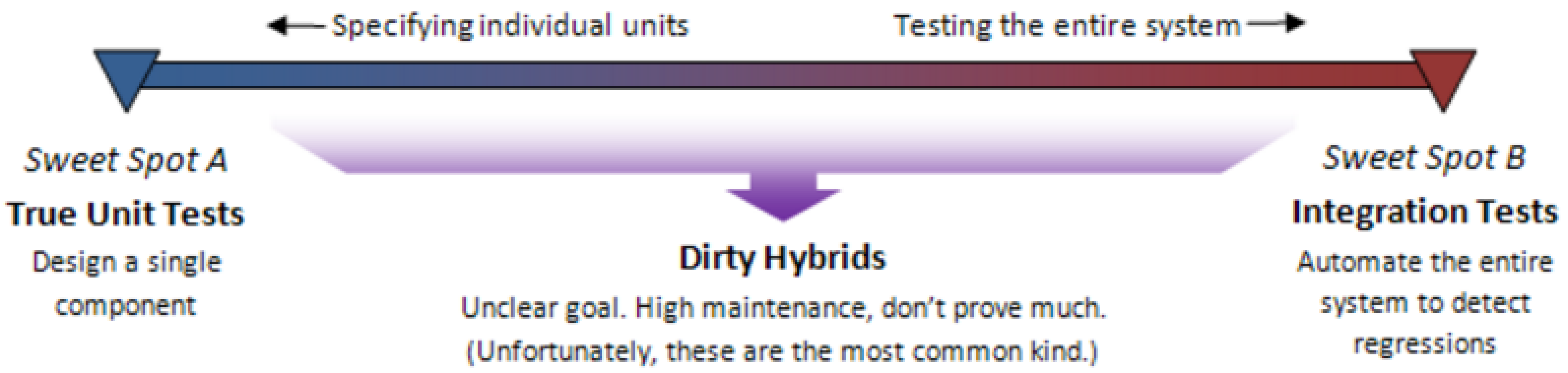
8. TDD WITH ASP.NET MVC

Inspection at the point of creation

Testing Methodologies

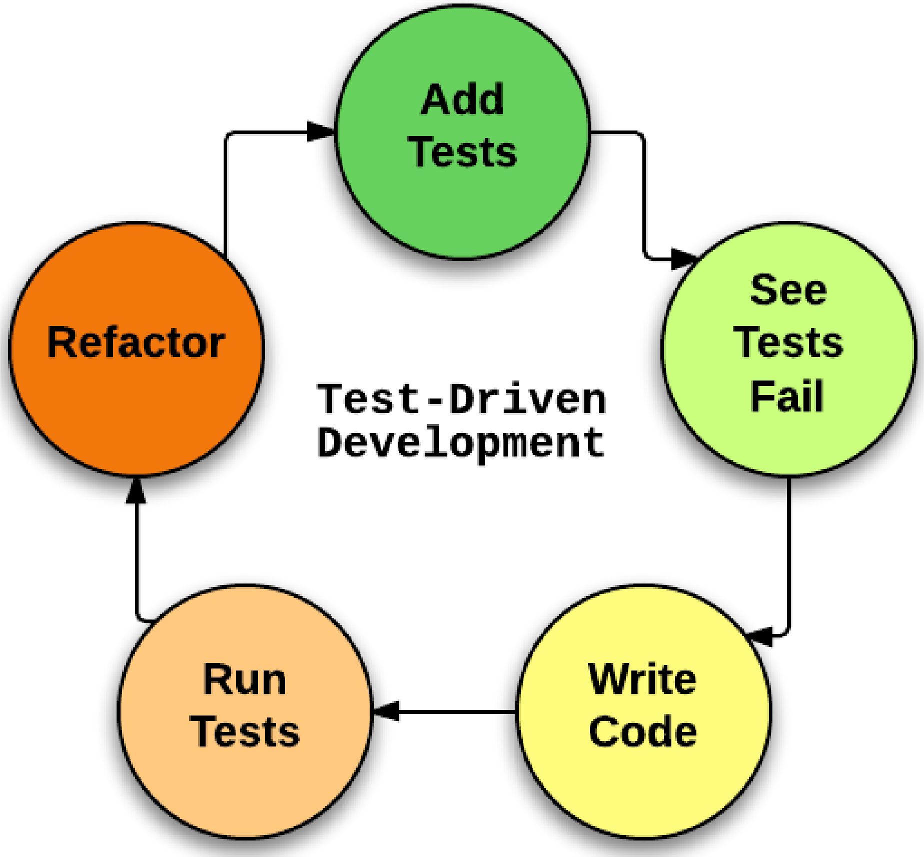
- Code First:
  - Write the **code** first
    - Then write the **test** method
- Test First [TDD]
  - Write the **test**
    - Then write the **code**
- Behaviour-Driven Development [BDD]
  - Based on TDD, but it also address some of the issues that using TDD poses, where to start, what to test. How much to test in a cycle and what to call in the tests. It focuses on behavioural requirements of the tested units

