Foundations of Programming

Karl Seguin
http://codebetter.com/blogs/karlseguin/

Part 1 - Introduction

Preamble

A few years ago I was fortunate enough to turn a corner in my programming career. The opportunity for solid mentoring presented itself, and I took full advantage of it. Within the space of a few months, my programming skills grew exponentially and over the last couple years, I've continued to refine my art. Doubtless I still have much to learn, and five years from now I'll look back on the code I write today and feel embarrassed. I used to be confident in my programming skill, but only once I accepted that I knew very little, and likely always would, did I start to actually understand.

My Foundations of Programming series is a collection of posts which focus on helping enthusiastic programmers help themselves. Throughout the series we'll look at a number of topics typically discussed in far too much depth to be of much use to anyone except those who already know about them. I've always seen two dominate forces in the .NET world, one heavily driven by Microsoft as a natural progression of VB6 and classic ASP (commonly referred to as The MSDN Way) and the other heavily driven by core object oriented practices and influenced by some of the best Java projects/concepts (known as ALT.NET).

In reality, the two aren't really comparable. The MSDN Way loosely defines a specific way to build a system down to each individual method call (after all, isn't the API reference documentation the only reason any of us visit MSDN?) Whereas ALT.NET focuses on more abstract topics while providing specific implementation. As Jeremy Miller puts it: ***the .Net community has put too much focus on learning API and framework details and not enough emphasis on design and coding fundamentals***. For a relevant and concrete example, The MSDN Way heavily favors the use of DataSets and DataTables for all database communication. ALT.NET however, focuses on discussions about persistence design patterns, object-relational impendence mismatch as well as specific implementations such as NHibernate (O/R Mapping), MonoRail (ActiveRecord) as well as DataSets and DataTables. In other words and despite what many people think, ALT.NET isn't about ALTernatives to The MSDN Way, but rather a belief that developers should know and understand alternative solutions and approaches of which The MSDN Way is part of.

Of course, it's plain from the above description that going the ALT.NET route requires a far greater commitment as well as a wider base of knowledge. The learning curve is steep and helpful resources are just now starting to emerge (which is the reason I decided to start this series). However, the rewards are worthwhile; for me, my professional success has resulted in greater personal happiness.

Design Goals

Although simplistic, every programming decision I make is largely based on maintainability. Maintainability is the cornerstone of enterprise development. Frequent CodeBetter readers are likely sick of hearing about it, but there's a good reason we talk about maintainability so often – it's the key to being a great software developer. I can think of a couple reasons why it's such an important design factor. First, both studies and first hand experience tell us that systems spend a considerable amount of time (over 50%) in a maintenance state - be it changes, bug fixes or support. Second, the growing adoption of iterative development means that changes and features are continuously made to existing code (and even if you haven't adopted iterative development such as Agile, your clients are likely still asking you to make all types of changes.) In short, a maintainable solution not only reduces your cost, but also increases the number and quality of features you'll be able to deliver.

Even if you're relatively new to programming, there's a good chance you've already started forming opinions about what is and isn't maintainable from your experience working with others, taking over someone's application, or even trying to fix something you wrote a couple months ago. One of the most important things you can do is consciously take note when something doesn't seem quite right and google around for better solutions. For example, those of us who spent years programming in classic-ASP knew that the tight integration between code and HTML wasn't ideal.

Creating maintainable code isn't the most trivial thing. As you get started, you'll need to be extra diligent until things start to become more natural. As you might have suspected, we aren't the firsts to put some thought into creating maintainable code. To this end, there are some sound ideologies you ought to familiarize yourself with. As we go through them, take time to consider each one in depth, google them for extra background and insight, and, most importantly, try to see how they might apply to a recent project you worked on.

Simplicity

The ultimate tool in making your code maintainable is to keep it as simple as possible. A common belief is that in order to be maintainable, a system needs to be engineered upfront to accommodate any possible change request. I've seen systems built on meta-repositories (tables with a Key column and a Value column), or complex XML configurations, that are meant to handle any changes a client might throw at the team. Not only do these systems tend to have serious technical limitation (performance can be orders of magnitude slower), but they almost always fail in what they set out to do (we'll look at this more when we talk about YAGNI). In my experience, the true path to flexibility is to keep a system as simple as possible, so that you, or another developer, can easily read your code, understand it, and make the necessary change. Why build a configurable rules engine when all you want to do is check that a username is the correct length? In a later part, we'll see how Test Driven Development can help us achieve a high level of simplicity by making sure we focus on what our client is paying us to do.

YAGNI

You Aren't Going to Need It is an Extreme Programming belief that you shouldn't build something now because you think you're going to need it in the future. Experience tells us that you probably won't actually need it, or you'll need something slightly different. You can spend a month building an amazingly flexible system just to have a simple 2 line email from a client make it totally useless. Just the other day I started working on an open-ended reporting engine to learn that I had misunderstood an email and what the client really wanted was a single daily report that ended up taking 15 minutes to build.

Last Responsible Moment

The idea behind Last Responsible Moment is that you defer building something until you absolutely have to. Admittedly, in some cases, the latest responsible moment is very early on in the development phase. This concept is tightly coupled with YAGNI, in that even if you really DO need it, you should still wait to write it until you can't wait any longer. This gives you, and your client, time to make sure you really DO need it after all, and hopefully reduces the number of changes you'll have to make while and after development.

DRY

Code duplication can cause developers major headaches. They not only make it harder to change code (because you have to find all the places that do the same thing), but also have the potential to introduce serious bugs and make it unnecessarily hard for new developers to jump onboard. By following the Don't Repeat Yourself (DRY) principal throughout the lifetime of a system (user stories, design, code, unit tests and documentation) you'll end up with cleaner and more maintainable code. Keep in mind that the concept goes beyond copy-and-paste and aims at eliminating duplicate functionality/behavior in all forms. Object encapsulation and highly cohesive code can help us reduce duplication.

Explicitness and Cohesion

It sounds straightforward, but it's important to make sure that your code does exactly what it says it's going to do. This means that functions and variables should be named appropriately and using standardized casing and, when necessary, adequate documentation be provided. A Producer class ought to do exactly what you, other developers in the team and your client think it should. Additionally, your classes and methods should be highly cohesive – that is, they should have a singularity of purpose. If you find yourself writing a Customer class which is starting to manage order data, there's a good chance you need to create an Order class. Classes responsible for a multitude of distinct components quickly become unmanageable. In the next part, we'll look at object oriented programming's capabilities when it comes to creating explicit and cohesive code.

Coupling

Coupling occurs when two classes depend on each other. When possible, you

want to reduce coupling in order to minimize the impact caused by changes, and increase your code's testability. Reducing or even removing coupling is actually easier than most people think; there are strategies and tools to help you. The trick is to be able to identify undesirable coupling. We'll cover coupling in detail in a later part.

Unit Tests and Continuous Integration
Unit Testing and Continuous Integration (commonly referred to as CI) are yet another topic we have to defer for a later time. There are two things that are important for you to know beforehand. First, both are paramount in order to achieve our goal of highly maintainable code. Unit tests empower developers with an unbelievable amount of confidence. The amount of refactoring and feature changes you're able/willing to make when you have safety net of hundreds or thousands of automated tests that validate you haven't broken anything is unbelievable. Secondly, if you aren't willing to adopt, or at least try, unit testing, you're wasting your time reading this. Much of what we'll cover is squarely aimed at improving the testability of our code.

Conclusion
Although this first part was void of any actual code, we did managed to cover quite a few items. Since I want this to be more hands-on than theoretical, we'll dive head first into actual code from here on end. Hopefully we've already managed to clear up some of the buzz words you've been hearing so much about lately. The next couple parts will lay the foundation for the rest of our work by covering OOP and persistence at a high level. Until then, I hope you spend some time researching some of the key words I've thrown around. Since your own experience is your best tool, think about your recent and current projects and try to list things that didn't work out well as well as those that did.

Part 2 – Domain Domain Domain

Intruduction
Starting this kind of series by talking about domain driven design and object oriented programming is rather predictable. At first I thought I could avoid the topic for at least a couple posts, but that would do both you and me a great disservice. There are a limited number of practical ways to design the core of your system. A very common approach for .NET developers is to use a data-centric model. There's a good chance that you're already an expert with this approach – having mastered nested repeaters, the ever-useful ItemDataBound event and skillfully navigating DataRelations. Another solution which is the norm for Java developers and quickly gaining speed in the .NET community favors a domain-centric approach.

