Unit Testing

Effective Unit Testing with MsTest/NUnit

Trainer: Venkat Shiva Reddy

Topics

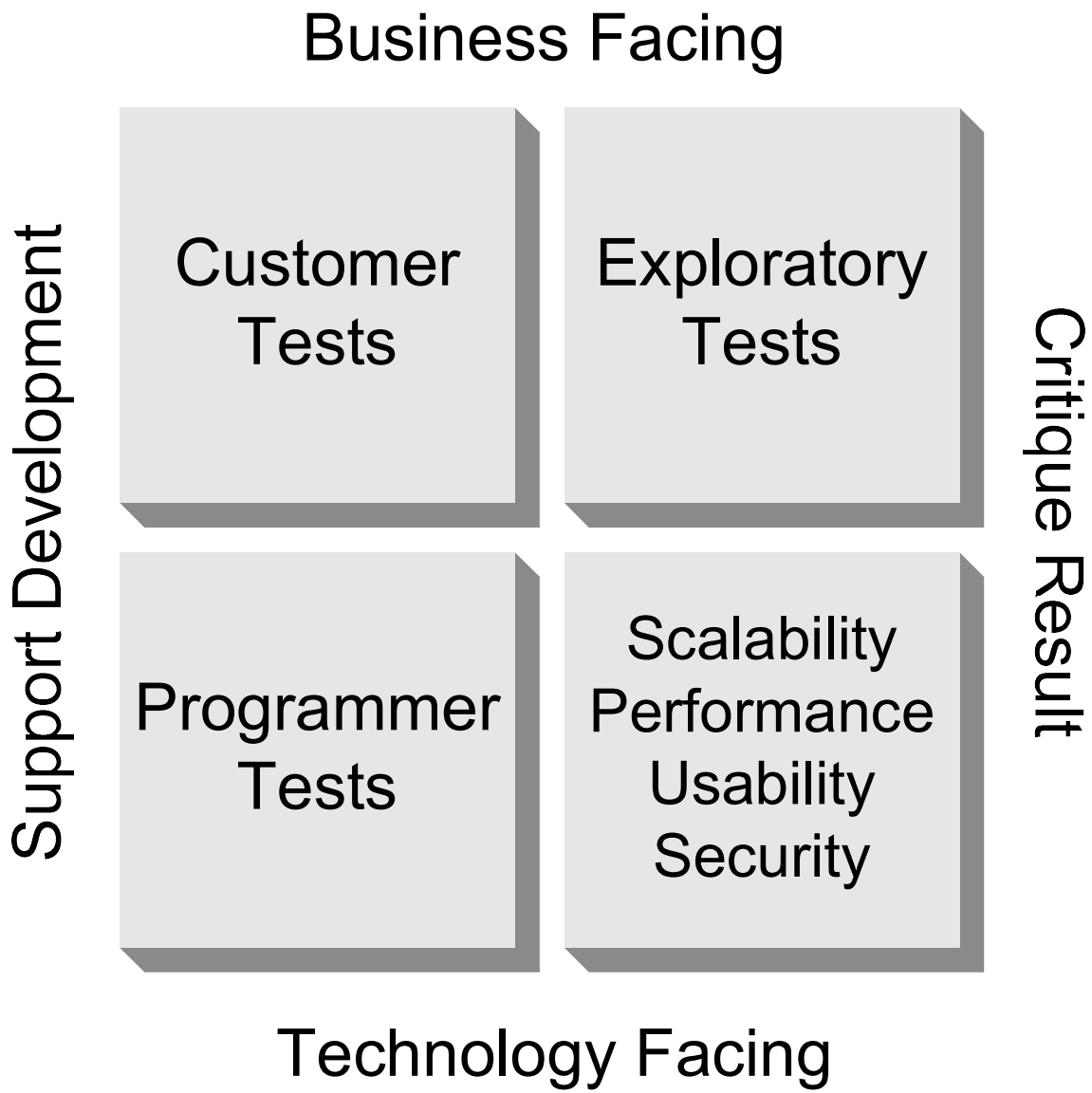
- What is Unit Testing?
- Typical Unit Testing Problems
- Best Practices for Effective Unit Testing
- Tool Demo
- Asserts
- Exceptions
- What to test for?
  - RIGHT BICEP
- Characteristics of good testing
- Mocks and stubs
- Design and Testing