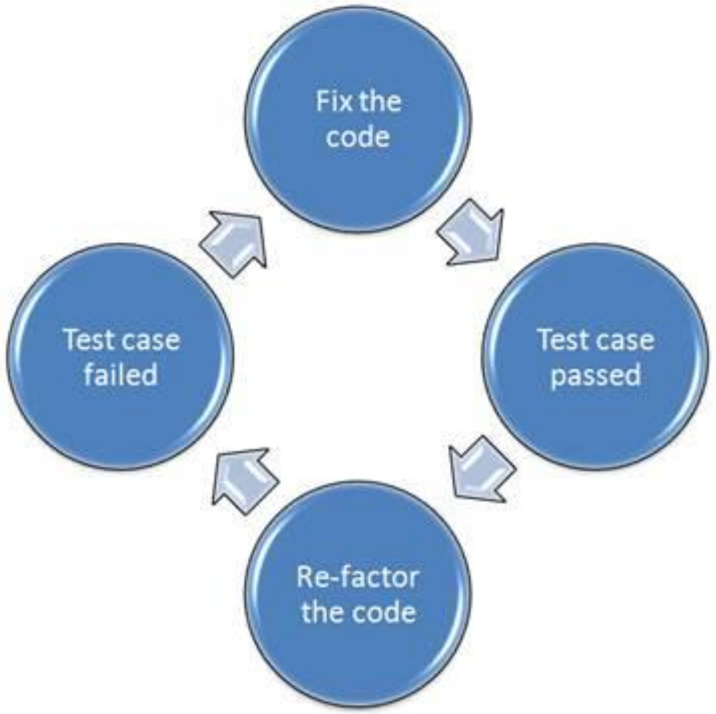
Overview


Specifying individual units
Testing the entire system
Sweet Spot A
True Unit Tests
Design a single component
Dirty Hybrids
Unclear goal. High maintenance, don't prove much.
(Unfortunately, these are the most common kind.)
Sweet Spot B
Integration Tests
Automate the entire system to detect regressions

http://blog.stevensanderson.com/2009/08/24/writing-great-unit-tests-best-and-worst-practises/

TDD LIFE CYCLE





TDD

- Validates your Design
- Enables Change
- Provides rapid feedback
- Forces Constant Integration
- Written by Developers to first test a functionality to be implemented.
- Automates the process
- Improves the quality of implementation
- Improves the Quality of design
- Highly Maintainable Code
- Well-designed applciation
- Readability, Flexibility and Simplicity in processing