DDD2
What do I mean by data and domain-centric approaches? Data-centric generally

means that you build your system around your understanding of the data you'll be interacting with. The typical approach is to first model your database by creating all the tables, columns and foreign key relationships, and then mimicking this in C#/VB.NET. The reason this is so popular amongst .NET developers is that Microsoft spent a lot of time automating the mimicking process with DataAdapters, DataSets and DataTables. We all know that given a table with data in it, we can have a website or windows application up and running in less than 5 minutes with just a few lines of code. The focus is all about the data – which in a lot of cases is actually a good idea. This approach is sometimes called data driven development.

Domain-centric design or, as it's more commonly called, domain driven design (DDD), focuses on the problem domain as a whole – which not only includes the data, but also the behavior. So we not only focus on the fact that an employee has a FirstName, but also on the fact that he or she can get a Raise. The Problem Domain is just a fancy way of saying the business you're building a system for. The tool we use is object oriented programming (OOP) – and just because you're using an object-oriented language like C# or VB.NET doesn't mean you're necessarily doing OOP.

The above descriptions are somewhat misleading – it somehow implies that if you were using DataSets you wouldn't care about, or be able to provide, the behavior of giving employees a raise. Of course that isn't at all the case – in fact it'd be pretty trivial to do. A data-centric system isn't void of behavior nor does it treat them as an after though. DDD is simply better suited at handling complex systems in a more maintainable way for a number of reasons – all of which we'll cover in following posts. This doesn't make domain driven better than data driven – it simply makes domain driven better than data driven **in some cases** and the reverse is also true. You've probably read all of this before, and in the end, you simply have to make a leap of faith and tentatively accept what we preach – at least enough so that you can judge for yourself.

(It may be crude and a little contradictory to what I said in my introduction, but the debate between The MSDN Way and ALT.NET could be summed up as a battle between data driven and domain driven design. True ALT.NETers though, ought to appreciate that data-driven is indeed the right choice in some situations. I think much of the hostility between the “camps” is that Microsoft disproportionately favors data-driven design despite the fact that it doesn't fit well with what most .NET developers are doing (enterprise development), and, when improperly used, results in less maintainable code. Many programmers, both inside and outside the .NET community, are probably scratching their heads trying to understand why Microsoft insists on going against conventional wisdom and clumsily playing follow the leader with a 5+ year lag (witness the recent announcement of a MVC pattern slated for 2008)).

Users, Clients and Stakeholders

Something which I take very seriously from Agile development is the close interaction the development team has with clients and users. In fact, whenever possible, I don't see it as the development team and the client, but a single entity: the team. Whether you're fortunate enough or not to be in such a situation (sometimes lawyers get in the way, sometimes clients aren't available for that much commitment, etc.) it's important to understand what everyone brings to the table. The client is the person who pays the bills and as such, should make the final decisions about features and priorities. Users actually use the system. Clients are oftentimes users, but rarely are they the only user. A website for example might have anonymous users, registered users, moderators and administrators. Finally, stakeholders consist of anyone with a stake in the system. The same website might have a sister or parent site, advertisers, PR or domain experts.

Clients have a very hard job. They have to objectively prioritize the features everyone wants, including their own and deal with their finite budget. Obviously they'll make wrong choices, maybe because they don't fully understand a user's need, maybe because you made a mistake in the information you provided, or maybe because they improperly give higher priority to their own needs over everyone elses (a lot like the big screen TV I bought my girlfriend for her birthday). As a developer, it's your job to help them out as much as possible and deliver on their needs.

Whether you're building a commercial system or not, the ultimate measure of its success will likely be how users feel about it. So while you're working closely with your client, hopefully both of you are working towards your users needs. If you and your client are serious about building systems for users, I strongly encourage you to read up on User Stories – a good place to start is Mike Cohn's excellent User Stories Applied.

Finally, and the main reason this little section exists, are domain experts. Domain experts are the people who know all the ins and outs about the world your system will live in. I was recently part of a very large development project for a financial institute and there were literally hundreds of domain experts most of which being economists or accountants. These are people who are as enthusiastic about what they do as you are about programming. Anyone can be a domain expert – a clients, a user, a stakeholder and, eventually, even you. Your reliance on domain experts grows with the complexity of a system.

The Domain Object

As I said earlier, object oriented programming is the tool we'll use to make our domain-centric design come to life. Specifically, we'll rely on the power of classes and encapsulation. In this part we'll focus on the basics of classes and some

tricks to get started – many developers will already know everything covered here. We won't cover persistence (talking to the database) just yet. If you're new to this kind of design, you might find yourself constantly wondering about the database and data access code. Try not to worry about it too much. In the next part we'll cover the basics of persistence, and in following parts, we'll look at persistence in even greater depth.

The idea behind domain driven design is to build your system in a manner that's reflective of the actual problem domain you are trying to solve. This is where domain experts come into play – they'll help you understand how the system currently works (even if it's a manual paper process) and how it ought to work. At first you'll be overwhelmed by their knowledge – they'll talk about things you've never heard about and be surprised by your dumbfounded look. They'll use so many acronyms and special words that'll you'll begin to question whether or not you're up to the task. Ultimately, this is the true purpose of an enterprise developer – to understand the problem domain. You already know how to program, but do you know how to program the specific inventory system you're being asked to do? Someone has to learn someone else's world, and if domain experts learn to program, we're all out of jobs.

Anyone who's gone through the above knows that learning a new business is the most complicated part of any programming job. For that reason, there are real benefits to making our code resemble, as much as possible, the domain. Essentially what I'm talking about is communication. If your users are talking about Strategic Outcomes, which a month ago meant nothing to you, and your code talks about StrategicOutcome then some of the ambiguity and much of the potential misinterpretation is cleaned up. Many people, myself included, believe that a good place to start is with key noun-words that your business experts and users use. If you were building a system for a car dealership and you talked to a salesman (who is likely both a user and a domain expert), he'll undoubtedly talk about Clients, Cars, Models, Packages and Upgrades, Payments and so on. As these are the core of his business, it's logical that they be the core of your system. Beyond noun-words is the convergence on the language of the business – which has come to be known as the ubiquitous language (ubiquitous means present everywhere). The idea being that a single shared language between users and system is easier to maintain and less likely to be misinterpreted.

Exactly how you start is really up to you. Doing Domain Driven Design doesn't necessarily mean you have to start with modeling the domain (although it's a good idea!), but rather it means that you should focus on the domain and let it drive your decisions. At first you may very well start with your data model, when we explore test driven development we'll take a different approach to building a system that fits very well with DDD. For now though, let's assume we've spoken to our client and a few salespeople, we've realized that a major pain-point is keeping track of the inter-dependency between upgrade options. The first thing we'll do is create four classes:

```
1.  public class Car { }
2.  public class Mode { }
3.  public class Package { }
4.  public class Upgrade { }
```

Next we'll fill-in these classes with some safe assumptions:

```
5.  using System.Collections.Generic;
6.  using System.Collections.ObjectModel;
7.
8.  public class Car
9.  {
10.    private Model _model;
11.    private List<Upgrade>  upgrades;
12.
13.    public void Add(Upgrade upgrade)
14.    {
15.       //todo
16.    }
17. }
18.
19. public class Model
20. {
21.    private int _id;
22.    private int _year;
23.    private string _name;
24.
25.    public ReadOnlyCollection<Upgrade> GetAvailableUpgrades()
26.    {
27.       //todo
28.       return null;
29.    }
30. }
31.
32. public class Upgrade
33. {
34.    private int _id;
35.    private string _name;
36.
37.    public ReadOnlyCollection<Upgrade> RequiredUpgrades
38.    {
39.       get
40.       {
41.          //todo
42.          return null;
43.       }
44.    }
45. }
```

Things are quite simple. We've added some pretty traditional fields (id, name), some references (both Cars and Models have Upgrades), and an Add function to the Car class. Now we can make slight modifications and start writing a bit of actual behavior.

```
46. using System.Collections.Generic;
47. using System.Collections.ObjectModel;
48.
49. public class Car
50. {
51.   private Model  model;
52.   //todo where to initialize this?
53.   private List<Upgrade> _upgrades;
54.
55.   public void Add(Upgrade upgrade)
56.   {
57.      _upgrades.Add(upgrade);
```

```
58.   }
59.   public ReadOnlyCollection<Upgrade> MissingUpgradeDependencies()
60.   {
61.      List<Upgrade> missingUpgrades = new List<Upgrade>();
62.      foreach (Upgrade upgrade in _upgrades)
63.      {
64.         foreach (Upgrade dependentUpgrade in upgrade.RequiredUpgrades)
65.         {
66.            if (!_upgrades.Contains(dependentUpgrade) && !missingUpgrades.Contains(
    dependentUpgrade))
67.            {
68.               missingUpgrades.Add(dependentUpgrade);
69.            }
70.         }
71.      }
72.      return missingUpgrades.AsReadOnly();
73.   }
74. }
```

First, we've implemented the Add method. Next we've implemented a method that lets us retrieve all missing upgrades. Again, this is just a first step; the next step could be to track which upgrades are responsible for causing missing upgrades, i.e. You must select 4 Wheel Drive to go with your Traction Control; however, we'll stop for now. The purpose was just to highlight how we might get started and what that start might look like.