Unit Testing

- Unit Testing is code that
  - Is written by developers, for developers
  - Exercises a small, specific area of functionality
  - Helps “prove” that a piece of code does what the developer expects it to do
  - For example, you might delete a pattern of characters from a string and then confirm that they are gone

Unit Testing

- **Marick's Four Quadrants of Unit Testing**



Unit Testing

- Unit Testing fits into the Programmer tests category here
- It is not Acceptance Testing [Functional Testing]
- Nor is it Performance or Scalability Testing
- Used to test the code written by the user in-order to ensure compliance with the requirement

Unit Testing

- **Why Unit Testing ?**
  - It will make your life easier
  - Better code
  - Better designs
  - Code is easier to maintain later
  - Confidence when you code

Unit Testing

- **Common Excuses for not testing**
  - I'm not a tester!
  - It takes too much time.
  - It takes too long to run the tests.
  - I don't know how to test it.
  - I don't really know what it is supposed to do, so I can't test it.
  - But it compiles! It doesn't need tests.

Best Practices - Unit Testing

- Start your development activities by writing down a list of things you want to test
- You will often think of a test while writing another one. When you do, add it to the list.
- Review your list frequently
- Test-driven development (TDD) is a proven way of improving quality*
- TDD's main objective is to aid programmers and customers during the development process with unambiguous requirements
- Use good tools

NUnit

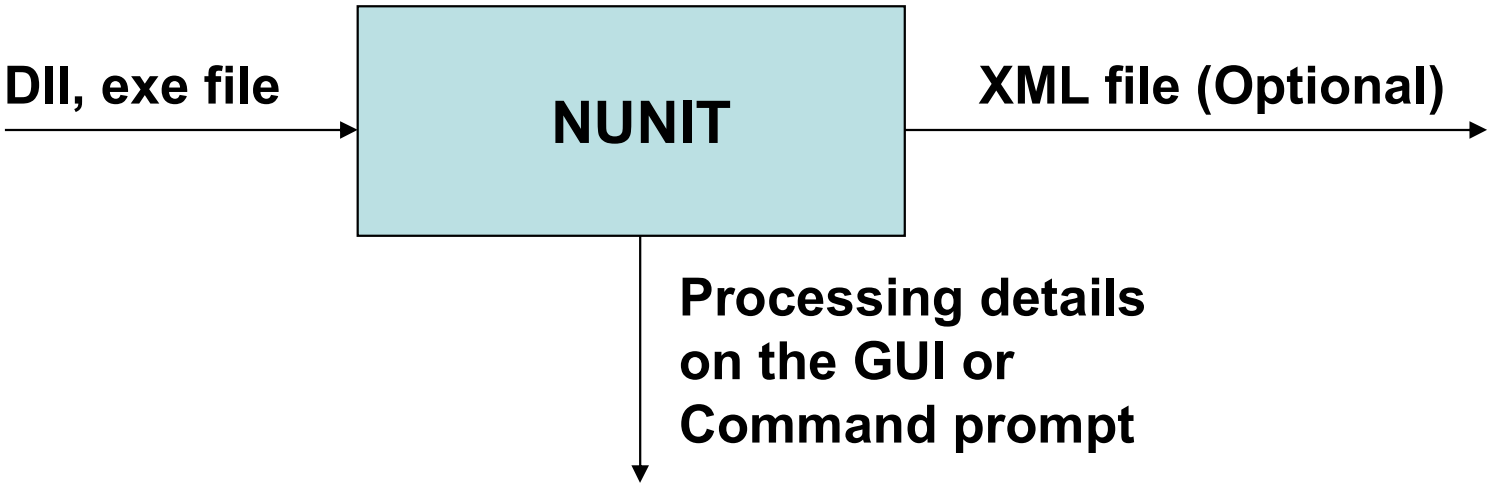
- **NUnit** is a **unit-testing framework** for all **.NET languages**
- It has been written in C#
- You can visit the link mentioned below for information on NUnit and its development

  http://www.nunit.org
- You can also download NUnit from the link mentioned here

  http://nunit.org/download.html

NUnit

- NUnit works in the following manner



- There are two ways to work with NUnit
  - GUI mode
  - Console mode

NUnit

DEMO

(Sample implementation of NUnit with Visual Studio)