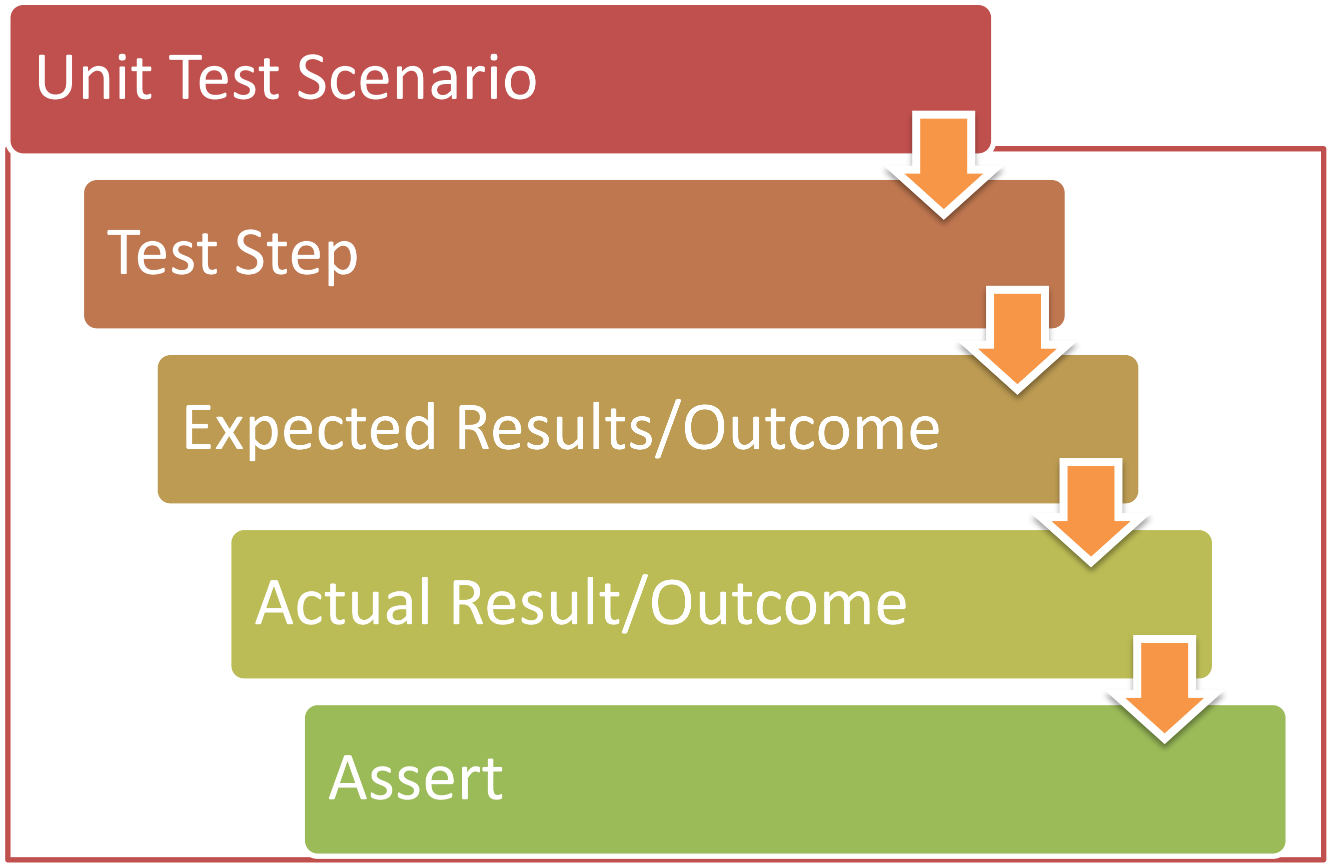
Some Design Principles

- Loosely coupled, highly cohesive
- Dependency Injection/Inversion of Control (IoC)
- Single Responsibility Principle
- Open/Closed Principle
- Law of Demeter (“Tell don’t ask”)
- Modularization
- Separation of Concerns
- Encapsulation
- Favour Immutability
- Domain Driven Design
- Decentralized management and decision making
- Strict Dependency Management
- Clean Code
- Eliminate Duplication/Code Reusability

Types of Tests

Unit Test	•Single Class (usually) •No external dependencies
User Interface Tests	
Integration Tests	•Multiple classes working together •Will use a real database
Acceptance Testing	•Can Encompass “System” or “Functional” tests •End-to-End is desirable when feasible
Performance Testing	
Accessibility Testing	
Security Testing	
End-to-End Tests	•UI tests •Interact like a real user

Unit Test Scenario



Inversion of Control

- Inversion of Control or more specifically dependency injection is a design pattern in which configuration is separated from use
- Dependencies are configured at runtime rather than at compile time by an assembler.
- This is achieved through the use of interfaces

Inversion of Control

- Why you should use it
  - IOC/DI is highly recommended when building software especially an MVC website because
    - Flexibility of configuration
    - Reusable configuration (registries)
    - Manages Object Lifetime/LifeCycle (singleton, request, transient, etc)
    - Hides implementation details/Allows switching out of implementing technologies (ie. Switch from Entity Framework to Nhibernate)
    - Decoupled classes makes for easy testing

When Not to Use Inversion of Control

- When speed and/or size is of the upmost importance, such as in a real-time system.
- IoC does add some lag to the system (varies based on IoC container, but can be as much as 10x slower)
- Also some IoC containers do not work in the more limited class space of Silverlight, Windows Phone or in places where Reflection related classes are unavailable.
- IoC Demo /Tools
  – **StructureMap, Unity(MS),AutoFac, HaveBox, SimpleInjector, StyleMVVM**

Why MVC is Better for Unit Testing

- WebForms aggregated and integrated the logic and display of an application
- MVC draws very sharp distinctions between the model and anything related to the model, the view is the display of data and very difficult to test with unit testing and the controller, responsible for tying the model to the view.
- **To what degree we can test the controller?**