UI

You might have noticed that we haven't talked about UIs yet. That's because our domain is independent of the presentation layer – it can be used to power a website, a windows application or a windows service. The last thing you want to do is intermixed your presentation and domain logic. Doing so won't only result in hard-to-change and hard-to-test code, but it'll also make it impossible to re-use our logic across multiple UIs (which might not be a concern, but readability and maintainability always is). Sadly though, that's exactly what many ASP.NET developers do – intermix their UI and domain layer. I'd even say it's common to see behavior throughout ASP.NET button click handlers and page load events. The ASP.NET page framework is meant to control the ASP.NET UI – not to implement behavior. The click event of the Save button shouldn't validate complex business rules (or worse, hit the database directly), rather its purpose is to modify the ASP.NET page based on the results on the domain layer – maybe it ought to redirect to another page, display some error messages or request additional information.

Remember, you want to write cohesive code. Your ASP.NET logic should focus on doing one thing and doing it well – I doubt anyone will disagree that it has to manage the page, which means it can't do domain functionality. Also, logic placed in codebehind will typically violate the Don't Repeat Yourself principal, simply because of how difficult it is to reuse the code inside an aspx.cs file. With that said, you can't wait too long to start working on your UI. First of all, we want to get client and user feedback as early and often as possible. I doubt they'll be very impressed if we send them a bunch of .cs/.vb files with our classes.

Secondly, making actual use of your domain layer is going to reveal some flaws and awkwardness. For example, the disconnected nature of the web might mean we have to make little changes to our pure OO world in order to achieve a better user experience. In my experience, unit tests are too narrow to catch these quirks while they are plainly visible as you create your real UI.

You'll also be happy to know that ASP.NET and WinForms deal with domain-centric code just as well as with data-centric classes. You can databind to any .NET collection, use sessions and caches like you normally do, and anything else you're used to doing. In fact, out of everything, the impact on the UI is probably the least significant. Of course, it shouldn't surprise you to know that ALT.NET'ers also think you should keep your mind open when it comes to your presentation engine. The ASP.NET Page Framework isn't necessary the best tool for the job – a lot of us consider it unnecessarily complicated and brittle. We'll talk about this more in a later part, but if you're interested to find out more, I suggest you look at MonoRails (which is a Rails framework for .NET) and find out about the new page framework being released by Microsoft in 2008. The last thing I want is for anyone to get discouraged with the vastness of changes, so for now, let's get back on topic.

Tricks and Tips

We'll finish off by looking at some useful things we can do with classes. We'll only cover the tip of the iceberg, but hopefully the information will help you get off on the right foot.

Factory Pattern

What do we do when a Client buys a new Car? Obviously we need to create a new instance of Car and specify the model. The traditional way to do this is to use a constructor and simply instantiate a new object with the new keyword. A different approach is to use a factory to create the instance:

```
75. using System.Collections.Generic;
76. public class Car
77. {
78.   private Model  model;
79.   private List<Upgrade> _upgrades;
80.
81.   private Car()
82.   {
83.      _upgrades = new List<Upgrade>();
84.   }
85.   public static Car CreateCar(Model model)
86.   {
87.      Car car = new Car();
88.      car. model = model;
89.      return car;
90.   }
91. }
```

There are two advantages to this approach. First, we can return a null object, which is impossible to do with a constructor – this may or may not be useful in your particular case. Secondly, if there are a lot of different ways to create an object, it gives you the change to provide more meaningful function names. The first example that comes to mind is when you want to create an instance of a User class, you'll likely have User.CreateByCredentials(string username, string password), User.CreateById(int id) and User.GetUsersByRole(string role). You can accomplish the same functionality with constructor overloading, but rarely with the same clarity. Truth be told, I always have a hard time deciding which to use, so it's really a matter of taste and gut feeling.

Interfaces

Interfaces will play a big part in helping us create maintainable code. We'll use them to decouple our code as well as create mock classes for unit testing. An interface is a contract which any implementing classes must adhere to. Let's say that we want to encapsulate all our database communication inside a class called SqlServerDataAccess such as:

```
92. using System.Collections.Generic;
93. internal class SqlServerDataAccess
94. {
95.   internal List<Upgrade> RetrieveAllUpgrades()
96.   {
97.      //todo implement
98.      return null;
99.   }
100.       }
101.
102.       public class Sample
103.       {
104.         public void SampleMethod()
105.         {
106.            SqlServerDataAccess da = new SqlServerDataAccess();
107.            List<Upgrade> upgrades = da.RetrieveAllUpgrades();
108.         }
109.       }
```

You can see that the sample code at the bottom has a direct reference to SqlServerDataAccess – as would the many other methods that need to communicate with the database. This highly coupled code is problematic to change and difficult to test (we can't test SampleMethod without having a fully functional RetrieveAllUpgrades method). We can relieve this tight coupling by programming against an interface instead:

```
110.       using System.Collections.Generic;
111.       internal interface IDataAccess
112.       {
113.         List<Upgrade> RetrieveAllUpgrades();
114.       }
115.
116.       internal class DataAccess
117.       {
118.         internal static IDataAccess CreateInstance()
119.         {
120.            return new SqlServerDataAccess();
```

```
121.          }
122.        }
123.
124.        internal class SqlServerDataAccess : IDataAccess
125.        {
126.          public List<Upgrade> RetrieveAllUpgrades()
127.          {
128.              //todo implement
129.              return null;
130.          }
131.        }
132.
133.        public class Sample
134.        {
135.          public void SampleMethod()
136.          {
137.              IDataAccess da = DataAccess.CreateInstance();
138.              List<Upgrade> upgrades = da.RetrieveAllUpgrades();
139.          }
140.        }
```

We've introduced the interface along with a helper class to return an instance of that interface. If we want to change our implementation, say to an OracleDataAccess, we simply create the new Oracle class, make sure it implements the interface, and change the helper class to return it instead. Rather than having to change multiple (possibly hundreds), we simply have to change one.

This is only a simple example of how we can use interfaces to help our cause. We can beef up the code by dynamically instantiating our class via configuration data or introducing framework specially tailored for the job (which is exactly what we're going to do). We'll often favor programming against interfaces over actual classes, so if you aren't familiar with them, I'd suggest you do some extra reading.

Information Hiding and Encapsulation

Information hiding is the principle that design decisions should be hidden from other components of your system. It's generally a good idea to be as secretive as possible when building classes and components so that changes to implementation don't impact other classes and components. Encapsulation is an OOP implementation of information hiding. Essentially it means that your objects data (the fields) and as much as the implementation should not be accessible to other classes. The most common example is making fields private with public properties. Even better is to ask yourself if the _id field even needs a public property to begin with.

Access Modifiers

As you focus on writing classes that encapsulate the behavior of the business a rich API is going to emerge for your UI to consume. It's a good idea to keep this API clean and understandable. The simplest method is to keep your API small by hiding all but the most necessary methods. Some methods clearly need to be

public and others private, but if ever you aren't sure, pick a more restrictive access modifier and only change it when necessary. I make good use of the internal modifier on many of my methods and properties. Internal members are only visible to other members within the same assembly – so if you're physically separating your layers across multiple assemblies (which is generally a good idea), you'll greatly minimize your API.

Conclusion

The reason enterprise development exists is that no single off-the-shelf product can successfully solve the needs of a complex system. There are simply too many odd or intertwined requirements and business rules. To date, no paradigm has been better suited to the task than object oriented programming. In fact, OOP was designed with the specific purpose of letting developers model actual systems in code. It may still be difficult to see the long-term value of domain driven design. Sharing a common language with your client and users in addition to having greater testability may not seem necessary. Hopefully as you go through the remaining parts and experiment on your own, you'll start adopting some of the concepts and tweaking them to fit your and your clients needs.

Part 3 - Persistence

The Foundations of Programming series looks at a number of key concepts, techniques and tools specifically designed to help developers meet the growing complexity of enterprise systems. Based on proven principals like unit testing, domain driven design, dependency injection and O/R Mappers, the series is aimed at developers interested in helping themselves.

