Advanced Unit Testing, Part I - Overview

By Marc Clifton

An Introduction To The Issues Of Unit Testing

Beginner
C#, XP, W2K, Win2003, Win9X
Posted 19 Sep 2003
Articles by this author
44,970 views

Toolbox
Broken links?
VS.NET 2003 from $899
MSDN Univ. from $1950
Print version
Bookmark this
Send to a friend
My Profile
My Settings
My Articles
My Bookmarks
Sign out

Oh dear God, anybody who votes [for unit-testing] has major problems. I hate the @#$% things. And the "Write tests first, code later" paradigm eludes me. I mean, come on, unit testing is not the end-all, be-all people! I don't care how sophisticated your tests are, a "pass/fail" grade is not sufficient to make sure you're ready for production. - David Stone, on the "do you like design or unit testing better" survey.

Contents

Preface

This is a meandering article on the issues of unit testing and the Extreme Programming (XP) process. And what's worse, this is Part I of four articles. I've combined these two threads here because I think it's important to have the context (Extreme Programming) for the content (unit testing). While unit testing isn't something that's exclusive to XP, it is *the* critical element of XP. I think that in order to get a really good understanding of unit testing, looking at how it's used in the XP process is very valuable. XP does a good job of defining processes in which it is easier to think in terms of test driven development and helps to identify how to at least create a core set of unit tests. Without this context, it becomes a lot harder to write about unit testing as an autonomous process.

The four articles in this series are:

- Part I: Introduction to unit testing and a case study taking the XP process up to the point of writing some unit tests
- Part II: Implementation of an **NUnit** look-alike and developing the case study further with real tests and real code
- Part III: Implementation of **NUnit** extensions, revising and progressing with the case study
- Part IV: Using reflection to create unit tests that are script based instead of code based, and the impact of that on the case study

Introduction

As brash as the title sounds, I personally feel that the concept of unit testing has a long way to go. I think it's overly simplistic, over-emphasized, and often misconstrued as a replacement for "mainstream" coding techniques - requirements documentation, design documents, code reviews, walkthroughs, instrumentation, profiling, and QA. And while I also don't particularly believe that formal design processes are appropriate for a lot of development efforts (so yes, some of the techniques in XP more accurately describe how I do things), I've developed a different solution that I find works very well for me.

On the other hand, I'm realizing that unit testing has its place in the code-writing process. Even if it does very little, it has its uses, especially as a program gets bigger and bigger and changes not just can, but will, result in undetected broken code.

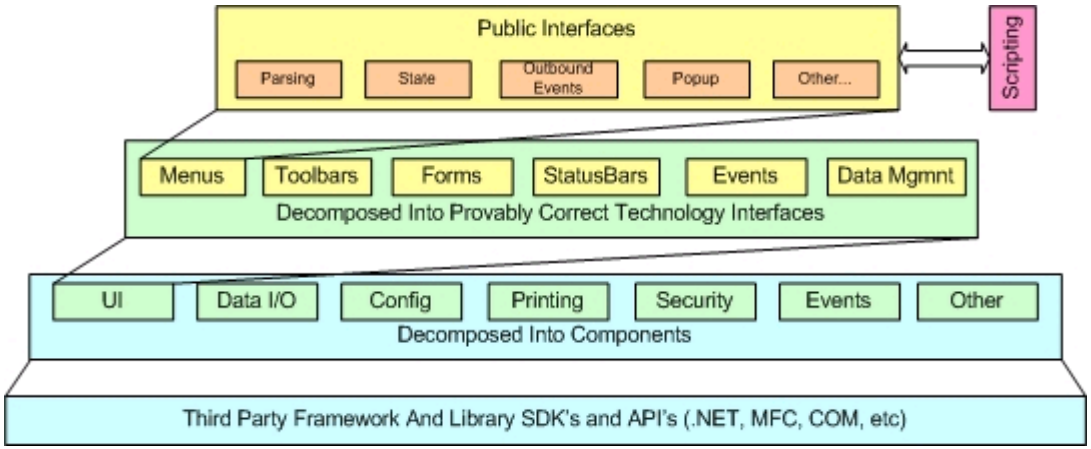
On the other hand, I also think unit testing is too simplistic. It won't find most of the higher level problems that can occur when code is changed. I'm talking about information flow problems primarily, like making sure that when an item is received down in inventory, it's automatically billed out to the customer if the purchase order for the part was associated with a work order. That's a complex information flow that requires a sophisticated unit test.

There is also the idea that if your unit tests pass, then your code is ready for production (or, at least, acceptance testing). This is an absurd assumption as David Stone so eloquently pointed out. The problem is that if you depend on your unit tests to determine whether your code is ready or not, then your unit tests had darn well be pretty good. And writing good unit tests is time consuming, boring, laborious, tedious, difficult, and requires skill. Oh wait. It also requires a good design document, so you can test the design, a good requirements document, so you can test the requirements, a good manager,

so you have the time to write all these tests, and a lot of patience, because the design and requirements are going to change, so you'll be needing to change all those nice tests you wrote too. While writing unit tests is something that we'd like to give to the grunt down the hall, sadly, it often requires a level of expertise that the grunt doesn't have.

So, here I am, of at least three minds, wondering if I just don't understand the benefits of unit testing, or if the rest of world simply loves hype. And before I make a foregone conclusion, I figured the best way to sink my teeth into the problem is to recreate **NUnit** and then add some of the features that I think are important to move it into the "real world". There's nothing like figuring out how something is done to understand why something is done.

One thing I'll say right from the get-go, though. I'm a framework guy. I believe in the concept of building on "provably correct constructs". That means I start small, I generalize, and I build a framework out of code that I have proven to do what it's supposed to do, without unit testing. There are other ways to test things, after all. So, after building the lowest foundation, proving that the objects and functions are correct, I then build the next layer, prove that layer, and so on. This is the basis for the Application Automation Layer, as illustrated by this diagram:



The result is that when I write applications, sure there's custom, application specific code that has to be written, but for the most part, most of what I'm doing is gluing together proven code, which means that most of what I'm doing is coding data flow between GUI and database, coding application specific rules (using a proven rule engine), and coding any other business layer issues. It's all data flow, in other words.

Also, I do very little refactoring. There isn't anything *to* refactor, because the proven code is well designed, simple, and works. Any refactoring that I do is usually related to improving the user interface. There are exceptions, of course. For example, my framework uses script files which code all the component glue. On a large project, there can be hundreds of script files which are pre-parsed when the application loads. Well, guess what? When the client is running a virus checker,

the application takes 15 seconds to load, as compared to 1-2 seconds. So there's a good example of some refactoring that needs to get done.

Why do I bring up refactoring? Because refactoring means changing code and object organization, often low-level code. So it would be nice if there were some automated regression tests lying around that you could use to make sure that the code still works the way its supposed to after all those changes are made. And regression testing is performed, in part, using unit tests. By the way, if you're wondering why XP relies so much on unit testing, well, it's because it relies on refactoring, and it relies on refactoring because code is "design as you go" - meaning, only design a little bit at a time. And of course, nobody is looking at the system-wide design, including the customer, so the customer makes lots of changes during the process. The idea behind XP is that it's supposed to handle projects where there's the potential for a lot of change during the development, coming from the customer. Well, that's a self-fulfilling prophecy. If you let the customer feed you requirements without really designing at least major components up front, then of course the customer is going to change the requirements, and of course you're going to need to refactor the code, and of course you're going to need unit testing to figure out what broke. And as for the "write the test first" idea, well, I'll get to that later. So much for any endorsement by Kent Beck, I suppose.

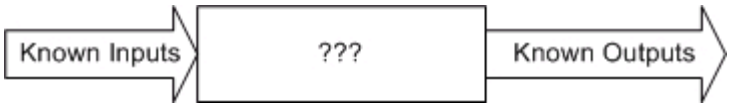
What Is A Unit Test?

A unit test verifies that a function or set of functions "honors its contract" - in other words, that the function(s) under test meet the requirements. Unit tests inspect both black boxes or white boxes.

Black Box vs. White Box Test

Black box testing is different from white box testing. The kind of testing that you can perform on the code determines, among other things, the complexity of the unit test.