Testing Tools

- Visual Studio 2012/2013
- Nunit/Xunit/MSTest
- JustMock
- Xania-AspNet.Simulator –Pre
- FluentAssertions
- Automated Build Tool
- Mocking Framework
- Build and Continuous Integration Servers
- Test Coverage Tools
- Code Quality Tools
- Shoudly
- Fixie
- WebGrease

Unit Testing and TDD

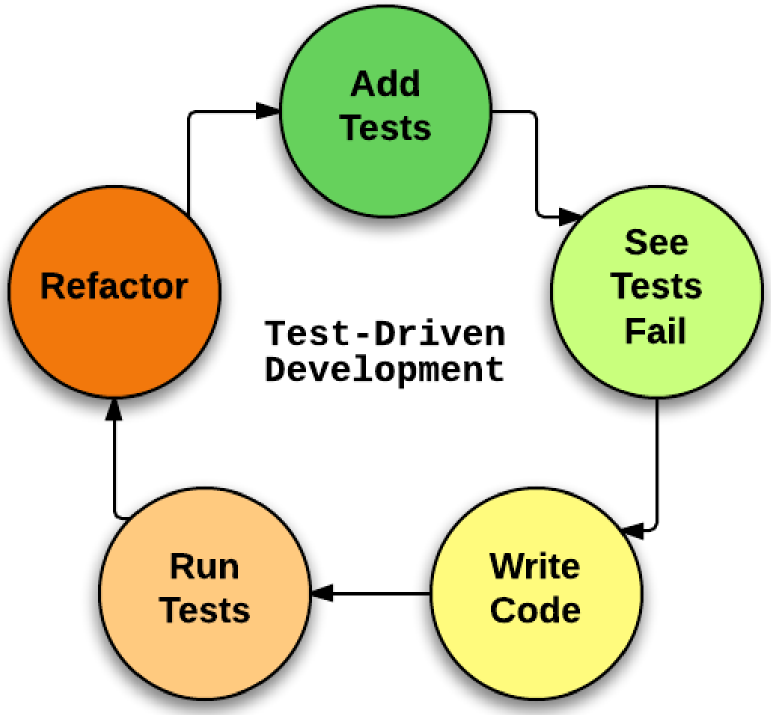
A unit test confirms functionality of a small unit of functionality or a component in a larger system

- Unit Testing
  - Test a single unit – a class or (better) a method
  - Test in isolation
- TDD
  - Uses unit testing to drive the design
  - Test-first

- Test one feature only
- Run the test –see it fail
  - Why this is important
- Fix only enough to make it pass
  - Why this important
- Run the test and see it succeed
- Refactor and retest

Test Driven Development

- Unit Tests
- Test First
- Red Green Refactor
  - Red – Create a failed test
  - Green – Write enough to pass the test
  - Refactor – Clean up the Code
  - Repeat



Robert C. Martin’s 3 LAWS for TDD

- Write no production code until you have written a failing test
- Write no more in your unit test than enough to make it fail
- Write no more production code than enough to make it pass

Mocking

- Creating fake objects for you
- Nothing you can't do manually
- Set and inspect values on a fake object
- Inspect method calls and args on a fake object

- Mocking comes in a number of flavours
  - Stub
    - Simple mocks that just return a single value
  - Fakes
    - Have a little bit of logic in them
  - Full Mocks
    - Can really represent the interactions with object that you are mocking.

Mocking

- **AAA**
  - **Arrange**
  - **Act**
  - **Assert**
- Remember, you are not testing the mocked object
- You are using the mock object to test your other code
- The mock object needs to be responsive to allow that testing
- **JUSTMOCK, MOCK**

- What can you Mock?
  - Interfaces
  - Virtual Methods
  - Abstract Methods
  - Properties
  - Static Classes, Methods and properties
  - LINQ Queries
  - Non-virtual methods and properties
  - Non-public members and types

Stub vs Mock

Stub

- Get/Set Properties
- Set method return value
- Test **State**

Mock

- Check method calls
- Check arguments used
- Test **interactions**

RhinoMocks https://www.hibernatingrhinos.com/oss/**rhino-mocks**
Moq – https://github.com/moq/moq4
TypeMock – www.**typemock**.com/
JustMock www.telerik.com/products/mocking.aspx

Assertion

- Assertion is a boolean expression at a specific point in a program which will be true unless there is a bug in the program.
- A test assertion is defined as an expression, which encapsulates some testable logic specified about a target under test