Writing maintainable code that delivers value to your client isn't trivial. That doesn't mean that being a successful enterprise developer has to be hard. The Foundations of Programming series looks at a number of key concepts, techniques and tools specifically designed to help developers meet the growing complexity of enterprise systems. Based on proven principals like unit testing, domain driven design, dependency injection and O/R Mappers, the series is aimed at developers interested in helping themselves. Readers are encouraged to first read Part1 and Part 2.

Introduction

In the previous part we managed to have a good discussion about DDD without talking much about databases. If you're used to programming with DataSets, you probably have a lot of questions about how this is actually going to work. DataSets are great in that a lot is taken care of for you. In this part we'll start the discussion around how to deal with persistence using DDD. We'll manually write code to bridge the gap between our C# objects and our SQL tables. In later sections we'll look at more advanced alternatives (two different O/R mapping approaches)

which, like DataSets, do much of the heavy lifting for us. This part is meant to bring some closure to the previous discussion while opening the discussion on more advanced persistence patterns.

The Gap

As you know, your program runs in memory and requires a place to store (or persist) information. These days, the solution of choice is a relational database. Persistence is actually a pretty big topic in the software development field because, without the help of patterns and tools, it isn't the easiest thing to successfully pull off. With respect to object oriented programming the challenge has been given a fancy name: the Object-Relational Impedance Mismatch. That pretty much means that relational data doesn't map perfectly to objects and objects don't map perfectly to relational stores. Microsoft basically tried to ignore this problem and simply made a relational representation within object-oriented code - a clever approach, but not without its flaws such as poor performance, leaky abstractions, poor testability, awkwardness, and poor maintainability. (On the other side are object oriented databases which, to the best of my knowledge, haven't taken off either.)

Rather than try to ignore the problem, we can, and should face it head on. We should face it so that we can leverage the best of both worlds - complex business rules implemented in OOP and data storage and retrieval via relational databases. Of course, that is providing that we can bridge the gap. But what gap exactly? What is this Impedance Mismatch? You're probably thinking that it can't be that hard to pump relational data into objects and back into tables. If you are, then you're absolutely right (mostly right anyways for now let's assume that it's always a simple process).

DataMapper

For small projects with only a handful of small domain classes and database tables, my preference has generally been to manually write code that maps between the two worlds. Let's look at a simple example. The first thing we'll do is expand on our Upgrade class (we're only focusing on the data portions of our class (the fields) since that's what gets persisted):

Listing 1: Fields and Properties of the Upgrade class

```
141.       public class Upgrade
142.       {
143.          private int _id;
144.          private string  name;
145.          private string _description;
146.          private decimal _price;
147.          private List<Upgrade> _requiredUpgrades;
148.
149.          public int Id
150.          {
151.             get { return _id; }
```

```
152.             internal set { _id = value; }
153.          }
154.          public string Name
155.          {
156.             get { return _name; }
157.             set { _name = value; }
158.          }
159.          public string Description
160.          {
161.             get { return  description; }
162.             set { _description = value; }
163.          }
164.          public decimal Price
165.          {
166.             get { return _price; }
167.             set {  price = value; }
168.          }
169.
170.          public List<Upgrade> RequiredUpgrades
171.          {
172.             get { return _requiredUpgrades; }
173.          }
174.       }
```

We've added the basic fields you'd likely expect to see in the class. Next we'll create the table that would hold, or persist, the upgrade information

Listing 2: CREATE TABLE Upgrades

```
175.       CREATE TABLE Upgrades
176.       (
177.           Id INT IDENTITY(1,1) NOT NULL PRIMARY KEY,
178.           [Name] VARCHAR(64) NOT NULL,
179.           Description VARCHAR(512) NOT NULL,
180.           Price MONEY NOT NULL,
181.        )
```

No surprises there. Now comes the interesting part (well, relatively speaking), we'll start to build up our data access layer, which sits between the domain and relational models (interfaces left out for brevity)

Listing 3: Simple DAL

```
182.       using System.Collections.Generic;
183.       using System.Data;
184.       using System.Data.SqlClient;
185.
186.       internal class SqlServerDataAccess
187.       {
188.          private readonly static string  connectionString = "GET FROM CONFIG OR
    SOMETHING";
189.
190.          internal List<Upgrade> RetrieveAllUpgrades()
191.          {
192.             //use a sproc if you prefer
193.             string sql = "SELECT Id, Name, Description, Price FROM Upgrades";
194.             using (SqlCommand command = new SqlCommand(sql))
195.             using (SqlDataReader dataReader = ExecuteReader(command))
196.             {
197.                List<Upgrade> upgrades = new List<Upgrade>();
198.                while (dataReader.Read())
199.                {
200.                   upgrades.Add(DataMapper.CreateUpgrade(dataReader));
201.                }
```

```
202.                return upgrades;
203.             }
204.          }
205.
206.
207.          private SqlDataReader ExecuteReader(SqlCommand command)
208.          {
209.             SqlConnection connection = new SqlConnection( connectionString);
210.             command.Connection = connection;
211.             connection.Open();
212.             return command.ExecuteReader(CommandBehavior.CloseConnection)

213.          }
214.       }
```

ExecuteReader is a helper method to slightly reduce the redundant code we have to write. RetrieveAllUpgrades is more interesting as it selects all the upgrades and loads them into a list via the DataMapper.CreateUpgrade function. CreateUpgrade, shown below, is the reusable code we use to map upgrade information stored in the database into our domain. It's straightforward because the domain model and data model are so similar.

Listing 4: DataMapper From Data to Object

```
215.       using System;
216.       using System.Data;
217.
218.       internal static class DataMapper
219.       {
220.          internal static Upgrade CreateUpgrade(IDataReader dataReader)
221.          {
222.             Upgrade upgrade = new Upgrade();
223.             upgrade.Id = Convert.ToInt32(dataReader["Id"]);
224.             upgrade.Name = Convert.ToString(dataReader["Name"]);
225.             upgrade.Description = Convert.ToString(dataReader["Description"]);
226.             upgrade.Price = Convert.ToDecimal(dataReader["Price"]);
227.             return upgrade;
228.          }
229.       }
```

If we need to, we can re-use CreateUpgrade as much as necessary. For example, we'd likely need the ability to retrieve upgrades by id or price - both of which would be new methods in the SqlServerDataAccess class.

Obviously, we can apply the same logic when we want to store Upgrade objects back into the store. Here's one possible solution:

Listing 5: DataMapper From Object to Data

```
230.       using System.Data;
231.       using System.Data.SqlClient;
232.
233.       internal static class DataMapper
234.       {
235.          internal static SqlParameter[] ConvertUpgradeToParameters(Upgrade upgra
    de)
236.          {
237.             SqlParameter[] parameters = new SqlParameter[4];
238.             parameters[0] = new SqlParameter("Id", SqlDbType.Int);
239.             parameters[0].Value = upgrade.Id;
```

```
            parameters[1] = new SqlParameter("Id", SqlDbType.Int);
            parameters[1].Value = upgrade.Id;
            parameters[2] = new SqlParameter("Description", SqlDbType.VarChar, 5
    12);
            parameters[2].Value = upgrade.Id;
            parameters[3] = new SqlParameter("Price", SqlDbType.Money);
            parameters[3].Value = upgrade.Price;
            return parameters;
        }
    }
```

!!!Mismatch Detected!!!

Despite the fact that we've taken a very simple and common example, we still ran into the dreaded impedance mismatch. Notice how our data access layer (either the SqlServerDataAccess or DataMapper) doesn't handle the much needed RequiredUpgrades collection. That's because one of the trickiest things to handle are relationships. In the domain world these are references (or collection of references) to other objects; whereas the relational world uses foreign keys. This difference is a constant thorn in the side of developers. The fix isn't too hard. First we'll add a many-to-many join table which associates an upgrade with the other upgrades that are required for it (could be 0, 1 or more).