Structuring Unit Tests

- **Test Code follows a standard formula:**
- Set up all conditions needed for testing (create any required objects, allocate any needed resources, etc.)
- Call the method to be tested
- Verify that the tested functionality worked as expected
- Clean up after itself

Structuring Unit Tests

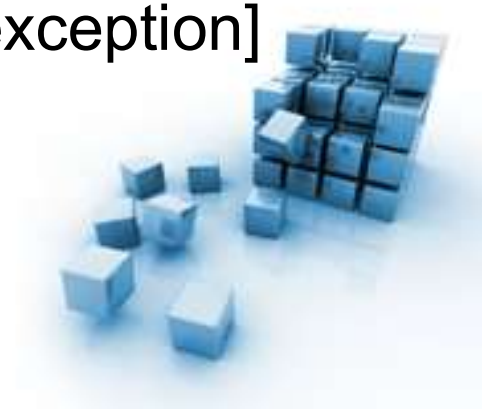
- You write test code and compile it in the normal fashion, as you would any other bit of source code in your project
- When it's time to execute the code, remember that you never actually run the production code directly
- Instead, you run the test code, which in turn exercises the production code under very carefully controlled conditions

Structuring Unit Tests

- For Example:
- If we have a method called CreateAccount()
- This method encapsulates the behavior and we can test the method with a method named CreateSavingsAccount()
- Next test method can be CreateCurrentAccount()
- Tests have to be organized on the behaviors and not necessarily individual methods

Asserts

- There are some helper methods that assist us in determining whether a method under test is performing correctly or not
- Generically, we call all these helper methods **assertions**
- They let us assert that some condition is true; that two bits of data are equal, or not, and so on
- These methods will report
  - failures [that's when the assertion is false]
  - errors [that's when we get an unexpected exception]



Asserts

- Asserts are the fundamental building block for unit tests;

- NUnit library provides a number of different forms of assert as static methods in the Assert class

- **AreEqual**

  Assert.AreEqual(expected, actual [, string message])

  - This is the most-often used form of assert
  - Expected is a value you hope to see (typically hard-coded)
  - Actual is a value actually produced by the code under test
  - Message is optional that will be reported in case of failure

Asserts

- **Assert.AreEqual(expected, actual, tolerance [, string message])**

  Assert.AreEqual(3.33, 10.0/3.0, 0.01);

  (Used for floating point numbers)

- **Less / Greater**

  Assert.Less(x, y)
  Assert.Greater(x,y)

- **IsNull / IsNotNull**

  Assert.IsNull(object [, string message])
  Assert.IsNotNull(object [, string message])

- **AreSame**

  Assert.AreSame(expected, actual [, string message])
  (expected and actual refer to the same object)

Constraint based - Asserts

- NUnit 2.4 introduced a new style of assertions that are a little less procedural and allow for a more object-oriented underlying implementation

- **Is.EqualTo**

  Assert.That(actual, Is.EqualTo(expected))

  This is equivalant to the Assert.AreEqual() classic assertion method

  The Is.EqualTo() method is a syntax helper in the NUnit.Framework.SyntaxHelpers namespace

  It's a static method that just returns an EqualConstraint object

Constraint based - Asserts

- **Is.Null**

  Assert.That(expected, Is.Null);

- **Is.Empty**

  Assert.That(expected, Is.Empty);

- **Is.InstanceOfType**

  Assert.That(actual, Is.InstanceOfType(expected));

- **List.Contains**

  Assert.That(actualCollection, List.Contains(expectedValue))
  Assert.That({5, 3, 2}, List.Contains(2))

- **Is.SubsetOf**

  Assert.That(actualCollection,
  Is.SubsetOf(expectedCollection))
  Assert.That(new byte[] {5, 3, 2},
  Is.SubsetOf(new byte[] {1, 2, 3, 4, 5}))

