Software Craftsmanship

"The proof is in the pudding!"

# Software as an art and a science

Now that we've covered the XAML markup language and the different "dialects", or anomalies that exist between WPF, Windows 8/WinRT, and Windows phone 8 XAML, we can now concentrate on the design patterns and industry best practices that we'll use throughout the rest of this book to design and develop a "Smart Client", lob (line of business) application. The primary idea behind writing this book is to take you through the experience of working with a "virtual" (fictional) team to assist in learning the concepts by presenting some of the common issues or tips that will resonate with the readers from all walks of IT.

The patterns and paradigms discussed in this book will present the information to learn to design and develop robust, reusable, business domain related software solutions that solve business problems found in the enterprise using effective design methodologies, design patterns, and other tools that are used throughout the software development community.

The book will offer tips and tricks on how to achieve the highest level of user interface layer component reuse between the three XAML based target platforms. The applications will be designed with flexibility in mind. We will concentrate on the S.O.L.I.D object oriented design principles to aid us when designing the classes that make up our applications. Our ultimate goal is to illustrate how to design applications that are divided into logical layers that promote a clear responsibility of concerns. Physically the layers will represents class libraries that are concise, flexible, extensible, and loosely coupled from other components throughout the system. This will maximize code reuse throughout your organization, as well as make your code easier to write unit test (which I'm sure all of you are writing unit tests… ☺).

n **Note** The S.O.L.I.D object design principles represent an acronym in which each letter represents a principle to consider when designing a class, property, or method in object oriented design.

# S.O.L.I.D Object Oriented Design

The (S) stands for the Single Responsibility Principle: This principle means that a class should have one and only one responsibility. As an example, your Order class shouldn't be concerned with how yearly reports are printed. This would be the responsibility of a class called ReportPrintService. The Order class should only be responsible for describing an Order in the system.

The (O) principle stands for The Open Close principle: This means that you should design your code to be open to extensibility and closed to modification. You may ask yourself, "How can I make a class open to be extended without modifying the code?" There are several ways to assist with this principle. For instance, you could create a Base class and inherit from the base class to implement methods on your own. My favorite approach is a design pattern called the Decorator pattern. The official definition of the Decorator pattern is:

"The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclass for extending functionality."

I will provide a full example of this pattern, as well as other useful design patterns later in this chapter. (L) Liskov substitution principle: Means you should be able to substitute for any type in which the class subclasses or any interface in which the class implements. (I) Interface segregation principle: states that when possible, you should code to an interface rather than a concrete implementation. (D) Dependency Inversion principle promotes two patterns, Dependency Injection (DI), and Inversion of Control.

# To unit test or not to unit test?

This topic can provide entertainment when there is a debate in which each participant feels strongly about their preference. "Unit tests sound great in theory, but I don't have time to meet deadlines AND write unit tests that I don't gain anything from." "Unit tests are the best thing since sliced bread and the butter that you put on it!" "I've written code for over 20 years just fine without unit tests." "With unit tests, I know the moment that a bug is introduced into the system, as well as where to find it." Those are just a few of the arguments that come to mind. Here's my two cents:

I believe there is no absolute "correct" answer. Honestly, does unit testing add additional development time? Generally yes. Does unit testing help reduce the bugs in an application? There have been many studies that suggest that unit testing does in fact help to make your code less error prone. Just like anything else, unit testing takes practice to gain proficiency, which means that your unit tests will only provide quality protection when they are quality tests. You have to know what to test and what unit tests are redundant or misleading. Practice makes perfect!

## Test Driven Development (TDD)

While we are on the topic of Unit Tests, I'm obligated to talk a bit about Test Driven Development (TDD). TDD is a software development paradigm in which you write unit tests BEFORE you even implement the class under test. If you write each test before the method you are testing, you guarantee that all methods are covered with a unit test. There is another benefit to using TDD that many people overlook. If you write a unit test for method PrintService.PrintReport() before a PrintService.PrintReport() exists, then you are designing the method and the public contract of the PrintService class. This design is based on how you would naturally imagine the class should be used. So not only do you guarantee that each method has at least on unit test associated with it, you also get to design your classes based on how you would use the class. When the unit test is finished, it can server as a succinct example of how the class should be used.