Listing 6: CREATE TABLE UpgradeDepencies

```
    CREATE TABLE UpgradeDepencies
     (
       UpgradeId INT NOT NULL,
       RequiredUpgradeId INT NOT NULL,
     )
```

Next we modify RetrieveAllUpgrade to load-in required upgrades:

Listing 7: Improved DAL

```
    using System.Collections.Generic;
    using System.Data;
    using System.Data.SqlClient;

    internal List<Upgrade> RetrieveAllUpgrades()
    {
        string sql = @"SELECT Id, Name, Description, Price FROM Updagrades;
                SELECT UpgradeId, RequiredUpgradeId FROM UpgradeDepencies";
        using (SqlCommand command = new SqlCommand(sql))
        using (SqlDataReader dataReader = ExecuteReader(command))
        {
            List<Upgrade> upgrades = new List<Upgrade>();
            Dictionary<int, Upgrade> localCache = new Dictionary<int, Upgrade>();
            while (dataReader.Read())
            {
                Upgrade upgrade = DataMapper.CreateUpgrade(dataReader);
                upgrades.Add(upgrade);
                localCache.Add(upgrade.Id, upgrade);
            }
            dataReader.NextResult();
            while (dataReader.Read())
            {
                int upgradeId = dataReader.GetInt32(0);
```

```
277.                int requiredUpgradeId = dataReader.GetInt32(1);
278.                Upgrade upgrade;
279.                Upgrade requiredUpgrade;
280.                if (!localCache.TryGetValue(upgradeId, out upgrade) || !localCache.
    TryGetValue(requiredUpgradeId, out requiredUpgrade))
281.                {
282.                    //probably should throw an exception
283.                    //since our db is in a weird state
284.                    continue;
285.                }
286.                upgrade.RequiredUpgrades.Add(requiredUpgrade);
287.            }
288.            return upgrades;
289.        }
290.    }
```

We pull the extra join table information along with our initial query and create a local lookup dictionary to quickly access our upgrades by their id. Next we loop through the join table, get the appropriate upgrades from the lookup dictionary and add them to the collections.

It isn't the most elegant solution, but it works rather well. We may be able to refactor the function a bit to make it little more readable, but for now and for this simple case, it'll do the job.

Limitations

Although we're only doing an initial look at mapping, it's worth it to look at the limitations we've placed on ourselves. Once you go down the path of manually writing this kind of code it can quickly get out of hand. If we want to add filtering/sorting methods we either have to write dynamic SQL or have to write a lot of methods. We'll end up writing a bunch of RetrieveUpgradeByX methods that'll be painfully similar from one and other.

Oftentimes you'll want to lazy-load relationships. That is, instead of loading all the required upgrades upfront, maybe we want to load them only when necessary. In this case it isn't a big deal since it's just an extra 32bit reference. A better example would be the Model's relationship to Upgrades. It is relatively easy to implement lazy loads, it's just, yet again, a lot of repetitive code.

The most significant issue though has to do with identity. If we call RetrieveAllUpgrades twice, we'll get to distinct instances of every upgrade. This can result in inconsistencies, given:

Listing 8: Oh oh!

```
291.        SqlServerDataAccess da = new SqlServerDataAccess();
292.        Upgrade upgrade1a = da.RetrieveAllUpgrades()[0];
293.        Upgrade upgrade1b = da.RetrieveAllUpgrades()[0];
294.
295.        upgrade1b.Price += 2000;
296.        upgrade1b.Save();
297.        //upgrade1a is still $2000 cheaper, get it while you can!
```

The price change to the first upgrade won't be reflected in the instance pointed to by upgrade1a. In some cases that won't be a problem. However, in many situations, you'll want your data access layer to track the identity of instances it creates and enforce some control (you can read more by googling the Identity Map pattern).

There are probably more limitations, but the last one we'll talk about has to do with units of work (again, you can read more by goolging the Unit of Work pattern). Essentially when you manually code your data access layer, you need to make sure that when you persist an object, you also persist, if necessary, updated referenced object. If you're working on the admin portion of our car sales system, you might very well create a new Model and add a new Upgrade. If you call Save on your Model, you need to make sure your Upgrade is also saved. The simplest solution is to call save often for each individual action - but this is both difficult (relationships can be several levels deep) and inefficient. Similarly you may change only a few properties and then have to decide between resaving all fields, or somehow tracking changed properties and only updating those. Again, for small systems, this isn't much of a problem. For larger systems, it's a near impossible task to manually do (besides, rather than wasting your time building your own unit of work implementation, maybe you should be writing functionality the client asked for).

Conclusion

In the end, we won't rely on manual mapping - it just isn't flexible enough and we end up spending too much time writing code that's useless to our client. Nevertheless, it's important to see mapping in action - and even though we picked a simple example, we still ran into some issues. Since mapping like this is straightforward, the most important thing is that you understand the limitations this approach has. Try thinking what can happen if two distinct instances of the same data are floating around in your code, or just how quickly your data access layer will balloon as new requirements come in. We won't revisit persistence for at least a couple parts - but when we do re-address it we'll examine full-blown solutions that pack quite a punch.

Part 4 – Dependency Injection

It's common to hear developers promote layering as a means to provide extensibility. The most common example, and one I used in Part 2 when we looked at interfaces, is the ability to switch out your data access layer in order to connect to a different database. If your projects are anything like mine, you know upfront what database you're going to use and you know you aren't going to have to change it. Sure, you could build that flexibility upfront - just in case - but what about keeping things simple and You Aren't Going To Need IT (YAGNI)?

I used to write about the importance of domain layers in order to have re-use across multiple presentation layers: website, windows applications and web services. Ironically, I've rarely had to write multiple front-ends for a given domain layer. I still think layering is important, but my reasoning has changed. I now see layering as a natural by-product of highly cohesive code with at least some thought put into coupling. That is, if you build things right, it should automatically come out layered.

The real reason we're spending a whole part on decoupling (which layering is a high-level implementation of) is because it's a key ingredient in writing testable code. It wasn't until I started unit testing that I realized how tangled and fragile my code was. I quickly became frustrated because method X relied on a functional class Y which needed a database up and running. In order to avoid the headaches I went through, we'll first cover coupling and then look at unit testing in the next part.

(A point about YAGNI. While many developers consider it a hard rule, I rather think of it as a general guideline. There are good reasons why you want to ignore YAGNI, the most obvious is your own experience. If you know that something will be hard to implement later, it might be a good idea to build it now, or at least put hooks in place. This is something I frequently do with caching, building an ICacheProvider and a NullCacheProvider implementation that does nothing, except provide the necessary hooks for a real implementation later on. That said, of the numerous guidelines out there, YAGNI, DRY and Sustainable Pace are easily the three I consider the most important.)

Sneak Peak at Unit Testing

Talking about coupling with respect to unit testing is something of a chicken and egg problem – which to talk about first. I think it's best to move ahead with coupling, providing we cover some basics about unit testing. Most importantly is that unit tests are all about the unit. You aren't focusing on end-to-end testing but rather on individual behavior. The idea is that if you test each behavior of each method thoroughly and test their interaction with one and other, you're whole system is solid. This is tricky given that the method you want to unit test might have a dependency on another class which can't be easily executed within the context of a test (such as a database, or a web-browser element). For this reason, unit testing makes use of mock classes – or pretend class.

Let's look at an example, saving a car's state:

```
298.        public class Car
299.        {
300.            private int id;
301.            public void Save()
302.            {
303.                if (!IsValid())
304.                {
305.                    //todo: come up with a better exception
306.                    throw new InvalidOperationException("The car must be in a valid state");
```

```
307.             }
308.             if ( id == 0)
309.             {
310.                 _id = DataAccess.CreateInstance().Save(this);
311.             }
312.             else
313.             {
314.                 DataAccess.CreateInstance().Update(this);
315.             }
316.         }
317.         private bool IsValid()
318.         {
319.             //todo: make sure the object is in a valid state
320.             return true;
321.         }
322.     }
```

To effectively test the Save method, there are three things we must do:

1. Make sure the correct exception is thrown when we try to save a car which is in an invalid state,
2. Make sure the data access' save method is called when it's a new car, and
3. Make sure the Update method is called when it's an existing car.

What we don't want to do (which is just as important as what we do want to do), is test the functionality of IsValid or the data access' Save and Update functions (other tests will take care of those). The last point is important – all we want to do is make sure these functions are called with the proper parameters and their return value (if any) is properly handled. It's hard to wrap your head around mocking without a concrete example, but mocking frameworks will let us intercept the Save and Update calls, ensure that the proper arguments were passed, and force whatever return value we want. Mocking frameworks are quite fun and effective....unless you can't use them because your code is tightly coupled.

Not ALL coupling is bad

In case you forgot from Part 1, coupling is simply what we call it when one class requires another class in order to function. It's essentially a dependency. All but the most basic lines of code are dependent on other classes. Heck, if you write *string site = "CodeBetter"*, you're coupled to the System.String class – if it changes, your code could very well break. Of course the first thing you need to know is that in the vast majority of cases, such as the silly string example, coupling isn't a bad thing. We don't want to create interfaces and providers for each and every one of our classes. It's ok for our Car class to hold a direct reference to the Upgrade class – at this point it'd be overkill to introduce and IUpgrade interface. What isn't ok is any coupling to an external component (database, state server, cache server, web service), any code that requires extensive setup (database schemas) and, as I learnt on my last project, any code that generates random output (password generation, key generators). That might be a somewhat vague description, but after this and the next part, and once you

play with unit testing yourself, you'll get a feel for what should and shouldn't be avoided.