Black Box Testing



A black box test (also known as a "functional test") is one in which you feed it inputs and verify the outputs without being able to inspect the internal workings. Furthermore, one doesn't usually have information regarding:

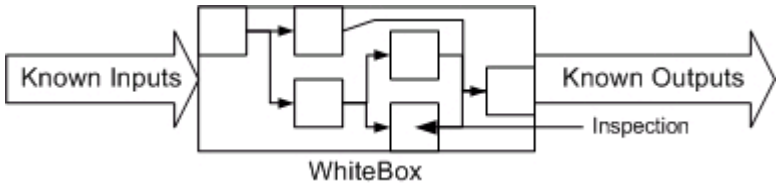
- how the box handles errors
- whether your inputs are executing all code pathways

- how to modify your inputs so that all code pathways are executed
- dependencies on other resources

Black box testing limits your ability to thoroughly test the code, primarily because the you don't know if you're testing all the code pathways. Typically, a black box test only verifies that good inputs result in good outputs (hence the term "functional test").

Classes are often implemented as black boxes, giving the "user" of the class access only to the public methods and properties that the implementer selected.

White Box Testing



A white box provides the information necessary to test all the possible pathways. This includes not only correct inputs, but incorrect inputs, so that error handlers can be verified as well. This provides several advantages:

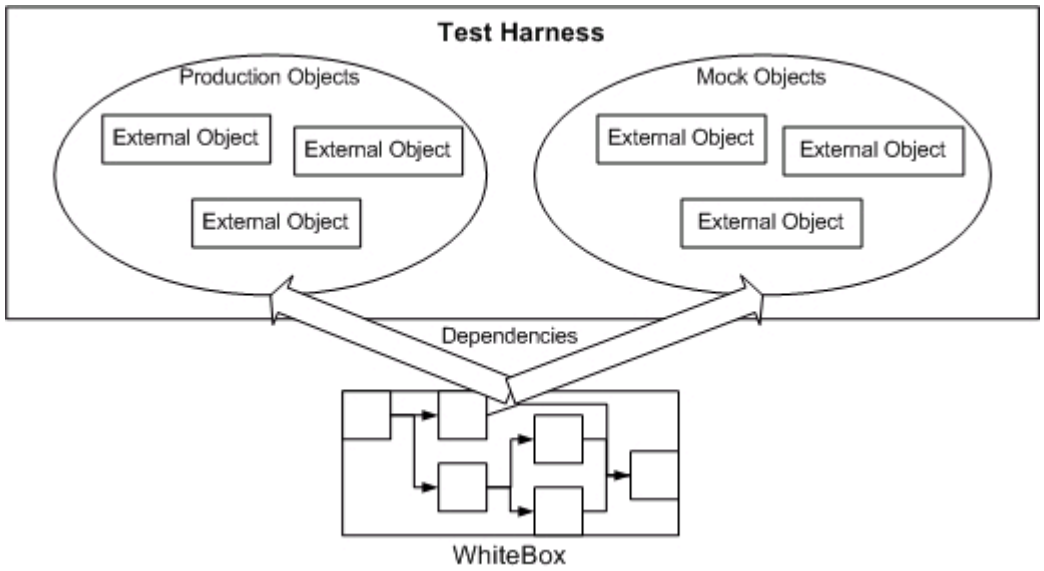
- you know how the box handles errors
- you can usually write tests that verify all code pathways
- the unit test, being more complete, is a kind of documentation guideline that the implementer can use when actually writing the code in the box
- resource dependencies are known
- internal workings can be inspected

In the "write the test first" scenario, the ability to write complete tests is vital information to the person that ultimately implements the code, therefore a good white box unit test must ensure that, at least conceptually, all the different pathways are exercised.

Another benefit of white box testing is the ability for the unit test to inspect the internal state of the box after the test has been run. This can be useful to ensure that internal information is in the correct state, regardless of whether the output was correct. Even though classes are often implemented with many private methods and accessors. with C# and reflection, unit tests can be written which provide you the ability to invoke private methods and set/inspect private properties.

Test Harness

A unit test also incorporates a "test fixture" or "test harness".

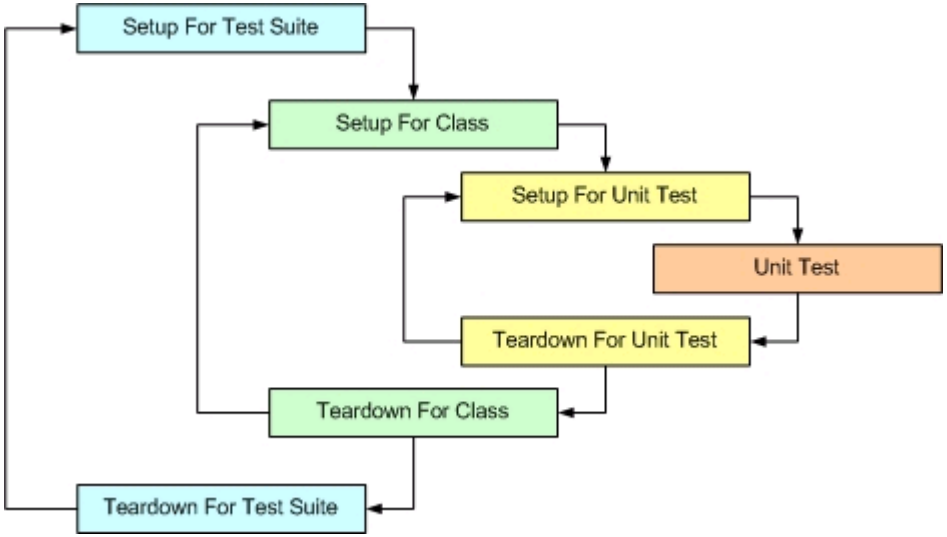


The test fixture performs any setup and teardown that the test requires. This might consist of creating a database connection, instantiating some dependant classes, initializing state, etc. The test fixture is one of the things that causes problems for unit testing. Non-trivial testing can require complex setup and teardown processes which in themselves may be buggy, time consuming, and difficult to maintain. Hence, the need for "mock objects".

The test fixture performs two levels of setup and teardown:

- necessary setup and teardown for the suite of tests
- necessary setup and teardown for individual tests

The point of having a separate setup and teardown for a suite of tests is primarily for performance reasons - repeatedly setting up and tearing down objects for each method is much less efficient than setting up and tearing down once, for a collection of objects. The following diagram illustrates this process:



Mock Objects

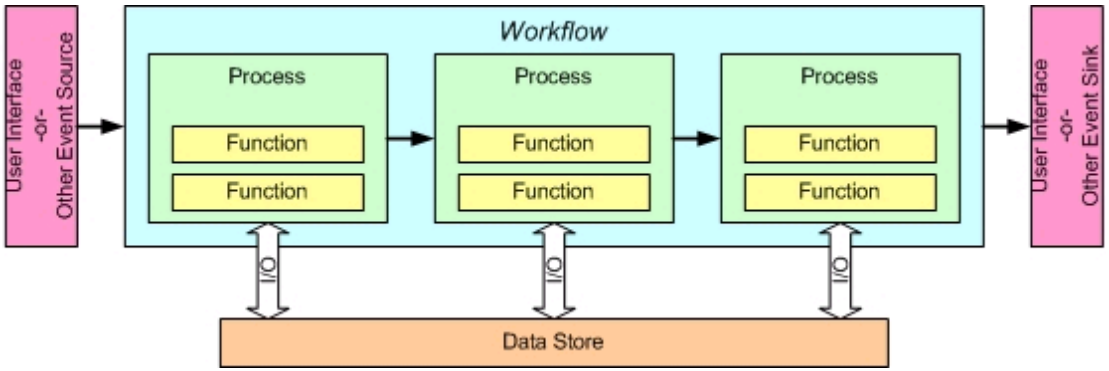
Mock objects are things that simulate complex objects with simplified functionality and make it easier to create a test harness. I'll go into mock objects in detail later, because to use them requires certain up-front design decisions to be made that are system-wide issues. But for now, simply keep in mind that unit testing often requires mock objects to simulate hardware, connections, or other resources that may not be available to the test harness. Unit tests also require mock objects for performance reasons - interfaces to production objects may be overly complex (requiring to much setup and teardown) and/or the production objects degrade the performance of the tests. Since unit tests are typically run very frequently, test performance is a factor.

What Does A Unit Test Test?

This is perhaps the most important question, and the question most difficult to answer. Simply put, the unit test verifies that the requirements are being met. Easy to say, but it's really hard to identify what the requirements are and which requirements are worthy of testing.

Customer Requirements

Customer requirements typically specify some combination of function, performance, data, and workflow. A general template for this can be illustrated as:



The customer typically thinks in terms of the user interface, clicking on a button that does something, and having the user interface change as a result. The customer also specifies the data, from the presentation level perspective.

The program implements this workflow by decomposing the workflow into a set of processes (typically determined by the customer as well, because the customer wants the processes in the workflow to remain familiar). Each process is then decomposed into a set of functions, again, often functions familiar to the customer. Automated workflows (in which the entire workflow is a black box to the customer) are less coupled to the customer's concept.