There is more to TDD than writing unit tests first, so here are the rules to TDD. The first rule that you should consider is "Make sure the unit test fails." So obviously, if you write the test before the class under test even exists, your solution will not compile. This isn't the same as the first failure we are after, so you'll want to create an empty class declaration for the classes used in the test so that it will build. Once the test will build, then run the test to experience the first failure. Once your test fails, the next step is to make the smallest change possible to the class under test then run the test again. Do this until the test passes. The key to this process is the importance of making small incremental changes until the test passes. If you were to get your unit test to build and fail for the first time, then you move on to make the test pass, I would then write the entire method very quickly, trying to anticipate issues and other ideas for the class which I would then implement. Instead of making small incremental changes to achieve a passing unit test, I would make one big change and even add some fields or methods that I knew would be coming up. So what's the big deal? Well when you make the smallest changes possible in an incremental fashion, you will only add what is absolutely necessary which will greatly reduce the size and complexity of your code.

This moves on to the next guideline, that is once you've achieved a Build -> Test -> Fail -> Small change to method, test, repeat until the test passes, now it's time to refactor the method. Since you used small incremental changes, you should refactor the method that you've created to make sure that it is clean and concise.

So the cycle goes Red->Green->Refactor->Retest->:

Red (unit test failed due to no implementation)

Green (unit test passes after small incremental changes. The small incremental changes guarantees that we don't jump ahead of ourselves and that we only add the code that is needed to make the test pass which can reduce complexity)

Refactor (Go back and refactor your new code to clean up the implementation) and Test again to make sure your refactoring didn't break anything.)

Move on to your next test.

Imagine that you wanted to write an application that would accept a user's name during login and it would return a welcome message to the user. A TDD style would go something like this. First I would create a new Blank solution in Visual Studio. I would create a class library for the functionality of the program, and I would create a new Unit Test Project to the solution and add a reference to the class library so that the test can actually use the library to test it.

Next, I would rename the default unit test class that is created with the unit test project. Then I would create my first test, which would looking something like this.

GreetingTests.cs

using System;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using ApressXaml.Usb;

namespace ApressXaml.Usb.UnitTests

{

[TestClass]

public class GreetingTests

{

private IUserLoginService \_UserLoginService;

[TestInitialize]

public void Test\_Init()

{

\_UserLoginService = new UserLoginService();

}

[TestMethod]

public void If\_A\_Name\_Length\_GreaterThan\_0\_Then\_Service\_Returns\_Message()

{

if (\_UserLoginService == null)

throw new NullReferenceException();

var validName = "Buddy";

var actualGreeting = \_UserLoginService.GreetUser(validName);

var expectedGreeting = "Hello Buddy!";

Assert.AreEqual(expectedGreeting, actualGreeting, "The user greeting is incorrect");

}

}

}

As you can see, we have a simple test class. I've created an IUserLgoinServer interface and a UserLoginService class in the class library so that the solution will compile, however, there is no implementation, so the test will fail first (which is what we want to happen).

Now that the test has successfully failed, it's time to make small incremental changes until the test passes.

Here is the empty IUserLoginService.cs interface definition.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ApressXaml.Usb

{

public interface IUserLoginService

{

string GreetUser(string userName);

}

}

Here is the empty UserLoginService.cs class definition.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ApressXaml.Usb

{

public class UserLoginService : IUserLoginService

{

public object GreetUser(string validName)

{

throw new NotImplementedException();

}

}

}

As you can see, the only thing that exists in the class is the GreetUser method declaration. This is required for the solution to build so that the test can fail. Now, I need to make the smallest change possible to make the test pass. This should be pretty easy. We'll replace the line in which the NonImplementedException is thrown with the functionality that is expected in the unit test. Here is the class with the small change added:

UserLoginService.cs with small incremental change to make the test pass

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ApressXaml.Usb

{

public class UserLoginService : IUserLoginService

{

public object GreetUser(string validName)

{

return string.Format("Hello {0}!", validName);

}

}

}

As you can see, I've changed one line of code and when I re-run the test, it passes, however, let's look at the code for a moment. There are no null reference checks on the validName parameter. The return type of GreetUser is of type object because it was generated by the refactor tools of visual studio. This validName parameter should have a name that fits better with its purpose, and you should always check for null values.

This is the part of TDD that was so hard for me to get used to. Starting out, I would have identified these issues, and resolved them on the first edit of the class. Though through much practice, I have learned to wait on the refactor stage of the process so I can reap the benefits of only changing the minimum code on each iteration until the test passes.

Now that it's time to refactor the method, I would make the following modifications, and retest until all tests pass, then move on to the next test and in return, the next feature.

