Unit Test Presentation Notes

Benefits

Find problems early

- Unit tests find problems early in the development cycle.
- In test-driven development (TDD) unit tests are created before the code itself is written. When the tests pass, that code is considered complete. The same unit tests are run against that function frequently as the larger code base is developed either as the code is changed or via an automated process with the build. If the unit tests fail, it considered to be a bug either in the changed code or the tests themselves. They unit tests then allow the location of the fault or failure to be easily traced. Since the unit tests alert the development team of the problem before handing the code off to testers or clients, it is still early in the development process.

Facilitates change

- Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly (e.g., in regression testing). The procedure is to write test cases for all functions and methods so that whenever a change causes a fault, it can be quickly identified and fixed.
- Readily available unit tests make it easy for the programmer to check whether a piece of code is still working properly.
- In continuous unit testing environments, through the inherent practice of sustained maintenance, unit tests will continue to accurately reflect the intended use of the executable and code in the face of any change. Depending upon established development practices and unit test coverage, up-to-the-second accuracy can be maintained.

Simplifies integration

- Unit testing may reduce uncertainty in the units themselves and can be used in a bottom-up testing style approach. By testing the parts of a program first and then testing the sum of its parts, integration testing becomes much easier.
- An elaborate hierarchy of unit tests does not equal integration testing. Integration with peripheral units should be included in integration tests, but not in unit tests. Integration testing typically still relies heavily on humans testing manually; high-level or global-scope testing can be difficult to automate, such that manual testing often appears faster and cheaper.

Documentation

- Unit testing provides a sort of living documentation of the system. Developers looking to learn what functionality is provided by a unit and how to use it can look at the unit tests to gain a basic understanding of the unit's API.
- Unit test cases embody characteristics that are critical to the success of the unit. These characteristics can indicate appropriate/inappropriate use of a unit as well as negative behaviors that are to be trapped by the unit. A unit test case, in and of itself, documents these critical characteristics, although many software development environments do not rely solely upon code to document the product in development.

- By contrast, ordinary narrative documentation is more susceptible to drifting from the implementation of the program and will thus become outdated (e.g., design changes, feature creep, relaxed practices in keeping documents up-to-date).

Design

- When software is developed using a test-driven approach, the unit test may take the place of formal design. Each unit test can be seen as a design element specifying classes, methods, and observable behavior.
- Unlike other diagram-based design methods, using a unit-test as a design has one significant advantage. The design document (the unit-test itself) can be used to verify that the implementation adheres to the design. With the unit-test design method, the tests will never pass if the developer does not implement the solution according to the design.
- It is true that unit testing lacks some of the accessibility of a diagram, but UML diagrams are now easily generated for most modern languages by free tools (usually available as extensions to IDEs). Free tools, like those based on the xUnit framework, outsource to another system the graphical rendering of a view for human consumption.

DEMOS

Creating a unit test project

File -> New -> Project -> Test

TestContext

this.TestContext.WriteLine("Executing test '{0}.{1}()'…", this.TestContext.FullyQualifiedTestClassName, this.TestContext.TestName);

Using the Assert object

AreEqual()/AreNotEqual()
AreSame()/AreNotSame()
Fail()
IsTrue()/IsFalse()
IsInstanceOfType()/IsNotInstanceOfType()
IsNull()/IsNotNull()

```
// Arrange
int x = 1;
int y = 2;
// Act
int result = Math.Max( x, y );
// Assert
Assert.That( result, Is.EqualTo( y ) );
```

Adding a unit test for a class and function

Build simple class for Circle
Circumference = 2(Pi)(Radius) = (Pi)(Diameter)
Area = (Pi)(Radius^2)

Links

A Unit Testing Walkthrough with Visual Studio Team Test
http://msdn.microsoft.com/en-us/library/ms379625(VS.80).aspx

Working with Unit Tests
http://msdn.microsoft.com/en-us/library/ms182515(v=vs.90)

http://www.slideshare.net/th-weller/introduction-to-testing-with-mstest-visual-studio-and-team-foundation-server-2010

http://www.solidsyntaxprogrammer.com/act-arrange-assert/