Unit testing of the customer requirements therefore consists several things, each at a different quantization. From bottom up:

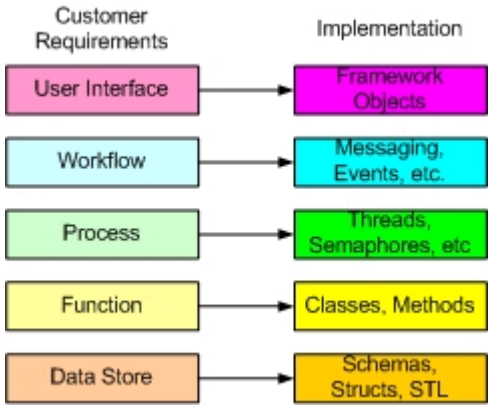
- Testing each function
- Testing each process
- Testing the workflow

It should be understood that unit testing on the function level is not sufficient. The process level integrates the functions, just as the workflow level integrates the processes. Just because the functions are working doesn't mean that the programmer put together the processes correctly, and the same for a workflow built out of the processes. Keep in mind that the term "function" relates to the customer's concept and does not necessarily map one for one to class methods.

Implementation Requirements

During the design/implementation phase, what's really going on is that the programmer is translating the customer requirements to schemas and implementations. The following is a rough idea of this concept (don't get the idea that these

are set in stone or definitive - the illustration is meant to be a rough categorization and provoke thought):

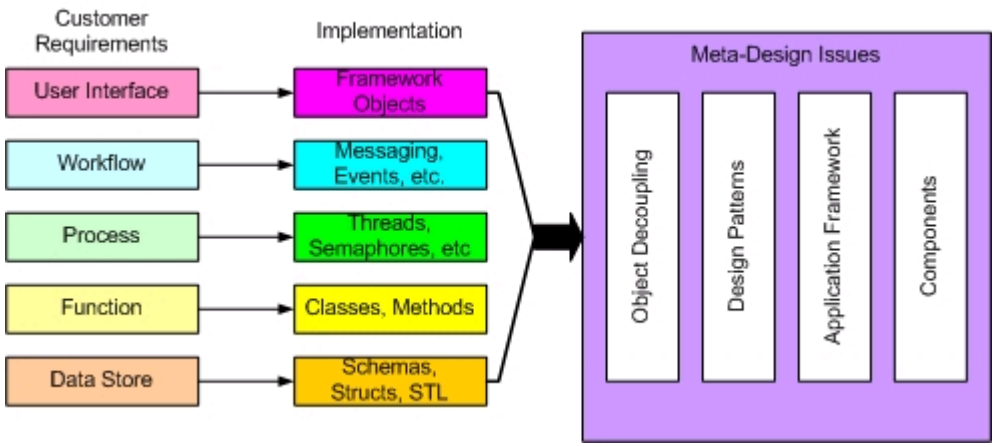


Unit tests that test implementation requirements are often different from unit tests that test customer requirements.

- Translating between customer data presentation to more optimal internal data presentation
- Translating functions into objects
- Translating data store into schemas (relational, XSD, structs, etc)

Global Requirements

This consists of meta-design issues, which the customer typically has no knowledge of. Taking the previous illustration and extending it, you see where meta-design unit testing comes in to play:



The meta-design considers the entire application from a holistic point of view and is concerned with such issues as:

- object decoupling through the use of design patterns
- an application wide framework
- componentization of different functional blocks
- instrumentation

Here again, unit testing takes a different shape. The meta-design typically relates to structure, abstraction, resource management, and other application-wide issues. Unit testing at this level is probably more concerned with validating the performance of containers, measuring resource utilization, network traffic, and other system-wide issues.

What Is NUnit?

NUnit is an application designed to facilitate unit testing. It consists of both a command line and Window's interface, allowing it to be used both interactively and in automated test batches or integrated with the build process. The following sections discuss **NUnit** as it applies to C# programming.

How Does NUnit Work?

NUnit utilizes attributes to designate the different aspects of a unit test class.

TestFixture

The `TestFixture` attribute designates that a class is a test fixture. Classes thus designated contain setup, teardown, and unit tests.

SetUp

The `SetUp` attribute is associated with a specific method inside the test fixture class. It instructs the unit test engine that this method should be called prior to invoking each unit test. A test fixture can only have one `SetUp` method.

TearDown

The `TearDown` attribute is associated with a specific method inside the test fixture class. It instructs the unit test engine that this method should be called after invoking each unit test. A test fixture can only have one `TearDown` method.

Test

The `Test` attribute indicates that a method in the test fixture is a unit test. The unit test engine invokes all the methods indicated with this attribute once per test fixture, invoking the set up method prior to the test method and the tear down method after the test method, if they have been defined.

The test method signature must be specific: `public void xxx()`, where "xxx" is a descriptive name of the test. In other words, a public method taking no parameters and returning no parameters.

Upon return from the method being tested, the unit test typically performs an assertion to ensure that the method worked correctly.

ExpectedException

The `ExpectedException` attribute is an optional attribute that can be added to a unit test method (designated using the Test attribute). As unit testing should in part verify that the method under test throws the appropriate exceptions, this attribute causes the unit test engine to catch the exception and pass the test if the correct exception is thrown.

Methods that instead return an error status need to be tested using the `Assertion` class provided with **NUnit**.

Ignore

The `Ignore` attribute is an optional attribute that can be added to a unit test method. This attribute instructs the unit test engine to ignore the associated method. A requires string indicating the reason for ignoring the test must be provided.

Suite

The `Suite` attribute is being deprecated. The original intent was to specify test subsets.

An Example

```
[TestFixture]
public class ATestFixtureClass
{
    private ClassBeingTested cbt;

    [SetUp]
    public void Initialize()
    {
        cbt=new ClassBeingTested();
    }

    [TearDown]
    public void Terminate()
    {
        cbt.Dispose();
    }

    [Test]
    public void DoATest()
    {
        cbt.LoadImage("fish.jpg");
    }

    [Test, Ignore("Test to be implemented")]
    public void IgnoreThisTest()
    {
    }

    [Test, ExpectedException(typeof(ArithmeticException))]
    public void ThrowAnException()
    {
        throw new ArithmeticException("an exception");
    }
}
```

This example illustrates the use of the six different attributes.

A Case Study

In order to put some meat onto the bones that I've been writing about, let's look at an example of a real requirement for my boatyard client - automatic customer billing. This is a fairly complex example, but I for one do not like trivial examples because they leave too many questions unanswered.

The Extreme Programming Process

I'll use the Extreme Programming methodology and look at the issue from the "test-first" perspective. This means that we're going to have to take a break from the topic of unit testing and do a quick walkthrough of the XP design process:

- User Story
- Release Planning
- Iteration Planning
- Tasks
- CRC Cards

Once we get through those steps, the discussion will return to unit testing!. If you'd like to skip this section, go ahead. However, the reason it's in here is to illustrate *how* the unit tests are determined. And for that, I'd like to take you through an example of how the unit tests are determined, which requires beginning with the "user story".

The User Story

The user story is thus:

> *Parts purchases against a specific work order should be automatically billed to the customer when the invoice comes in and should include additional vendor charges as part of the bill.*

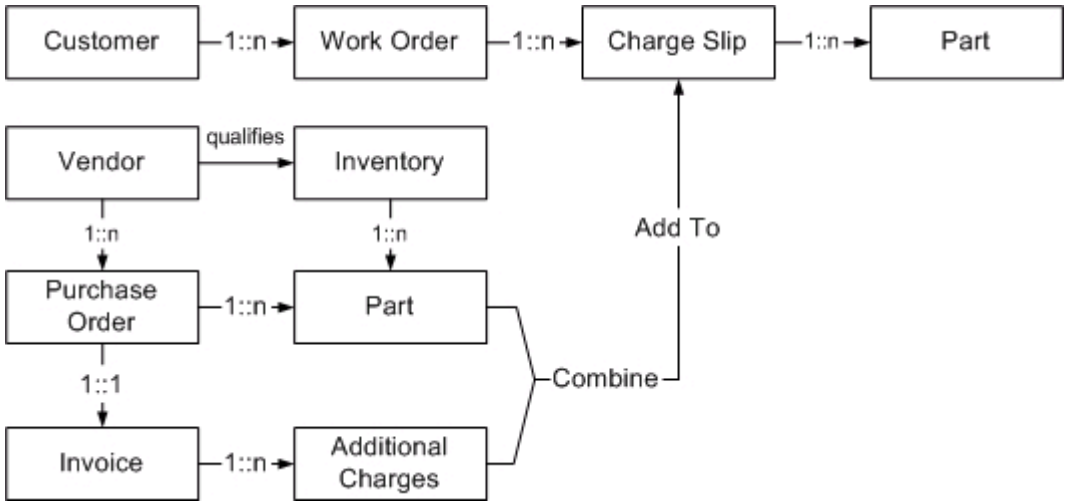
Release Planning

The detailed requirements for this story are thus:

- One or more work orders is associated with a specific customer
- The purchase order is associated with a vendor
- Each line item on the purchase order reflects a part purchased for that vendor

- A part in the inventory system may be available from several vendors
- Each vendor has its own cost for the part
- The inventory system manages its own "cost" of the part
- The inventory cost is adjusted using a moving average: (4*oldCost + newCost)/5
- Parts are designated as taxable or not
- Each PO line item is purchased either for inventory or as a part needed for a work order
- When the vendor invoice comes in, additional charges, such as shipping, hazmat, etc., are added to the purchase order
- The purchase order is closed when the purchased items are reconciled with the invoice
- When the PO is closed, any parts that were purchased against a work order are automatically billed to the customer
- Additional charges on the PO are added to the charge slip
- Since line items on a purchase order may be associated with different work orders from different customers, the additional charges have to be fairly distributed
- The only rule that anyone can come up with regarding this distribution is to divvy up the charges based on the part cost in relation to the total purchase order cost
- Parts are billed on charge slips, which emulates the manual process being used
- One or more charge slips are associated with one work order
- Parts are assigned added to the PO by selecting the part out of inventory.
- Only parts for the vendor from which the parts will be purchased should be available for addition to the PO
- Customers get different discount rates and may or may not be taxable

Which can be pictorially represented as:



Iteration Plan

Since this is a complete "package" of functionality, the customer selects the entire user story for the iteration.

Tasks

The team breaks down the user story into specific tasks:

- Create a database schema to manage this data
- Design/Implement the user interface
- Design/Implement the purchase order system
    - Create PO's
    - Add Parts
    - Reconcile additional charges from vendor invoice
    - Data access layer interface
    - All parts on a PO are purchased one vendor
- Design/Implement the work order entry system
    - Create WO's
    - Add Parts
    - Assign work orders to a customer
    - Data access layer interface

- Design/Implement the charge slip system
    - Automatically create a charge slip
    - Automatically add parts to the charge slip
    - Automatically add additional parts to the charge slip based on the cost distribution rule
    - Data access layer interface

- Design/Implement the concept of a part
    - Parts have internal part number
    - Parts have a vendor cost (and there can be more than one vendor for a part)
    - Parts have a vendor part number
    - Parts have an internal cost
    - Parts have a "taxable" field. If cleared, the part is not taxed

- There is an implied concept of a "vendor"
    - Implement the vendor as a mock object
- Implement the concept of a customer
    - The customer gets different discount rates

- Parts purchased for the customer may or may not be taxable

Non-Tasks

Since this user story excludes the idea of managing PO's, WO's, and parts (delete or modify), these processes will be left out of the current iteration. This also allows us to focus primarily on the infrastructure necessary to get build and validate the automatic billing process. Furthermore, on closer inspection, this iteration and the user story doesn't imply that persistent data storage is required, so for the time being, we'll ignore all interactions with the data access layer.

CRC Cards

Using Class, Responsibility, and Collaboration cards (CRC Cards), we can create some models for this system:

Customer	
Responsibilities:	*Collaborators:*
Track work orders assigned to the customer	WorkOrder

WorkOrder	
Responsibilities:	*Collaborators:*
Track charge slips associated with the work order	ChargeSlip

Inventory	
Responsibilities:	*Collaborators:*
Manage a collection of parts with information specific to the boatyard	Part

ChargeSlip	
Responsibilities:	*Collaborators:*
Track charges Parts Additional Charges	ChargeSlip Part Charge

Part	
Responsibilities:	*Collaborators:*
Has a vendor cost, internal cost, quantity, and tax flag and belongs in inventory	Inventory Vendor
The part is associated with a work order when being purchases	WorkOrder PurchaseOrder
Parts go on charge slips	ChargeSlip

PurchaseOrder	
Responsibilities:	*Collaborators:*
Assign parts to a purchase order	Part
Reconcile the purchase order with the vendor's invoice	VendorInvoice Vendor
When the PO is closed, the parts are automatically billed to a charge slip	Customer WorkOrder ChargeSlip Part

Charge	
Responsibilities:	*Collaborators:*
Manage additional charges on a charge slip	ChargeSlip

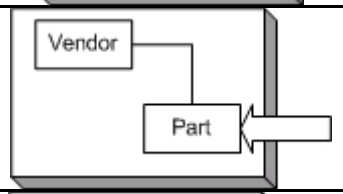
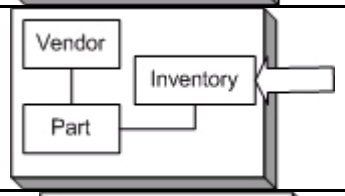
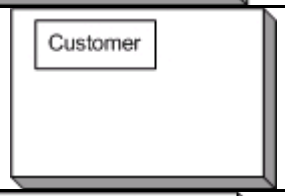
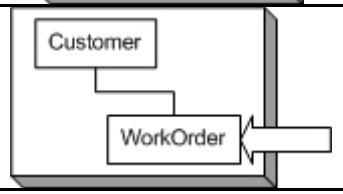
Vendor	
Responsibilities:	*Collaborators:*
Manage Vendors	
Track parts associated with a particular vendor	Part

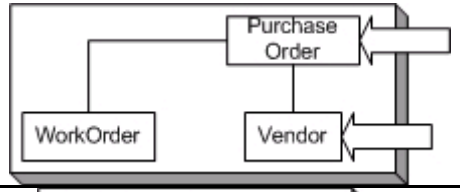
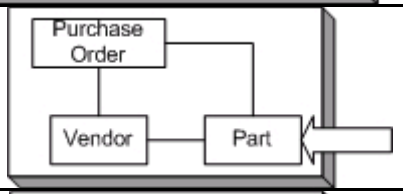
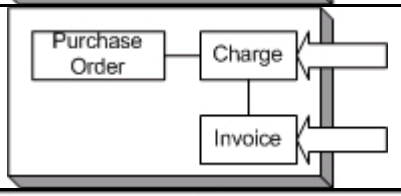
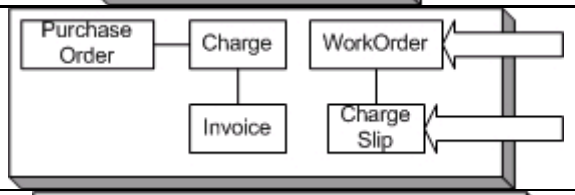
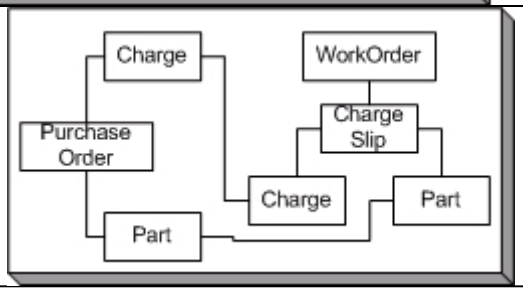
Invoice	
Responsibilities:	*Collaborators:*
Track additional charges	Charge
An invoice is associated with a purchase order	PurchaseOrder

There's probably some things I left out or could have done better. Next is to see how the CRC Cards work in simulating the scenario the customer has in mind.

Scenario Walk Through Using CRC Cards

Once the CRC Cards have been created, it's important to "place them on the table" and see how they come and go throughout the workflow. This also helps to identify data transactions between the objects, which is something that static CRC Cards does not do by itself unless we play with them in simulating our scenarios.