Since it's always a good idea to decouple your database from your domain, we'll use that as the example throughout this part.

Dependency Injection
In Part 2 we saw how interfaces can help our cause – however, the code provided didn't allow us to dynamically provide a mock implementation of IDataAccess for the DataAccess factory class to return. In order to achieve this, we'll rely on a pattern called Dependency Injection (DI). DI is specifically tailored for the situation because, as the name implies, it's a pattern that turns a hard-coded dependency into something that can be injected at runtime. We'll look at two forms of DI, one which we manually do, and the other which leverages a third party library.

Constructor Injection
The simplest form of DI is constructor injection – that is, injecting dependencies via a class' constructor. First, let's look at our DataAccess interface again and create a fake (or mock) implementation (don't worry, you won't actually have to create mock implementations of each component, but for now it keeps things obvious):

```
internal interface IDataAccess
{
   int Save(Car car);
   void Update(Car car);
}
internal class MockDataAccess : IDataAccess
{
   private readonly List<Car> _cars = new List<Car>();
   public int Save(Car car)
   {
      _cars.Add(car);
      return _cars.Count;
   }
   public void Update(Car car)
   {
      _cars[_cars.IndexOf(car)] = car;
   }
}
```

Although our mock's upgrade function could probably be improved, it'll do for now. Armed with this fake class, only a minor change to the Car class is required:

```
public class Car
{
   private int _id;
   private IDataAccess  dataProvider;
   public Car() : this(new SqlServerDataAccess())
   {
   }
   internal Car(IDataAccess dataProvider)
   {
       dataProvider = dataProvider;
   }
   public void Save()
```

```
353.           {
354.              if (!IsValid())
355.              {
356.                 //todo: come up with a better exception
357.                 throw new InvalidOperationException("The car must be in a valid s
    tate");
358.              }
359.              if ( id == 0)
360.              {
361.                  id =  dataProvider.Save(this);
362.              }
363.              else
364.              {
365.                 _dataProvider.Update(this);
366.              }
367.           }
368.           private bool IsValid()
369.           {
370.              //todo: make sure the object is in a valid state
371.              return true;
372.           }
373.        }
```

Take a good look at the code above and follow it through. Notice the clever use of constructor overloading means that the introduction of DI doesn't have any impact on existing code – if you choose not to inject an instance of IDataAccess, the default implementation is used for you. On the flip side, if we do want to inject a specific implementation, such as a MockDataAccess instance, we can:

```
374.        public void AlmostATest()
375.        {
376.           Car car = new Car(new MockDataAccess());
377.           car.Save();
378.           if (car.Id != 1)
379.           {
380.              //something went wrong
381.           }
382.        }
```

There are minor variations available – we could have injected an IDataAccess directly in the Save method or could set the private _dataAccess field via an internal property – which you use is mostly a matter of taste.

Frameworks

Doing DI manually works great in simple cases, but can become unruly in more complex situations. A recent project I worked on had a number of core components that needed to be injected – one for caching, one for logging, one for a database access and another for a web service. Classes got polluted with multiple constructor overloads and too much thought had to go into setting up classes for unit testing. Since DI is so critical to unit testing, and most unit testers love their open-source tools, it should come as no surprise that a number of frameworks exist to help automate DI. The rest of this article will focus on StructureMap, a Dependency Injection framework created by fellow CodeBetter blogger Jeremy Miller. (http://structuremap.sourceforge.net/)

Before using StructureMap you must configure it using an XML file (called StructureMap.config) or by adding attributes to your classes. The configuration

essentially says this is the interface I want to program against and here's the default implementation. The simplest of configurations to get StructureMap up and running would look something like:

```
<StructureMap>
  <DefaultInstance PluginType="CodeBetter.Foundations.IDataAccess, CodeBetter.Foundations"
                   PluggedType="CodeBetter.Foundations.SqlServerDataAccess, CodeBetter.Foundations" />
</StructureMap>
```

While I don't want to spend too much time talking about configuration, it's important to note that the XML file must be deployed in the /bin folder of your application. You can automate this in VS.NET by selecting the files, going to the properties and setting the Copy To Ouput Directory attribute to Copy Always. (There are a variety of more advanced configuration options available. If you're interested in learning more, I suggest the StructureMap website).

Once configured, we can undo all the changes we made to the Car class to allow constructor injection (remove the _dataProvider field, and the constructors). To get the correct IDataAccess implementation, we simply need to ask StructureMap for it, the Save method now looks like:

```
public class Car
{
    private int id;
    public void Save()
    {
        if (!IsValid())
        {
            //todo: come up with a better exception
            throw new InvalidOperationException("The car must be in a valid state");
        }
        IDataAccess dataAccess = ObjectFactory.GetInstance<IDataAccess>();
        if ( id == 0)
        {
            _id = dataAccess.Save(this);
        }
        else
        {
            dataAccess.Update(this);
        }
    }
    private bool IsValid()
    {
        //todo: make sure the object is in a valid state
        return true;
    }
}
```

To use a mock rather than the default implementation, we simply need to inject the mock into StructureMap:

```
public void AlmostATest()
{
    ObjectFactory.InjectStub(typeof(IDataAccess), new MockDataAccess());
    Car car = new Car();
    car.Save();
```

```
418.             if (car.Id != 1)
419.             {
420.                 //something went wrong
421.             }
422.             ObjectFactory.ResetDefaults();
423.         }
```

We use InjectStub so that subsequent calls to GetInstance return our mock, and make sure to reset everything to normal via ResetDefaults.

DI frameworks such as StructureMap are as easy to use as they are useful. With a couple lines of configuration and some minor changes to our code, we've greatly decreased our coupling which increased our testability. In the past, I've introduced StructureMap into existing large codebases in a matter of minutes – the impact is minor.

Conclusion

Reducing coupling is one of those things that's pretty easy to do yet yields great results towards our quest for greater maintainability. All that's required is a bit of knowledge and discipline – and of course, tools don't hurt either. It should be obvious why you want to decrease the dependency between the components of your code – especially between those components that are responsible for different aspects of the system (UI, Domain and Data being the obvious three). In the next part we'll look at unit testing which'll really leverage the benefits of dependency injection. If you're having problems wrapping your head around DI, take a look at my more detailed article on the subject at http://dotnetslackers.com/articles/designpatterns/IntroducingDependencyInjectionFrameworks.aspx

Part 5 – Unit Testing

Throughout this series we've talked about the importance of testability and have looked at techniques to make it easier to test our system. It goes without saying that a major benefit of writing tests for our system is the ability to deliver a better product to our client. Although this is true for unit tests as well, the main reason I write unit tests is that nothing comes close to improving the maintainability of a system as much as a properly developed suite of unit tests. You'll often hear unit test advocates speak of how much confidence unit tests give them – and that's what it's really all about. On a project I'm currently working on, we're continuously making changes and tweaks to improve the system (functional improvements, performance, refactorings, you name it). Being that it's a fairly large system, we're sometimes asked to make a change that flat out scares us. Is it doable? Will it have some weird side effect? What bugs will be introduced? Without unit tests, we'd likely refuse to make the higher risk changes. But we know, and our client knows, that it's the high risk changes that have the most potential for success. It turns out that having 700+ unit tests which run within a couple minutes lets us rip components apart, reorganize code, and build features we never thought about a year ago, without worrying too much about it. Because we

are confident in the completeness of the unit tests, we know that we aren't likely to introduce bugs into our production environment – our changes might still cause bugs but we'll know about them right away.

Unit tests aren't only about mitigating high-risk changes. In my programming life, I've been responsible for major bugs caused from seemingly low-risk changes as well. The point is that I can make a fundamental or minor change to our system, right click the solution, select "Run Tests" and within 2 minutes know where we stand.

Why wasn't I unit testing 3 years ago?