Custom Asserts

- The standard asserts that NUnit provides are usually sufficient for most testing
- However, you may run into a situation where it would be handy to have your own, customized asserts
- Perhaps you've got a special data type, or a common sequence of actions that is done in multiple tests

Custom Asserts

```
using System;
using NUnit.Framework;
using NUnit.Framework.SyntaxHelpers;
public class MoneyAssert
{
        // Assert that the amount of money is an even
        // number of dollars (no cents)
        public static void AssertNoCents(Money amount, String message)
        {
                Assert.That( Decimal.Truncate(amount.AsDecimal()),
                        Is.EqualTo(amount.AsDecimal()), message);
        }
}
```

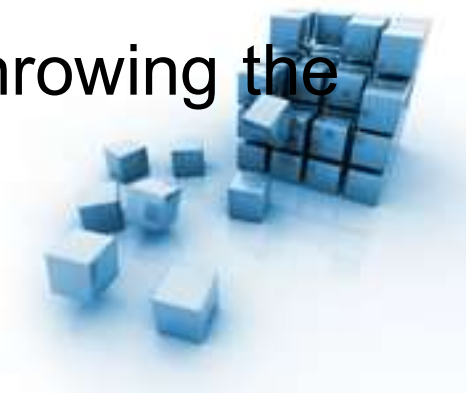
Note: There are many more Asserts. ColletionAsserts and FileAsserts which help us deal with collections and files respectively.

Exceptions and Exception Testing

- We might be interested in two different kinds of exceptions:
  - Expected exceptions resulting from a test
  - Unexpected exceptions from something that's gone horribly wrong
- Sometimes in a test, we want the method under test to throw an exception
- Consider a method which will help us divide two numbers

Exceptions and Exception Testing

- If the second number is a zero, code is going to throw an error informing us about *DivideByZeroException*
- If the method already has the code checking for the second number being 0 and if so throw DivideByZeroException, then we would like to check if the method is throwing the same properly
- If exception is thrown, we can identify that in NUnit and the test would pass for given values [num2=0]
- In short – we are checking if a method is throwing the expected exceptions



Exceptions and Exception Testing

- If the method throws any other exception apart from the one handled, then the test would give a negative result
- Once the expected exception fires, any remaining code in the test method will be skipped
- Unexpected exceptions, NUnit will take care accordingly
- It will give us the entire stack trace right down to the bug itself

Setup and Teardown

- **Per-method Setup and Teardown**
- Each test should run independently of every other test; this allows you to run any individual test at any time, in any order
- To accomplish this feat, you may need to reset some parts of the testing environment in between tests, or clean up after a test has run
- NUnit lets you specify two methods to set up and then tear down the environment per test using attributes

Setup and Teardown

- **Per-method Setup and Teardown**

```
[TestFixture]
public class DBTest
{
        private Connection dbConn;
        [SetUp]
        public void PerTestSetup()
        {
                dbConn = new Connection("oracle" , 1521, user,
        pw);
                dbConn.Connect();
        }
        [TearDown]
        public void PerTestTeardown()
        {
                dbConn.Disconnect();
                dbConn.Dispose();
        }
```

Setup and Teardown

- **Per-method Setup and Teardown**

```
[Test]
public void AccountAccess()
{
      // Uses dbConn
      xxx xxx xxxxxx xxx xxxxxxxxx;
      xx xxx xxx xxxx x xx xxxx;
}
[Test]
public void EmployeeAccess()
{
      // Uses dbConn
      xxx xxx xxxxxx xxx xxxxxxxxx;
      xxxxxx xx xxx xx xxxx;
}
}
```

Setup and Teardown

- **Per-fixture Setup and Teardown**

- Normally per-method setup is all you need, but in some circumstances you may need to set something up or clean up after the entire test class has run

- All you need to do is annotate your setup methods with the following attributes:

```
[TestFixtureSetUp]
public void PerFixtureSetup() {
...
}
[TestFixtureTearDown]
public void PerFixtureTeardown() {
...
}
```

Categories

- NUnit provides an easy way to mark and run individual tests and fixtures by using categories

- You can associate different test methods with one or more categories, and then select which categories you want to exclude (or include) when running the tests

```
[Test]
[Category(“Mathematical")]
public void TestAdd()
{
        Calculator Obj = new Calculator();

        Assert.AreEqual(Obj.Add(5,5), Obj.AddLong(5, 5));

}
```

Question time

Please try to limit the questions to the topics discussed during the session. Thank you.