Create some vendors	
Add parts to the vendor list	
Associate vendor parts with inventory parts	
Create some customers	
Create some work orders associated with a customer	

Create a purchase order for a particular vendor and associated with a customer's work order	
Add parts to the purchase order, selected from the vendor's part list	
Wait for the invoice to come in, the log any additional charges on the purchase order	
Create a charge slip associated with the work order	
Add parts and charges from the purchase order to the charge slip	

Newly Discovered Non-Tasks

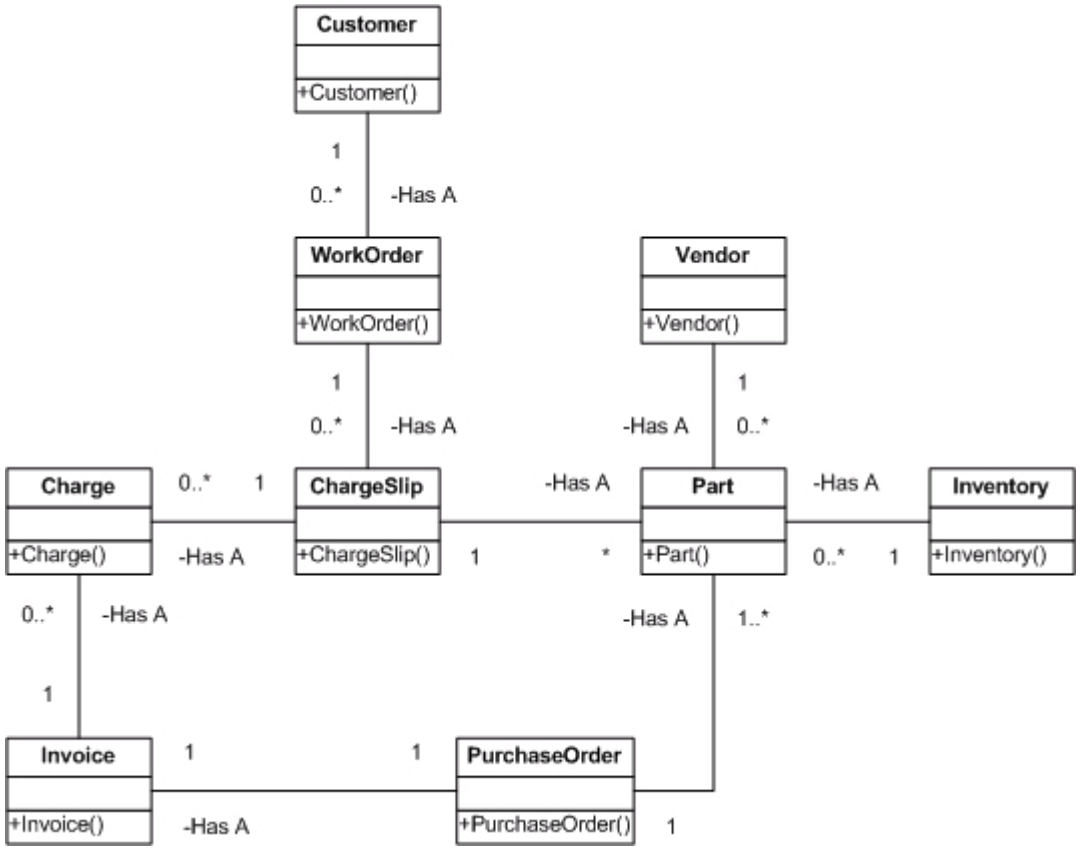
So now we have a pretty good idea of how are objects interact in all the different pieces of the project. One thing that came out of this process is that there isn't really any need for an inventory object at this point. All parts are directly associated with the vendor part list, so while the customer views things in relation to inventory (which is natural, because purchased parts go into inventory and parts used are taken out of inventory), the user story requirements can be met without the

inventory object.

Another point is that the above example demonstrates that the parts are properly added to the charge slip without regard to tax rate, discount, etc. These issues are really part of a second process, the "customer billing" cycle. It is important to realize that during the design phase of the different tasks, we've learned some additional vital information that changes the user story and the tasks derived from the user story. This information really needs to get back to the customer.

A Word About Object Entanglement And System-Wide Planning

Consider for a moment the entanglement that has been created by a corresponding object model:



This is exactly what should be avoided in a large system, because changing any one object affects most, if not all, of the other objects. OK, you say:

- That's why we have unit tests that can be applied to regression tests!
- And if we design the system to compartmentalize processes and classes into components, then the problem can be managed!
- And even better yet, if we use some good design pattern practices, we can decouple these dependencies!

Well, now there's the rub:

- Where in the XP process do we have the opportunity to consider system planning issues such as a component management framework?
- Where/when do you look at the issue of abstracting your objects a bit, for example using interfaces so that you can plug in different solutions?
- Where/when in the XP process do we look at design patterns such as class factories and messaging to manage a little abstraction?
- Where/when is instrumentation put in - debug traces to catch the problems that unit testing doesn't find, and to have an audit trail of what the customer did to create problem?

A good framework should provide instrumentation automatically, and that means using messaging or some other mechanism to communicate between objects, which also reduces object entanglement. But all of these consideration are part of system-wide planning for which there is no explicit consideration in the XP process. Heck, these issues are usually not considered in *any* development process, from my experience. But anyways, this is why I'm a framework person - I address these issues *first*, not last (or never), in software design.

A Word About Objects

Are all these objects necessary? No! Do we really need to copy data from the database into an object? No! Can't some of these transaction be handled entirely in SQL with a little abstraction and some smart interfacing between the GUI, the business layer, and the data access layer, and, ummm, maybe some *scripting*? Yes! And that' something else a good framework will do for you - it'll literally cut down the amount of hard coded objects, which reduces how much code you have to write, which eliminates a whole lot of unit testing. Now, that's simplicity, in my book (err, figuratively speaking).

Unit Tests

The first thing in unit testing that can be accomplished, now that we have the CRC Cards and object concepts in place, is to write tests for the core classes. These classes are (in order of dependencies):

- Part
- Vendor
- Charge
- ChargeSlip
- WorkOrder
- PurchaseOrder
- Invoice
- Customer

We will create unit tests for each of these classes from the information on the CRC Cards using the **NUnit** attribute syntax. I think this process illustrates rather clearly the critical importance of writing good unit tests up front in order to prevent serious problems downstream, and the complexity and difficulty of writing good unit tests.

Part

Part	
Responsibilities:	*Collaborators:*
Has a vendor cost, internal cost, quantity, and tax flag and belongs in inventory	Inventory Vendor
The part is associated with a work order when being purchases	WorkOrder PurchaseOrder
Parts go on charge slips	ChargeSlip
Parts are purchased from vendors	Vendor VendorPart

Given this CRC Card, the Part unit test needs to primarily validate that a part class can be created without errors and has setters and getters that set and return the expected values. In addition, we want to validate that a newly constructed Part object is initialized to a well-defined state.

```
[TestFixture]
public class PartTest
{
    [Test]
    public void ConstructorInitialization()
    {
        Part part=new Part();
        Assertion.Assert(part.VendorCost==0);
        Assertion.Assert(part.Taxable==false);
        Assertion.Assert(part.InternalCost==0);
        Assertion.Assert(part.Markup==0);
        Assertion.Assert(part.Number=="");
    }

    [Test]
    public void SetVendorInfo()
    {
        Part part=new Part();
        part.Number="FIG 4RAC #R11T";
        part.VendorCost=12.50;
        part.Taxable=true;
        part.InternalCost=13.00;
        part.Markup=2.0;
```

```
        Assertion.Assert(part.Number=="FIG 4RAC #R11T");
        Assertion.Assert(part.VendorCost==12.50);
        Assertion.Assert(part.Taxable==true);
        Assertion.Assert(part.InternalCost==13.00);
        Assertion.Assert(part.Markup==2.0);
    }
```

Vendor

Vendor	
Responsibilities:	*Collaborators:*
Manage Vendors Track parts associated with a particular vendor	Part

From the vendor CRC card, we see that it manages parts associated with that particular vendor. Not exactly specified in the user story or iteration (probably my fault, but hey, things like this are overlooked all the time) is that the same part number should not exist more than once for a particular vendor (although, often, the same part number is used by different vendors, so that's OK). In this particular case, notice that for each test we use a setup method to instantiate a `Vendor` class.

```
[TestFixture]
public class VendorTest
{
    private Vendor vendor;

    [Setup]
    public void VendorSetup()
    {
        vendor=new Vendor();
    }

    [Test]
    public void ConstructorInitialization()
    {
        Assertion.Assert(vendor.Name=="");
        Assertion.Assert(vendor.PartCount==0);
    }
```

```
[Test]
public void VendorName()
{
    vendor.Name="Jamestown Distributors";
    Assertion.Assert(vendor.Name=="Jamestown Distributors";
}

[Test]
public void AddUniqueParts()
{
    CreateTestParts();
    Assertion.Assert(vendor.PartCount==2);
}

[Test]
public void RetrieveParts()
{
    CreateTestParts();
    Part part;
    part=vendor.Parts[0];
    Assertion.Assert(part.PartNumber=="BOD-13-25P");
    part=vendor.Parts[1];
    Assertion.Assert(part.PartNumber=="BOD-13-33P");
}

[Test, ExpectedException(DuplicatePartException)]
public void DuplicateParts()
{
    Part part=new Part();
    part.PartNumber="Same Part Number";
    vendor.Add(part);
    vendor.Add(part);
}

[Test, ExpectedException(UnassignedPartNumberException)]
public void UnassignedPartNumber()
{
    Part part=new Part();
    vendor.Add(part);
}

void CreateTestParts()
{
    Part part1=new Part();
    part1.PartNumber="BOD-13-25P";
    vendor.Add(part1);
```

```
            Part part2=new Part();
            part2.PartNumber="VOD-13-33P";
            vendor.Add(part2);
        }
    }
```