For those of us who've discovered the joy of unit testing, it's hard to understand why everyone isn't doing it. For those who haven't adopted it, you probably wish we'd shut up about it already. For many years I'd read blogs and speak to colleagues who were really into unit testing, but didn't practice myself. Looking back, here's why it took me a while to get on the bandwagon:

1. I had misconception about the goals of unit testing. As I've already said, unit testing does improve the quality of a system, but it's really all about making it easier to change / maintain the system later on. Furthermore, if you go to the next logical step and adopt Test Driven Development, unit testing really becomes about design. To paraphrase Scott Bellware, TDD isn't about testing because you're not thinking as a tester when doing TDD – you're thinking as a designer.
2. Like many, I used to think developers shouldn't write tests! I don't know the history behind this belief, but I now think this is just an excuse for lazy programmers. Testing is the process of both finding bugs in a system as well as validating that it works as expected. Maybe developers aren't good at finding bugs in their own code, but they are the best suited to make sure it works the way they intended it to (and clients are best suited to test that it works like it should (if you're interested to find out more about that, I suggest you research Acceptance Testing and FitNess (http://fitnesse.org/)). Even though unit testing isn't all that much about testing, developers who don't believe they should test their own code simply aren't accountable.
3. Testing isn't fun. Sitting in front of a monitor, inputting data and making sure everything's ok sucks. But unit testing is coding, which means there are a lot of challenges and metrics to gauge your success. Sometimes, like coding, it's a little mundane, but all in all it's no different than the other programming you do every day.
4. It takes time. Advocates will tell you that unit testing doesn't take time, it SAVES time. This is true in that the time you spend writing unit tests is likely to be small compared to the time you save on change requests and bug fixes. That's a little too Ivory Tower for me. In all honesty, unit testing DOES take a lot of time (especially when you just start out). You may very well not have enough time to unit test or you client might not feel the

upfront cost is justified. In these situations I suggest you identifier the most critical code and test it as thoroughly as possible – even a couple hours spent writing unit tests can have a big impact.

Ultimately, unit testing seemed like a complicated and mysterious thing that was only used in edge cases. The benefits seemed unattainable and timelines didn't seem to allow for it anyways. It turns out it took a lot of practice (I had a hard time learning what to unit test and how to go about it), but the benefits were almost immediately noticeable.

The Tools

With StructureMap already in place from the last part, we need only add 2 frameworks and 1 tool to our unit testing framework: nUnit, RhinoMocks and TestDriven.NET.

TestDriven.NET is an addon for Visual Studio that basically adds a "Run Test" option to our context (right-click) menu. We won't spend any time talking about it. The personal license of TestDriven.NET is only valid for open source and trial users. Don't worry too much if the licensing doesn't suite you, nUnit has its own test runner tool, just not integrated in VS.NET. (http://www.testdriven.net/) (Resharper users can also use its built-in functionality).

nUnit is the testing framework we'll actually use. There are alternatives, such as mbUnit, but I don't know nearly as much about them as I ought to. (http://www.nunit.org/)

RhinoMocks is the mocking framework we'll use. In the last part we manually created our mock – which was both rather limited and time consuming. RhinoMocks will automatically generate a mock class from an interface and allow us to verify and control the interaction with it. (http://www.ayende.com/projects/rhino-mocks.aspx)

nUnit

The first thing to do is to add a reference to the nunit.framework.dll and the Rhino.Mocks.dll. My own preference is to put my unit tests into their own assembly. For example, if my domain layer was located in *CodeBetter.Foundations*, I'd likely create a new assembly called *CodeBetter.Foundations.Tests*. This does mean that we won't be able to test private methods (more on this shortly). In .NET 2.0+ we can use the *InternalsVisibleToAttribute* to allow the Test assembly access to our internal method (we'd open Properties/AssemblyInfo.cs and add *[assembly: InternalsVisibleTo("CodeBetter.Foundations.Tests")]* – which is something I typically do.

There are two things you need to know about nUnit. First, you configure your tests via the use of attributes. The TestFixtureAttribute is applied to the class that

contains your tests, setup and teardown methods. The SetupAttribute is applied to the method you want to have executed before each test – you won't always need this. Similarly, the TearDownAttribute is applied to the method you want executed after each test. Finally, the TestAttribute is applied to your actual unit tests. (There are other attributes, but these 4 are the most important). This is what it might look like:

```
424.
425.       using NUnit.Framework;
426.
427.       [TestFixture]
428.       public class CarTests
429.       {
430.          [SetUp]
431.          public void SetUp()
432.          {
433.             //todo
434.          }
435.          [TearDown]
436.          public void TearDown()
437.          {
438.             //todo
439.          }
440.          [Test]
441.          public void SaveThrowsExceptionWhenInvalid()
442.          {
443.             //todo
444.          }
445.          [Test]
446.          public void SaveCallsDataAccessAndSetsId()
447.          {
448.             //todo
449.          }
450.          //more tests
451.       }
```

Notice that each unit test has a very explicit name – it's important to state exactly what the test is going to do, and since your test should never do too much, you'll rarely have obscenely long names.

The second thing to know about nUnit is that you confirm that your test executed as expected via the use of the Assert class and its many methods. I know this is lame, but if we had a method that took a param int[] numbers and returned the sum, our unit test would look like:

```
452.        [TestFixture]
453.       public class MathUtilityTester
454.       {
455.         [Test]
456.         public void MathUtilityReturnsZeroWhenNoParameters()
457.         {
458.            Assert.AreEqual(0, MathUtility.Add());
459.         }
460.         [Test]
461.         public void MathUtilityReturnsValueWhenPassedOneValue()
462.         {
463.            Assert.AreEqual(10, MathUtility.Add(10));
464.         }
465.         [Test]
466.         public void MathUtilityReturnsValueWhenPassedMultipleValues()
467.         {
468.            Assert.AreEqual(29, MathUtility.Add(10,2,17));
```

```
469.          }
470.          [Test]
471.          public void MathUtilityWrapsOnOverflow()
472.          {
473.             Assert.AreEqual(-2, MathUtility.Add(int.MaxValue, int.MaxValue));
474.          }
475.       }
```

You wouldn't know it from the above example, but the Assert class has more than one function, such as Assert.IsFalse, Assert.IsTrue, Assert.IsNull, Assert.IsNotNull, Assert.AreSame, Assert.AreNotEqual, Assert.Greater, Assert.IsInstanceOfType and so on.

What is a Unit Test

Unit tests are methods that test behavior at a very granular level. Developers new to unit testing often let the scope of their tests grow. Most unit tests follow the same pattern: execute some code from your system and assert that it worked as expected. The goal of a unit test is to validate a specific behavior. You might have noticed that the above two examples use multiple unit tests on the same function. We don't want to write an all encompassing test for Save, but rather want to write a test for each of the behavior it contains – failing when the object is in an invalid state, calling our data access's Save method and setting the id, and calling the data access's Update method. It's important that our unit test pinpoint a failure as best possible.

I'm sure some of you will find the 4 tests used to cover the MathUtility.Add method a little excessive. You may think that all 4 tests could be grouped into the same one – and in this minor case I'd say whatever you prefer. However, when I started unit testing, I fell into the bad habit of letting the scope of my unit tests grow. I'd have my test which created an object, executed some of its members and asserted the functionality. But I'd always end up saying, *well as long as I'm here, I might as well throw in a couple extra asserts to make sure these fields are set the way they ought to be*. This is very dangerous because a change in your code could break numerous unrelated tests – definitely a sign that you've given your tests too little focus.

This brings us back to the topic about testing private methods. If you google you'll find a number of discussions on the topic, but the general consensus seems to be that you shouldn't test private methods. I think the most compelling reason not to test private methods is that our goal is not to test methods or lines of code, but rather to test behavior. This is something you must always remember. If you thoroughly test your code's public interface, then private methods should automatically get tested. Another argument against testing private methods is that it breaks encapsulation. We talked about the importance of information hiding already. Private methods contain implementation detail that we want to be able to change without breaking calling code. If we test private methods directly, implementation changes will likely break our tests, which doesn't bode well for higher maintainability. Here's a question for you: should we care that a change to a private method broke a test?

...

...

Hopefully you're starting to get my drift, the answer is NO. What we care about is whether the behavior/functionality is broken, which our tests against the public API will do.

Mocking

To get started, it's a good idea to test simple pieces of functionality. Before long though, you'll want to test a method that has a dependency on an outside component – such as the database. For example, you might want to complete your test coverage of the Car class by testing the Save method. Since we want to keep our tests as granular as possible (and as light as possible – tests should be quick to run so we can execute them often and get instant feedback) we really don't want to figure out how we'll set up a test database with fake data and make sure it's kept in a predictable state from test to test. In keeping with this spirit, all we want to do is make sure that Save interacts property with the DAL. Later on we can unit test the DAL on its own. If Save works as expected and the DAL works as expected and they interact properly with each other, we have a good base to move to more traditional testing.

In the previous part we saw the beginnings of testing with mocks. We were using a manually created mock class which had some pretty major limitations. The most significant of which was our inability to confirm that calls to our mock objects were occurring as expected. That, along with ease of use, is exactly the problem RhinoMock is meant to solve. Using RhinoMock couldn't be simpler, tell it what you want to mock (an interface or a class – preferably an interface), tell it what method(s) you expect to be called, along with the parameters, execute the call, and have it verify that your expectations were met.