What do we Test for?

- Now that you know how to test, we need to look at what to test; or more precisely, the kinds of things that might need testing

- It can be hard to look at a method or a class and try to come up with all the ways it might fail and to anticipate all the bugs

- With enough experience, you start to get a feel for those things that are “likely to break,” and can effectively concentrate on testing in those areas

What do we Test for?

- There are six specific areas to test that will help strengthen your testing skills

- *RIGHT BICEP*

- Right — Are the results right?
- B — Are all the boundary conditions CORRECT?
- I — Can you check inverse relationships?
- C — Can you cross-check results using other means?
- E — Can you force error conditions to happen?
- P — Are performance characteristics within bounds?

What do we Test for?

- **Right Result**

- The first and most obvious area to test is simply to see if the expected results are right—to validate the results.

- You can use data from file or XML or database to check

- If requirements are unclear, it can be verified with the stake holders

What do we Test for?

- **Boundary Conditions**

- Identifying boundary conditions is one of the most valuable parts of unit testing, because this is where most bugs generally live—at the edges

- Some conditions you might want to think about:
  - Totally bogus or inconsistent input values, such as a file name of "!*W:X\&Gi/w>g/h#WQ@“

  - Badly formatted data that is missing delimeters or terminators, such as an e-mail address without a top-level domain ("fred@foobar.")

What do we Test for?

- Empty or missing values (such as 0, 0.0, an empty string, an empty array, or null), or missing in a sequence (such as a missing TCP packet)
- Values far in excess of reasonable expectations, such as a person's age of 10,000 years or a password string with 10,000 characters in it
- Duplicates in lists that shouldn't have duplicates
- Ordered lists that aren't, and vice-versa. Try handing a pre-sorted list to a sort algorithm, for instance—or even a reverse-sorted list

What do we Test for?

- An easy way to think of possible boundary conditions is to remember the acronym CORRECT

Conformance - Does the value conform to an expected format?

For Example:

Email id has to be in the correct format

Password has to be minimum 6 characters and one alphabet

Ordering - Is the set of values ordered or unordered as appropriate?

For Example

If while registering an order for a new customer, customer details have to be added and then order has to be registered and finally order items have to be stored

Range — Is the value within reasonable minimum and maximum values?

For Example

Check if the value entered for age is between 1 and a maximum value of 150

To check for min and max budget to on a shopping cart portal

What do we Test for?

Reference — Does the code reference anything external that isn't under direct control of the code itself?

For Example

In a web application before we show the account summary, user has to be signed in

Existence — Does the value exist (e.g., is non-null, nonzero,
present in a set, etc.)?

For Example

*Verify if u*ser account exists after logging in

Verify if account object is not null, for a withdrawal operation

What do we Test for?

Cardinality — Are there exactly enough values? Zero – one – n rule

For Example

Quiz will have questions. Min of 2 and max of 20

Quiz will have several of question references

Time (absolute and relative) — Is everything happening in order? At the right time? In time?

For Example

Concurrency and synchronization to be taken care

Transfer of funds to outside banks only between business hours

What do we Test for?

- **Check Inverse Relationships**

- Some methods can be checked by applying their logical inverse

- For instance, you might check a method that calculates a square root by squaring the result, and testing that it is tolerably close to the original number:

```
[Test]
public void SquareRootUsingInverse()  {
      double x = MyMath.SquareRoot(4.0);
      Assert.That(4.0, Is.EqualTo(x*x).Within(0.0001));
}
```

What do we Test for?

- **Check Inverse Relationships**

- You might check that some data was successfully inserted into a database, then search for it, and then delete it.

- You might transfer money into an account, then transfer the same amount out of the account.

- For any of these operations apply an “inverse” to see if you get back to an original state

What do we Test for?

- **Cross-check Using Other Means**
- You might also be able to cross-check results of your method using different means
- Usually there is more than one way to calculate some quantity
- One would be used for production and we might pick the other for testing
- This technique is especially helpful when there's a proven, known way of accomplishing the task that happens to be too slow or too inflexible to use in production code

What do we Test for?