As illustrated by the unit tests, there are certain things that the implementer must abide by:

- Duplicate parts cause an exception
- Parts not assigned a part number throw an exception
- Parts are retrieved in the same order that they are added
- Parts are retrieved by ordinal

Weakness #1: Incomplete Unit Tests

This last requirement, "parts are retrieved by ordinal", illustrates an interesting "artifact" of unit testing. This would imply to the person that implements the class that an `ArrayList` is sufficient to manage parts. But now consider a more complete test:

```
...
    [Test]
    public void RetrievePartsByName()
    {
        CreateTestParts();
        Part part;
        part=vendor.Parts["BOD-13-25P"];
        Assertion.Assert(part.PartNumber=="BOD-13-25P");
        part=vendor.Parts["BOD-13-33P"];
        Assertion.Assert(part.PartNumber=="BOD-13-33P");
    }
    ...
```

This test requires that the `Vendor` class implements a retrieval mechanism based on the part number, not just the ordinal in the part collection. This may change how the parts collection is implemented and have an impact on the "retrieve in the same order..." implementation.

Weakness #2: No Performance Measurements

Given both the ordinal and the string-based part lookup, the implementer may still choose to use an array list and implement the string-based lookup using a simple 0..n search. This is highly inefficient, but the unit test does not measure this performance.

> This is an issue that we will look at further, in part III - extending the unit test features. If the unit test included a performance measure, then the implementer would have a guideline for choosing an appropriate collection.

Weakness #3: No Resource Utilization Measurements

A typical vendor will have thousands of parts. Therefore, the Vendor class should implement a Dispose method that manually clears out the part's collection instead of waiting for the Garbage Collector (GC) to get around to freeing up unreferenced memory. Again, a unit test for this is missing.

> If memory management isn't part of unit testing, performance problems will result later on, causing a lot of unnecessary refactoring. I'll look at this issue also in Part III.

Weakness #4: Lower Order Dependencies

If the implementation changes later on, perhaps for performance reasons, so that parts are no longer maintained in the same order as they are created, then the unit test has to be modified. Unfortunately, changing the unit test does not make it obvious that code that ends up depending on this requirement also needs to be refactored. There really isn't anything that can be done about this except to recognize this dependency an write unit tests for the "higher" objects that specifically test this requirement. This kind of "float upward" test requirement is difficult to keep track of. The key is, when a higher level process depends on some specific lower order functionality, then the higher order unit test must also ensure that the lower order functions still perform as expected.

One may argue that this problem is resolved by not changing the functionality such that the unit test breaks. This is unrealistic. Both form (class architecture) and function must occasionally be changed to a degree that causes unit test refactoring. While this can be mitigated through a formal deprecation process, the point is that if the higher level processes are missing dependant lower order functional unit tests, then you might easily miss all the places where refactoring is necessary. Sure, the lower order class passes its unit test, but the higher order classes and their unit tests *may not*. This is where one of the cost-benefit tradeoffs in unit testing exists - do you take the time to write unit tests for all lower order dependencies in higher order unit tests?

Charge

Charge	
Responsibilities:	*Collaborators:*
Manage additional charges on a charge slip	ChargeSlip

Charges are very simplistic - they have a description and an amount. There's not much to the unit test for this object.

```
[TestFixture]
public class ChargeTest
{
    [Test]
    public void ConstructorInitialization()
    {
        Charge charge=new Charge();
        Assertion.Assert(charge.Description=="");
        Assertion.Assert(charge.Amount==0);
    }

    [Test]
    public void SetChargeInfo()
    {
        Charge charge=new Charge();
        charge.Description="Freight";
        charge.Amount=8.50;

        Assertion.Assert(charge.Description=="Freight");
        Assertion.Assert(charge.Amount==8.50);
    }
```

ChargeSlip

ChargeSlip	
Responsibilities:	*Collaborators:*
Track charges Parts Additional Charges	ChargeSlip Part Charge

A charge slip is a collection of parts and charges. There aren't that many charges/parts on a charge slip, so performance

and memory utilization isn't really an issue, making the unit tests fairly simple. Similar the vendor unit tests, we want to ensure that parts and charges are added correctly and that "empty" parts and charges cause an exception to be thrown. Ordering is irrelevant.

```
[TestFixture]
public class ChargeSlipTest
{
    private ChargeSlip chargeSlip;

    [Setup]
    public void Setup()
    {
        chargeSlip=new ChargeSlip();
    }

    [Test]
    public void ConstructorInitialization()
    {
        Assertion.Assert(chargeSlip.Number=="000000");
        Assertion.Assert(chargeSlip.PartCount==0);
        Assertion.Assert(chargeSlip.ChargeCount==0);
    }

    [Test]
    public void ChargeSlipNumberAssignment()
    {
        chargeSlip.Number="123456";
        Assertion.Assert(chargeSlip.Number=="123456";
    }

    [Test, ExpectedException(BadChargeSlipNumberException)]
    public void BadChargeSlipNumber()
    {
        chargeSlip.Number="12345";        // must be six digits or letters
    }

    [Test]
    public void AddPart()
    {
        Part part=new Part();
        part.PartNumber="VOD-13-33P";
        chargeSlip.Add(part);
        Assertion.Assert(chargeSlip.PartCount==1);
    }
```

```
    [Test]
    public void AddCharge()
    {
        Charge charge=new Charge();
        charge.Description="Freight";
        charge.Amount=10.50;
        chargeSlip.Add(charge);
        Assertion.Assert(chargeSlip.ChargeCount==1);
    }

    [Test]
    public void RetrievePart()
    {
        Part part=new Part();
        part.PartNumber="VOD-13-33P";
        chargeSlip.Add(part);
        Part p2=chargeSlip.Parts[0];
        Assertion.Assert(p2.PartNumber==part.PartNumber);
    }

    [Test]
    public void RetrieveCharge()
    {
        Charge charge=new Charge();
        charge.Description="Freight";
        charge.Amount=10.50;
        chargeSlip.Add(charge);
        Charge c2=chargeSlip.Charges[0];
        Assertion.Assert(c2.Description==charge.Description);
    }

    [Test, ExpectedException(UnassignedPartNumberException)]
    public void AddUnassignedPart()
    {
        Part part=new Part();
        chargeSlip.Add(part);
    }

    [Test, ExpectedException(UnassignedChargeException)]
    public void UnassignedCharge()
    {
        Charge charge=new Charge();
        chargeSlip.Add(charge);
    }
}
```

Weakness: Entanglement And Complexity

We can see something developing as we move from lower order functions to higher order ones:

- the unit tests are becoming entangled with the objects required by the object under tests
- setting up the unit tests is becoming more involved because of other required setups

These are issues that are addressed in Part IV, where I discuss a scripting approach to unit testing and demonstrate the advantages of exporting setup data to a file, for example an XML file. The primary advantage of this is that you can have a data-driven unit test iterating through a variety of data combinations which can be easily changed without recompiling the program.

WorkOrder

WorkOrder	
Responsibilities:	*Collaborators:*
Track charge slips associated with the work order	ChargeSlip

A work order has a required six digit work order number and tracks all the charge slips associated with it. It's a lot like the vendor class, in that it tracks a collection of charge slips.

```
[TestFixture]
public class WorkOrderTest
{
    private WorkOrder workOrder;

    [Setup]
    public void WorkOrderSetup()
    {
        workOrder=new WorkOrder();
    }

    [Test]
    public void ConstructorInitialization()
    {
        Assertion.Assert(workOrder.Number=="000000");
```

```
    Assertion.Assert(workOrder.ChargeSlipCount==0);
}

[Test]
public void WorkOrderNumber()
{
    workOrder.Number="112233";
    Assertion.Assert(workOrder.Number=="112233";
}

[Test, ExpectedException(BadWorkOrderNumberException)]
public void BadWorkOrderNumber()
{
    workOrder.Number="12345";
    Assertion.Assert(workOrder.Number=="12345";
}

[Test]
public void AddChargeSlip()
{
    ChargeSlip chargeSlip=new ChargeSlip();
    chargeSlip.Number="123456";
    workOrder.Add(chargeSlip);
}

[Test]
public void RetrieveChargeSlip()
{
    ChargeSlip chargeSlip=new ChargeSlip();
    chargeSlip.Number="123456";
    workOrder.Add(chargeSlip);
    ChargeSlip cs2=workOrder.ChargeSlips[0];
    Assertion.Assert(chargeSlip.Number==cs2.Number);
}

[Test, ExpectedException(DuplicateChargeSlipException)]
public void DuplicateParts()
{
    ChargeSlip chargeSlip=new ChargeSlip();
    chargeSlip.Number="123456";
    workOrder.Add(chargeSlip);
    workOrder.Add(chargeSlip);
}

[Test, ExpectedException(UnassignedChargeSlipException)]
public void UnassignedChargeSlipNumber()
```

```
        {
            ChargeSlip chargeSlip=new ChargeSlip();
            workOrder.Add(chargeSlip);
        }
    }
```

Weakness: Sort-Of Useless Test

At this point, I'm beginning to question the usefulness of my "Retrieve..." tests, both in this class and the other classes. Does it really test that I'm getting back the same charge slip that I put into the system? I don't think so. It can be easily fooled by the implementer creating a new charge slip and copying just the charge slip number! But should the test rely on a shallow comparison between two charge slips, namely their memory addresses? No! The returned item could easily be a copy. Which leads to the next problem...