The final version of the UserLoginService.cs after Refactoring and retesting

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ApressXaml.Usb

{

public class UserLoginService : IUserLoginService

{

public string GreetUser(string userName)

{

if (string.IsNullOrEmpty(userName))

throw new ArgumentNullException("userName");

if (userName.Length == 0)

throw new ArgumentException("userName cannot be empty.");

return string.Format("Hello {0}!", userName);

}

}

}

As you can see, I've added some defensive programming (null reference checks), renamed the parameter, and changed the method's return type. Once I finished these refactorings, I re-ran the test and it passed. That's how I prefer to use TDD. We will write many more unit tests before the solution is complete. Later, I'll explain some more advanced concepts like Stubbing, Mocking, and more in Chapter X.

Design methodologies and design patterns are major components of any software project, but, before we get to the design patterns that we will use in this book, let's meet our virtual team!

# Meet the team

Now the story is, you've recently accepted an offer from Acme systems to assist in the development of several line of business applications. The company has recently changed owners and they've decided to bring all enterprise systems in-house. The team is full of individuals with their own roles, opinions, and levels of experience with their job.

## The development manager

The development manager is the person that you report to. His job is to manage all of the resources allocated for this new company initiative. He must stay on top of people, project budgets, and timelines.

## The business analyst

The business analyst knows the business. Even more important, the business analyst knows what to ask the business users to get useful information regarding the software project. The business analyst will take requirements gathering questions from the developers and provide explanations on why they are problematic and offer corrections. The B.A’s purpose is to emphasize the importance of gaining knowledge of the business domain to ensure the user’s needs are met.

## The junior developer

The junior developer is an entry level .NET developer with the least amount of experience on the team. He has he is fresh out of school and this is his first job in software development. Everyone says that he is a quick learner and that he loves software development. He is an enthusiastic fellow, sometimes which can be misinterpreted as overzealous, but he means well. The junior developer stays on top of all of the newest technologies and he's quick to suggest them for projects. He will look to you for guidance and in observing his mistakes, we can learn something new too. I think that most of us can relate to the junior developer, after all, we were all there once in our careers.

## The guru

The guru works magic with the keyboard. The guru’s experience includes all of the design patterns and technologies used throughout the book. He is the goto guy (yes, pun intended) for anything related to software (and then some). The guru can be stubborn when it comes to his opinions on software development, though he has been known to adopt a new technology once he has personally considered all of the factors that he feels are important. The guru is a bit of an introvert, however, when asked, he's happy to provide guidance to the team.

## The DBA

The DBA holds the key to the realm of data access and storage. The DBA holds the keys to the database (yes, pun intended again). The DBA is in control of data access related decisions from DB server types used to the schemas, roles, users, and permissions. The DBA is in charge of Schema change requests during development. He is also responsible for database back up, as well as restoration (disaster recovery). We must always remember that the DBA is in charge of new systems as well as old legacy systems.

# The first team design meeting

The first design meeting consists of you, the junior developer, the Guru developer, the Business Analyst, the DBA, and the product champion.

During this meeting, the team's agenda is as follows:

* Allow the team to meet and get acquainted
* Discuss the business problem that will be solved by the team.
* Identify each person's role on the team.
* Discuss any project deadlines that exist
* Decide on a team development paradigm
* Create meeting schedules
* Decide on the methods of communication that the team will use.
* Decide on technologies that will be used (based on the information that's been given).

In the first meeting we will learn about the business problem that needs to be solved by the team. The business analysts has the most knowledge of the business processes on the team not counting the Product Champion. The Product champion is an employee of the client and they are considered a "power user", so they've been chosen to allocate time every week to assist with the development of the solution.

As the meeting starts, the business analysts explains the details of the business problem that is to be solved. "As many of you know, Acme has decided to bring all system development in house. This decision came about when it was discovered that a mandatory accounting system update broke the integration with the in house purchasing requisition system used throughout Acme Inc.

The plan is to rewrite the entire accounting system over the course of one year. The decision to start with the purchasing requisition system was made due to the fact that the old solution no longer worked with the legacy system.

If there is anything that we've learned over the last ten years in software development, it's this: If you provide a software developer with twenty reams of printed UML diagrams and other legacy style design artifacts and request that they create a software solution, nine times out of ten you will end up with an application that makes perfect sense to the developers and yet still does not click with business user's ideas of what the system should be.

This is because typical software developers are experts in software development, not the business domain in which they are solving a business problem. Developers don't think like users.

The point is, you need feedback from a user of the system and you need the feedback often during the development of the software solution. With this constant feedback during development, you can rest assured that you are developing for the user.

All team members have experience with the SCRUM agile team paradigm and so they decide to use SCRUM and agile methodologies. As such, it's decided that the team will work in 2 week sprints, and that they will use Team Foundation Services 2013 to record the product backlog, plan and schedule sprints, create user stories and tasks as well as for source control for the application.