- **Cross-check Using Other Means**

```
[Test]
public void SquareRootUsingStd() {
    double number = 3880900.0;
    double root1 = MyMath.SquareRoot(number);
    double root2 = Math.Sqrt(number);
    Assert.That(root2, Is.EqualTo(root1).Within(0.0001));
}
```

- Separate pieces of data may be reported by objects of different classes, but they still have to agree, and so can be used to cross-check one another

What do we Test for?

- **Force Error Conditions**
- In the real world, errors happen
- Disks fill up, network lines drop, e-mail goes into a black hole, and programs crash.
- You should be able to test that your code handles all of these real world problems by forcing errors to occur

What do we Test for?

- **Force Error Conditions**

- Here are a few environmental things you could think of
  - Running out of memory
  - Running out of disk space
  - Issues with wall-clock time
  - Network availability and errors
  - Insufficient File or Path permissions
  - System load
  - Very high or very low video resolution

What do we Test for?

- **Performance Characteristics**
- One area that might prove beneficial to examine is performance characteristics
- Not performance itself, but trends as input sizes grow, as problems become more complex, and so on
- For example we might want to find out the time taken to read from an xml file and update database for about 10000 employees

Question time

Please try to limit the questions to the topics discussed during the session. Thank you.

Characteristics of good testing

- **A-TRIP**
  - **Automatic**
    - They should be automated
    - Running continuously to test the code checked-in
    - Avoid intervention of manual process
  - **Thorough**
    - Good unit tests are thorough
    - Test everything that can break your code
    - Estimate the depth of your test needed
  - **Repeatable**
    - Every test should be able to run over and over again, in any order, and produce the same results
    - Isolate the external dependencies like databases, global variables

Characteristics of good testing

- **A-TRIP**
  - **Independent**
    - Tests need to be kept neat and tidy
    - Tightly focused, and independent from the environment and each other [Other developers can also run the test]
    - Testing only one thing at a time
    - Tests should be on behavior and not the method
    - One method can have many test cases
  - **Professional**
    - Maintained to same standards as your production code
    - Test code is real code
    - Might need larger code base than production code

Characteristics of good testing

- **Testing the tests**
  - Testing code to make sure it works is a great idea
  - What happens when there are bugs in our test code ?
  - Two things you can do to help ensure that the test code is correct:
    - Improve tests when fixing bugs
    - Prove tests by introducing bugs

Characteristics of good testing

- When a bug is found “in the wild” and reported back, that means there’s a hole in the safety net—a missing test

- Four simple steps to tackle this
  - Identify the bug, or bugs, that caused the errant behavior
  - Write a test that fails, for each individual bug, to prove the bug exists
  - Fix the code such that the test now passes
  - Verify that all tests still pass (i.e., you didn’t break anything else as a result of the fix).

Question time

Please try to limit the questions to the topics discussed during the session. Thank you.

Mocks and Stubs

- The objective of unit testing is to exercise just one behavior at a time, but what happens when the method containing that behavior depends on other things
- Hard-to-control things such as the network, or a database, or even specialized hardware
- There's a testing pattern that can help: mock objects
- A mock object is simply a testing replacement for a real-world object

Mocks and Stubs

- There are a number of situations that come up where mock objects can help us
- The real object has nondeterministic behavior (it produces unpredictable results, like a stock-market quote feed.)
- The real object is difficult to set up, like requiring a certain file system, database, or network environment
- The real object has behavior that is hard to trigger (for example, a network error)
- The real object is slow or doesn't exist
- The real object has (or is) a user interface

Mocks and Stubs

- Using mock objects, we can get around all of these problems
- The three key steps to using mock objects for testing are:
  - Use an interface to describe the relevant methods on the object
  - Implement the interface for production code
  - Implement the interface in a mock object for testing

Mocks and Stubs

- What we need to do is stub out
- In many cases, stubs just implement an interface and return dummy values for the methods in said interface
- In even simpler cases, all the implemented methods in the stub just throw a NotImplementedException
- A common scenario is when there is a class that encapsulates database access, but we don't want to actually configure and populate a database to run simple tests

Question time

Please try to limit the questions to the topics discussed during the session. Thank you.