Weakness: The Un-Designed

There really needs to be a deep comparison operator built into each of these classes. But I didn't think of that when I did the design because I was too focused on the user story, forgetting good overall design practices. So, as a result, I didn't think of a deep comparison operator and I didn't code any unit tests to make sure that the comparison operator worked correctly. Now, maybe this wouldn't have happened if I was working with a team of programmers. Maybe one of them would have said, gee, we need to follow good design practices here and implement deep comparison operators for these classes.

Weakness: Solving Problems That Don't Exist

This also brings up a whole slew of design issues that really are easily missed. For example, are two charge slips equal if they have the same parts and charges but the ordering of their parts and charges is different? And where else would we really require a deep comparison operator except in the unit testing? Are we solving a problem that doesn't really exist? In this, I'd say yes.

Invoice

Invoice	
Responsibilities:	*Collaborators:*
Track additional charges An invoice is associated with a purchase order	Charge PurchaseOrder

The invoice is straight forward, basically just being a placeholder for charges associated with an invoice, which are associated with a purchase order and added to the charge slip on a work order. For all that, all that needs to be tracked is the invoice number and a collection of charges.

```
[TestFixture]
public class InvoiceTest
{
    private Invoice invoice;

    [Setup]
    public void InvoiceSetup()
    {
        invoice=new Invoice();
    }

    [Test]
    public void ConstructorInitialization()
    {
        Assertion.Assert(invoice.Number=="000000");
        Assertion.Assert(invoice.ChargeCount==0);
        Assertion.Assert(invoice.Vendor=null);
    }

    [Test]
    public void InvoiceNumber()
    {
        invoice.Number="112233";
        Assertion.Assert(invoice.Number=="112233";
    }

    [Test]
    public void InvoiceVendor()
    {
        Vendor vendor=new Vendor();
        vendor.Name="Nantucket Parts";
```

```
            invoice.Vendor=vendor;
            Assertion.Assert(invoice.Vendor.Name=vendor.Name);
        }

        [Test, ExpectedException(BadInvoiceNumberException)]
        public void BadInvoiceNumber()
        {
            invoice.Number="12345";
            Assertion.Assert(invoice.Number=="12345");
        }

        [Test]
        public void AddCharge()
        {
            Charge charge=new Charge();
            charge.Number="123456";
            invoice.Add(charge);
        }

        [Test]
        public void RetrieveCharge()
        {
            Charge charge=new Charge();
            chargeSlip.Number="123456";
            invoice.Add(charge);
            Charge c2=invoice.Charges[0];
            Assertion.Assert(chargeSlip.Number==c2.Number);
        }

        [Test, ExpectedException(UnassignedChargeException)]
        public void UnassignedChargeNumber()
        {
            Charge charge=new Charge();
            invoice.Add(charge);
        }
    }
```

Customer

Customer	
Responsibilities:	*Collaborators:*
Track work orders assigned to the customer	WorkOrder

The customer manages a collection of work orders.

```
[TestFixture]
public class CustomerTest
{
    private Customer customer;

    [Setup]
    public void CustomerSetup()
    {
        customer=new Customer();
    }

    [Test]
    public void ConstructorInitialization()
    {
        Assertion.Assert(customer.Name=="");
        Assertion.Assert(customer.WorkOrderCount==0);
    }

    [Test]
    public void CustomerName()
    {
        customer.Name="Marc Clifton";
        Assertion.Assert(customer.Name=="Marc Clifton");
    }

    [Test]
    public void AddWorkOrder()
    {
        WorkOrder workOrder=new WorkOrder();
        workOrder.Number="123456";
        customer.Add(workOrder);
    }

    [Test]
    public void RetrieveWorkOrder()
```

```
        {
            WorkOrder workOrder=new WorkOrder();
            workOrder.Number="123456";
            customer.Add(workOrder);
            WorkOrder wo2=customer.WorkOrders[0];
            Assertion.Assert(customer.Name==wo2.Name);
        }

        [Test, ExpectedException(UnassignedWorkOrderException)]
        public void UnassignedWorkOrderNumber()
        {
            WorkOrder workOrder=new WorkOrder();
            customer.Add(workOrder);
        }
    }
```

Something that you might have noticed is that these tests are all similar, and believe me, it sure is getting boring writing out these tests for this article. We'll look at automating similar tests using reflection in Part IV to alleviate some of this drudgery.

PurchaseOrder

PurchaseOrder	
Responsibilities:	*Collaborators:*
Assign parts to a purchase order	Part
Parts are associated with the workorder requiring them.	WorkOrder
Reconcile the purchase order with the vendor's invoice	VendorInvoice Vendor
When the PO is closed, the parts are automatically billed to a charge slip	Customer ChargeSlip Part

The purchase order is the final piece that glues all of these concepts together. The purchase order has parts that are associated with the work order. When the invoice comes in, the part pricing may need to be adjusted and additional charges

may need to be added. After this is done, the purchase order is "closed". The parts and charges are then added to a charge slip, and the charge slip is added to the work order. A purchase order can be have parts associated with different work orders on it, so that adds a small level of complexity. The unit test for this process is large. A design issue, as to whether the automatic billing logic should be part of the purchase order or extracted from it is not thoroughly considered. For now, it'll remain in the purchase order object.