TFS is a great choice because of the awesome integration with Visual Studio. It's a one stop shop for issue tracking, bug tracking, recording user stories, and providing estimates. You can literally create a user story through its lifecycle from creation, through to creating specific child tasks that contain the information you need to develop the story, all the way through to acceptance testing, and finally closed and saved.

The integration with SharePoint adds a nice web friendly user interface to TFS. You can create excel spreadsheets to display graphs regarding sprints completed and the remainder of your product backlog. The SharePoint TFS portal has a wiki that the team can use to capture important information regarding the project. You can plain sprints from the web and much more.

One of the features of TFS is an absolute god-send. I'm speaking of the integration with Microsoft Office. You can create queries to retrieve a subset of TFS work items, and export the list to an excel spreadsheet. The spreadsheet will allow you to work with batch work item creation with very little effort when compared to manually entering fifty plus user stories. Plus some people will demand a spreadsheet over a URL to all of the information they could possibly need. Ahh, the communal spreadsheet. One worksheet, one network share, 4 different users. Those of you that have been there I'm sure you are smiling and gritting your teeth. Those of you that haven't been there, be thankful.

The team decides that the software will be created with as much unit test coverage as possible, and they will use the mstest functionality that is built into Visual Studio to maintain the unit tests. The team has requested another meeting in two weeks to work with the business user to create user stories and identify other requirements.

# How to gather software requirements from the user

## How to talk the talk when it comes to gathering requirements

As a developer, I'm sure that you've realized that your users of your line of business applications don't care that you implemented a linked list from scratch while writing the program. They care about how the system is used, the results that it produces, and the speed in which the results are produced.

So it's very important for us a developers to work closely with the business user to learn as much about the business side of their day to day work load and how to create software to improve on these things.

One major component to that is vocabulary. During brainstorming sessions, be sure to identify nouns and verbs as the Business analyst or the business user explains the system. If you ask a question and the answer starts with something like "Do you mean a \_\_\_\_\_\_?" these are the terms that you need to identify and record so they can later become part of your ubiquitous language (Chapter 3).

The more that you know about the business side of the solution that you are creating, the better off your chances will be of project success. Try to study up on the domain in non-programming terms. Perhaps offer to buy the business analyst and/or the business user lunch so you can discuss any questions that you may have. Try to speak their language.

## User stories and how to create them

In agile development, a user story is basically a simplified use case. It represents an interaction between the user and the system. Sometimes, a user story will represent an interaction between two systems!

### The anatomy of a good user story

There are many known traits of a good user story. First and foremost, the user story must serve value to the business. This means that the user story should be written using the ubiquitous language of the domain so that anyone involved can understand the request.

Another clue to a great user story is a short user story. The idea isn't to hash out every aspect of the design and development of a user story's feature. We'd rather capture just enough information to be able to refer to the user story later in development and if the feature makes the cut, then we'll use the user story to start planning the design of the feature. Had we created the user story with design choices to begin with, and the user story didn't make the cut, we would have lost all of the extra time to a feature that will not be implemented. Keep it short, sweet, and in the terms of the business domain.

I'll use index cards when I'm creating user stories. I've even asked the business user if they would mind trying to write the stories on the cards in their own words. The benefits were two fold. One, I knew with the user explaining the user story, we would definitely capture what the user wants, and in the business domain's terms.

Two, no one needed to witness my subpar handwriting. It was a win! I usually write a story title, the date, and then a brief description. Remember, the idea is not to capture all of the important details, the point is to write enough to remember how to ask for the details when they are needed.

# Conclusion

Indeed, I do believe that software development is both an art and a science. Over the last 10 years, there have been major advancements in programming styles, software design, as well as requirements gathering.

Somewhere along the way we learned to stop creating massive amounts of work at the beginning of a project. We began to understand that change is nearly grunted, and the current style of design was not conducive nor friendly to change. These changes ruined estimates, then budgets, and ultimately produced software that the user had to learn to deal with rather than enjoy.

Now we have agile methodologies that put the business first. Without the business problem, there is no software solution to build. We've learned that the user's involvement in design is vital to a line of business application, and that it's best to be involved with a chosen user to show progress and to receive feedback immediately.

There's another software design style that lines right up with our new found views on the business centric software development solution. What's this I speak of? It's called DDD (Domain Driven Design) and we will discuss the design methodology in the next chapter.

n **Note** This page could be intentionally left blank, because every chapter should end on an EVEN numbered page, even if a blank needs to be added at the end.