Testing and Design

- Unit testing offers several opportunities to improve the design and architecture of your code as well

- Few important areas to consider are
  - Designing for testability
  - Refactoring for testing
  - Testing the class invariant
  - TDD – Test Driven Design
  - Testing for invalid parameters

Testing and Design

- **Designing for testability**

- "Separation of Concerns" is probably the single most important concept in software design and implementation

- For example, suppose you are writing a method that will sleep until the top of the next hour

```
public void SleepUntilNextHour() {
        int howLong;
        howLong= xxxxxxxxx;
        Xxxxxxxxxx;
        Xxxxxxxxxxxxxx;
        Thread.Sleep(howLong);
        return;
}
```

Testing and Design

- **Designing for testability**
- How do we test this?
- Do we wait for an hour? Set a timer and then call the method and wait for it to return
- Instead of combining the calculation of how many milliseconds to sleep with the Sleep() method itself, split them up:

```
public void SleepUntilNextHour() {
        int howlong = MilliSecondsToNextHour(DateTime.Now);
        Thread.Sleep(howlong);
        return;
}
```

Testing and Design

- **Designing for testability**
- Now we can test the methods separately
- MilliSecondsToNextHour() can be tested if it is giving the right value
- SleepUntilNextHour() can be tested to see if the thread is put to sleep mode properly
- Assert.AreEqual(10000, MilliSecondsToNextHour(DATE_1));

Testing and Design

- **Refactoring for testing**
- Many a times we re-factor the code once it is tested or even to accommodate unit testing
- If there is a method which is internally implementing many different pieces of logic, then it becomes difficult to test
- When a test is run, it becomes difficult to know which piece of code is actually resulting in an error
- We then give scope for re-factoring of code

Testing and Design

- **Refactoring for testing**
- Consider a method to process an order RegisterOrder() and has to send the total amount back for an invoice to be generated
- If this method also has to calculate the discount based on the type of member placing the order, then it would be difficult to test for incorrect calculations for members
- So we divide the method into 2 RegisterOrder() and GetDiscountForCustomer()
- Now these methods can be independently tested

Testing and Design

- **Testing the class Invariant**
- A class invariant is an assertion, or some set of assertions, about objects of a class
- For an object to be valid, all of these assertions must be true. They cannot vary
- For instance, a class that implements a sorted list may have the invariant that its contents are in sorted order
- That means that no matter what else happens, no matter what methods are called, the list must always be in sorted order

Testing and Design

- **Testing the class Invariant**

- Within a method, of course, the invariant may be momentarily violated as the class performs whatever housekeeping is necessary

- But by the time the method returns, or the object is otherwise available for use (as in a multi-threaded environment), the invariant must hold true or else it indicates a bug

- Possible areas where class invariants might apply
  - Structural
  - Mathematical
  - Data Consistency

Testing and Design

- **Testing the class Invariant**

- **Structural**
  - The most common invariants are structural in nature
  - For instance, in an order-entry system you might have invariants such as:
    - Every line item must belong to an order
    - Every order must have one or more line items

- **Mathematical**
  - Other constraints are more mathematical in nature
  - Debits and credits on a bank account match the balance
  - Amounts measured in different units match after conversion

Testing and Design

- **Testing the class Invariant**

- Often an object may present the same data in different ways

  - A list of items in a shopping cart

  - The total amount of the sale and the total number of items in the cart are closely related

  - From a list of items with details, you can derive the other two figures

  - It must be an invariant, that these figures are consistent

Testing and Design

- **Test Driven Development - TDD**

- Test-driven development is a valuable technique where you always write the tests themselves before writing the methods that they test

- You start with what user wants and that results in writing classes that you need and not just because statements in your requirement doc says

Testing and Design

- **Testing Invalid Parameters**
- Is your class supposed to validate its parameters?
- Who's responsible for validating input data?
- Depends on the project. If UI layer is handing at the boundaries, you don't need to at class level
- If not, then we need to handle at the class level
- Mission critical applications and confidential systems may think about validating at both ends

Question time

Please try to limit the questions to the topics discussed during the session. Thank you.