- Benefits of Assertions
  - Used to detect subtle erros which might go unnoticed
  - Detect errors sooner after they occur
  - Make a statement about the effects of the code that is guaranteed to be true

Asserts

- Assert what you expect to be true
- Only one “logical”assert per test

Equality	AreEqual, AreNotEqual
Identity	AreSame,AreNotSame, Contains
Condition	IsTrue,IsFalse, IsNull, IsNotNull, ISNan, ISEmpty, IsNotEmpty
Type	IsInstanceOf<T>, IsNotInstanceOf<T>,IsAssignableFrom<T>
Exception	Assert.Throws
String	Contains, StatsWith, EndsWith,

Limitations of assertions

- Assertions themselves may fail to report a bug that exists
- Reporting an error when it does not exist
- Can lead to other side effects
- Can take time to execute if it contains errors and occupies memory as well
- If an assertion is failing due to one or the other reason, the consequence of the same can be severe.
- An assertion could elevate to a stumbling block which might result in test failure

Expected Outcome & Actual Outcome

Expected Outcome

- Each test case has an expected outcome against which the actual outcomes are compared.
- The deviation, if any, is known as defect

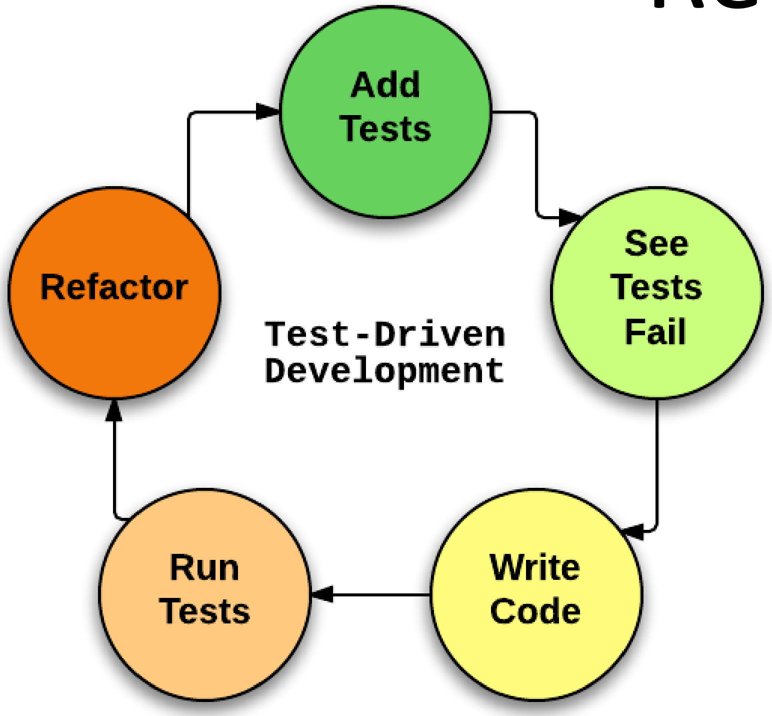
Actual Outcome

- Actual Result, which a tester gets after performing the test.
- It is always documented along with the test case during the test execution phase.
- After performing the tests, the actual outcome is compared with the expected outcome and the deviation are noted.
- The deviation is known as defect.

Dependency Injection

- Two primary forms of dependency injection
  - Constructor injection
    - Where you insert your mock into constructor of the object
  - Property injection
    - Where you feed your mock to a specific property

Refactoring



Classes = nouns
Methods = verbs
Collections are plural

Functions should be small

Don't Repeat Yourself

- Single Responsibilty Principle
  - Exactly one reason to change
  - One responsibility per method/class
- Open/Closed
  - Open for extension, closed for modification
  - Derive but don't modify internals
- Liskov Substitution Principle
  - Ability to replace instances with their subtype
- Interface Segregation
  - No client should be forced to depend on methods it does not use
  - Small interface
- Dependency Inversion
  - Depend on abstractions, not concrete implementations

Refactoring Tests

- FAST
  - If it isn’t fast you won’t run it
  - If you don’t run it, it has zero value
- Independent
  - Order must not matter
  - No dependecies between tests
- Repeatable
  - Running the same test twice should yield the same answer
- Self-validating
  - Tests return Booleans:Pass or Fail, no ambiguity
- Timely
  - Written just before production code

Summary

- We have presented the basic concepts behind test driven development.
- How to perform simple unit tests in asp.net mvc