The other complexity is distributing charges (which are global to the purchase order) with some fairness amongst the charge slips created for different work order. The only "fairness" algorithm that is possible at this point is to distribute additional charges based on the relative dollar amounts of each charge slip.

```
[TestFixture]
public class PurchaseOrderTest
{
    private PurchaseOrder po;
    private Vendor vendor;

    [Setup]
    public void PurchaseOrderSetup()
    {
        po=new PurchaseOrder();
        vendor=new Vendor();
        vendor.Name="West Marine";
        po.Vendor=vendor;
    }

    [Test]
    public void ConstructorInitialization()
    {
        Assertion.Assert(po.Number=="000000");
        Assertion.Assert(po.PartCount==0);
        Assertion.Assert(po.ChargeCount==0);
        Assertion.Assert(po.Invoice==null);
        Assertion.Assert(po.Vendor==null);
    }

    [Test]
    public void PONumber()
    {
        po.Number="123456";
        Assertion.Assert(po.Number=="123456");
    }

    [Test]
```

```
public void AddPart()
{
    WorkOrder workOrder=new WorkOrder();
    workOrder.Number="123456";
    Part part=new Part();
    part.Number="112233";
    vendor.Add(part);
    po.Add(part, workOrder);
}

[Test, ExpectedException(PartNotFromVendorException)]
public void AddPartNotFromVendor()
{
    WorkOrder workOrder=new WorkOrder();
    workOrder.Number="123456";
    Part part=new Part();
    part.Number="131133";
    vendor.Add(part);
    po.Add(part, workOrder);
}

[Test]
public void AddInvoice()
{
    Invoice invoice=new Invoice();
    invoice.Number="123456";
    invoice.Vendor=vendor;
    po.Add(invoice);
}

[Test, ExpectedException(DifferentVendorException)]
public void AddInvoiceFromDifferentVendor()
{
    Invoice invoice=new Invoice();
    invoice.Number="123456";
    Vendor vendor2=new Vendor();
    invoice.Vendor=vendor2;
    po.Add(invoice);
}

[Test]
public void RetrievePart()
{
    WorkOrder workOrder=new WorkOrder();
    workOrder.Number="123456";
    Part part=new Part();
```

```
    part.Number="112233";
    po.Add(part, workOrder);
    WorkOrder wo2;
    Part p2;
    po.GetPart(0, out p2, out wo2);
    Assertion.Assert(p2.Number==part.Number);
    Assertion.Assert(wo2.Number==workOrder.Number);
}

[Test]
public void RetrieveCharge()
{
    Invoice invoice=new Invoice();
    invoice.Number="123456";
    po.Add(invoice);
    Invoice i2=po.Invoices[0];
    Assertion.Assert(i2.Number==invoice.Number);
}

[Test, ExpectedException(UnassignedWorkOrderException)]
public void UnassignedWorkOrderNumber()
{
    WorkOrder workOrder=new WorkOrder();
    Part part=new Part();
    part.Number="112233";
    po.Add(part, workOrder);
}

[Test, ExpectedException(UnassignedPartException)]
public void UnassignedPartNumber()
{
    WorkOrder workOrder=new WorkOrder();
    workOrder.Number="123456";
    Part part=new Part();
    po.Add(part, workOrder);
}

[Test, ExpectedException(UnassignedInvoiceException)]
public void UnassignedInvoiceNumber()
{
    Invoice invoice=new Invoice();
    po.Add(invoice);
}

[Test]
public void ClosePO()
```

```
{
    WorkOrder wo1=new WorkOrder();
    WorkOrder wo2=new WorkOrder();

    wo1.Number="000001";
    wo2.Number="000002";

    Part p1=new Part();
    Part p2=new Part();
    Part p3=new Part();

    p1.Number="A";
    p1.VendorCost=15;

    p2.Number="B";
    p2.VendorCost=20;

    p3.Number="C";
    p3.VendorCost=25;

    vendor.Add(p1);
    vendor.Add(p2);
    vendor.Add(p3);

    po.Add(p1, wo1);
    po.Add(p2, wo1);
    po.Add(p3, wo2);

    Charge charge=new Charge();
    charge.Description="Freight";
    charge.Amount=10.50;
    po.Add(charge);

    po.Close();

    // one charge slip should be added to both work orders
    Assertion.Assert(wo1.ChargeSlipCount==1);
    Assertion.Assert(wo2.ChargeSlipCount==1);

    ChargeSlip cs1=wo1.ChargeSlips[0];
    ChargeSlip cs2=wo2.ChargeSlips[0];

    // three charges should exist for charge slip #1: two parts and one
    // freight charge
    Assertion.Assert(cs1.ChargeCount==3);
```

```
// the freight for CS1 should be 10.50 * (15+20)/(15+20+25) = 6.125
Assertion.Assert(cs1.Charges[0].Amount=6.125;

// two charges should exist for charge slip #2: one part and one
// freight charge
Assertion.Assert(cs2.ChargeCount==2);

// the freight for CS2 should be 10.50 * 25/(15+20+25) = 4.375
// (also = 10.50-6.125)
Assertion.Assert(cs2.Charges[0].Amount=4.375;

// while we have a unit test that verifies that parts are added to
// charge slips correctly, we don't have a unit test to verify that
// the purchase order Close process does this correctly.

Part cs1p1=cs1.Parts[0];
Part cs1p2=cs1.Parts[1];
if (cs1p1.Number=="A")
{
    Assertion.Assert(cs1p1.VendorCost==15);
}
else if (cs1p1.Number=="B")
{
    Assertion.Assert(cs1p1.VendorCost==20);
}
else
{
    throw(IncorrectChargeSlipException);
}

Assertion.Assert(cs1p1.Number != cs1p1.Number);

if (cs1p2.Number=="A")
{
    Assertion.Assert(cs1p2.VendorCost==15);
}
else if (cs1p2.Number=="B")
{
    Assertion.Assert(cs1p2.VendorCost==20);
}
else
{
    throw(IncorrectChargeSlipException);
}

Assertion.Assert(cs2.Parts[0].Number="C");
```

```
            Assertion.Assert(cs2.Parts[0].VendorCost==25);
        }
    }
```

As can be seen in the purchase order unit tests, it would be effective to implement these tests as a progression - if one passes, then proceed with the next. The final unit test, which puts all the parts and charges together and makes sure that they get added to the charge slip can leverage this progression, reducing the amount of time it takes to write the tests. It also encourages more robust testing of the basic functions, if for no other reason than the programmer knows that the effort put into the simpler tests can be leveraged in the more complex tests. I'll look at this more in Part III when I extend the basic unit test capability.

Coming Next...

This concludes the rather long subject introducing unit testing. In the next article, I implement a unit testing environment similar to the **NUnit** windows-based application and implement the case study functionality to illustrate the basic unit testing. Note that the unit tests above haven't actually been compiled or tested, and could very well have errors in them. I'm writing this as if I were actually going through the "test-first" process, so you get to see all of my mistakes.

About Marc Clifton

Marc lives in Rhode Island, eeking out a living as a consultant for clients in San Diego and Connecticut (the one in Rhode Island having flopped miserably). Having no formal education in software development, he feels exceptionally qualified to inflict his opinions on the community at large regarding how programming should be done, which means here at Code Project.

Currently he is writing a book on "Software Methods for C# Programmers" to be published by Addison-Wesley, which will explore the issue of supporting methods like AOP, XP, SCRUM, and others from the code perspective. Other activities include enjoying summer vacation by goofing off with his 12 year old son, taking evening walks on Narragansett Beach with his girlfriend, and writing and editing articles for The Code Project, and running the Scripting Framework project. Oh yes, a couple hours of work here and there to help pay the bills too.

Click here to view Marc Clifton's online profile.

Other popular C# Programming articles:

- I/O Ports Uncensored Part 2 - Controlling LCDs (Liquid Crystal Displays) and VFDs (Vacuum Fluorescent Displays) with Parallel Port
  Controlling LCDs (Liquid Crystal Displays) and VFDs (Vacuum Fluorescent Displays) with Parallel Port
- Minesweeper, Behind the scenes
  This article demonstrates directly reading another processes memory in C# using P/Invoke and Win32 Api's.
- Reversi in C#
  The game of Reversi in C#.
- Using MSMQ for Custom Remoting Channel.
  This article describes how to design, implement (C#) and configure the Custom Remoting Channel using MSMQ.

[Top]

Rate this Article for us! *Poor* ○ ○ ○ ○ ○ *Excellent* Vote

FAQ

Noise level Medium Search comments Set Options

View Dynamic Per page 25

New thread Msgs 1 to 25 of 36 (Total: 36) (Refresh) First Prev Next Last

Subject	Author	Date
Don't test the compiler	**Sam Thomas**	**14:53 6 Feb '04**
Other Unit Testing Tools	**Manfred Lange**	**1:59 6 Feb '04**
Unit Test Framework (UTF)	**Bnaya Eshet**	**12:55 27 Jan '04**
Assertion in NUnit	**Reuven Yagel**	**10:57 9 Oct '03**
Re: Assertion in NUnit	Marc Clifton	11:44 9 Oct '03
"provably correct components"	**pholser**	**11:39 3 Oct '03**
Re: "provably correct components"	Marc Clifton	7:27 9 Oct '03
Excellent article, but...	**Gigi Sayfan**	**0:56 1 Oct '03**
Re: Excellent article, but...	Jerry Li	17:58 1 Oct '03
Re: Excellent article, but...	Anonymous	18:25 1 Oct '03

About unit testing	**Samiva**	**12:50 28 Sep '03**
Re: About unit testing	bradw2k	17:42 25 Jan '04
how to print one ASP table using ASP among 100 tables	**jeganjirediff**	**2:33 26 Sep '03**
Re: how to print one ASP table using ASP among 100 tables	Marc Clifton	7:09 26 Sep '03
Assuming Test Order is invalid	**Anonymous**	**15:09 24 Sep '03**
Re: Assuming Test Order is invalid	Marc Clifton	15:58 24 Sep '03
Framework development	**daragh**	**6:07 24 Sep '03**
SO GOOD!!	**WilsonWei**	**5:43 23 Sep '03**
Good one	**tstih**	**5:15 23 Sep '03**
Re: Good one	Marc Clifton	20:46 24 Sep '03
Re: Good one	tstih	7:43 30 Sep '03
Weakness #3: No Resource Utilization Measurements	**Tor Hovland**	**10:42 20 Sep '03**
Re: Weakness #3: No Resource Utilization Measurements	Daniel Turini	11:01 20 Sep '03
Re: Weakness #3: No Resource Utilization Measurements	Marc Clifton	11:16 20 Sep '03
Re: Weakness #3: No Resource Utilization Measurements	Daniel Turini	17:36 20 Sep '03

Last Visit: 12:33 Tuesday 9th March, 2004

First Prev Next Last

All Topics, C#, .NET >> C# Programming >> General
Updated: 19 Sep 2003
Editor: Marc Clifton