Before we can get started, we need to give RhinoMock access to our internal types. This is quickly achieved by adding [assembly: InternalsVisibleTo("DynamicProxyGenAssembly2")] to our Properties/AssemblyInfo.cs file.

Now we can start coding by writing a test to cover the update path of our Save method:

```
476.       using NUnit.Framework;
477.
478.       [TestFixture]
479.       public class CarTest
480.       {
481.         [Test]
482.         public void SaveCarCallsUpdateWhenAlreadyExistingCar()
483.         {
484.            MockRepository mocks  = new MockRepository();
485.            IDataAccess dataAccess = mocks.CreateMock<IDataAccess>();
486.            ObjectFactory.InjectStub(typeof(IDataAccess), dataAccess);
```

```
487.
488.            Car car = new Car();
489.            dataAccess.Update(car);
490.            mocks.ReplayAll();
491.
492.            car.Id = 32;
493.            car.Save();
494.
495.            mocks.VerifyAll();
496.            ObjectFactory.ResetDefaults();
497.          }
498.        }
```

Once a mock object is created, which took 1 line of code to do, we inject it into our dependency injection framework (StructureMap in this case). When a mock is created, it enters record-mode, which means any subsequent operations against it, such as the call to dataAccess.UpdateCar(car), is recorded by RhinoMock. We exit record-mode by calling ReplayAll, which means we are now ready to execute our real code and have it verified against the recorded sequence. When we then call VerifyAll after having called Save on our Car object, RhinoMock will make sure that our actual call behaved the same as what we expected. In other words, you can think of everything before ReplayAll as stating our expectations, everything after it as our actual test code with VerifyAll doing the final check.

If we were to change the test to something along the lines of (notice the extra dataAccess.Update call):

```
499.        using NUnit.Framework;
500.
501.        [TestFixture]
502.        public class CarTest
503.        {
504.          [Test]
505.          public void SaveCarCallsUpdateWhenAlreadyExistingCar()
506.          {
507.            MockRepository mocks  = new MockRepository();
508.            IDataAccess dataAccess = mocks.CreateMock<IDataAccess>();
509.            ObjectFactory.InjectStub(typeof(IDataAccess), dataAccess);
510.
511.            Car car = new Car();
512.            dataAccess.Update(car);
513.            dataAccess.Update(car);
514.            mocks.ReplayAll();
515.
516.            car.Id = 32;
517.            car.Save();
518.
519.            mocks.VerifyAll();
520.            ObjectFactory.ResetDefaults();
521.          }
522.        }
```

RhinoMock would cause our test to fail and tell us that it didn't expect two calls to update.

For the save behavior, the interaction is slightly more complex – we have to make sure the return value is properly handled by the Save method. Here's the test:

```
523.       using NUnit.Framework;
524.
525.       [TestFixture]
526.       public class CarTest
527.       {
528.         [Test]
529.         public void SaveCarCallsSaveWhenNew()
530.         {
531.            MockRepository mocks  = new MockRepository();
532.            IDataAccess dataAccess = mocks.CreateMock<IDataAccess>();
533.            ObjectFactory.InjectStub(typeof(IDataAccess), dataAccess);
534.
535.            Car car = new Car();
536.            Expect.Call(dataAccess.Save(car)).Return(389);
537.            mocks.ReplayAll();
538.
539.            car.Save();
540.
541.            mocks.VerifyAll();
542.            Assert.AreEqual(389, car.Id);
543.            ObjectFactory.ResetDefaults();
544.         }
545.       }
```

Using the Expect.Call method allows us to specify the return value we want. Also notice the Assert.Equals we've added – which is the last step in validating the interaction. Hopefully the possibilities of having control over return values (as well as output/ref values) lets you see how easy it is to test for edge cases. Imagine that we changed our Save function to throw an exception if the returned id was invalid, our test would look like:

```
546.       using NUnit.Framework;
547.
548.       [TestFixture]
549.       public class CarTest
550.       {
551.         private MockRepository _mocks;
552.         private IDataAccess _dataAccess;
553.
554.         [SetUp]
555.         public void SetUp()
556.         {
557.            _mocks = new MockRepository();
558.             dataAccess =  mocks.CreateMock<IDataAccess>();
559.            ObjectFactory.InjectStub(typeof(IDataAccess), _dataAccess);
560.         }
561.         [TearDown]
562.         public void TearDown()
563.         {
564.             mocks.VerifyAll();
565.         }
566.
567.         [Test, ExpectedException("CodeBetter.Foundations.PersistenceException")]

568.         public void SaveCarCallsSaveWhenNew()
569.         {
570.            Car car = new Car();
571.            using ( mocks.Record())
572.            {
573.               Expect.Call(_dataAccess.Save(car)).Return(0);
574.            }
575.            using ( mocks.Playback())
576.            {
577.               car.Save();
578.            }
579.         }
580.       }
```

We've actually changed a lot. First, we've shown how nUnit ExpectedException attribute can be used to test for an exception. Secondly, we've extracted the repetitive code that creates, sets up and verifies the mock object into the SetUp and TearDown methods. Finally, we used a different, more explicit, RhinoMocks syntax for setting up our record and playback states - I generally prefer this syntax.

More on nUnit and RhinoMocks

So far we've only looked at the basic features offered by nUnit and RhinoMocks, but there's a lot more than can actually be done with them. For example, RhinoMocks can be setup to ignore the order of method calls, instantiate multiple mocks but only replay/verify specific ones, or mock some but not other methods of a class (a partial mock), or simply create a stub.

Combined with a utility like NCover (http://www.ncover.com/), you can also get reports on your tests coverage. Coverage basically tells you what percentage of an assembly/namespace/class/method was executed by your tests. NCover has a visual code browser that'll highlight any un-executed lines of code in red. Generally speaking, I dislike coverage as a means of measuring the completeness of unit tests. After all, just because you've executed a line of code does not mean you've actually tested it. What I do like NCover for is to highlight any code that has no coverage. In other words, just because of line of code or method has been executed by a test, doesn't mean you're test is good. But if a line of code or method hasn't been executed, then you need to look at adding some tests.

We've mentioned Test Driven Development briefly throughout this series. As has already been mentioned, Test Driven Development, or TDD, is about design, not testing. TDD means that you write your test first and then write corresponding code to make your test pass. In TDD we'd write our Save test before having any functionality in the Save method. Of course, our test would fail. We'd then write the specific behavior and test again. The general mantra for developers is red ? green ? refactor. Meaning the first step is to get a failing unit testing, then to make it pass, then to refactor the code as required.

In my experience, TDD goes very well with Domain Driven Design, because it really lets us focus on the business rules of the system. If our client says tracking dependencies between upgrades has been a major pain-point for them, then we set off right away with writing tests that'll define the behavior and API of that specific feature. I recommend that you familiarize yourself with unit testing in general before adopting TDD.

UI and Database Testing

Unit testing your ASP.NET pages probably isn't worth the effort. The ASP.NET

framework is complicated and suffers from very tight coupling. More often than not you'll require an actual HTTPContext, which requires quite a bit of work to setup. If you're making heavy use of custom HttpHandlers, you should be able to test those with quite a bit of ease.

On the other hand, testing your Data Access Layer is possible and I would recommend it. There may be better methods, but my approach has been to maintain all my CREATE Tables / CREATE Sprocs in text files along with my project, create a test database on the fly, and to use the Setup and Teardown methods to keep the database in a known state. The topic might be worth of a future blog post, but for now, I'll leave it up to your creativity.

Conclusion

Unit testing wasn't nearly as difficult as I first thought it was going to be. Sure my initial tests weren't the best – sometimes I would write near-meaningless test (like testing that a plain-old property was working as it should) and sometimes they were far too complex and well outside of a well-defined scope. But after my first project, I learnt a lot about what did and didn't work. One thing that immediately became clear was how much cleaner my code became. I quickly came to realize that if something was hard to test and I rewrote it to make it more testable, the entire code became more readable, better decoupled and overall easier to work with. The best advice I can give is to start small, experiment with a variety of techniques, don't be afraid to fail and learn from your mistakes. And of course, don't wait until your project is complete to unit test – write them as you go!

Part 6 - NHibernate

In part 3 we took our first stab at bridging the data and object world by hand-writing our own data access layer and mapper. The approach turned out to be rather limited and required quite a bit of repetitive code (although it was useful in demonstrating the basics). Adding more object and more functionality would bloat our DAL into an enormously unmaintainable violation of DRY (don't repeat yourself). In this section we'll look at an actual O/R Mapping framework to do all the heavy lifting for us. Specifically, we'll look at the popular open-source NHibernate framework (http://www.hibernate.org/343.html).

The single greatest barrier preventing people from adopting domain driven design is the issue of persistence. My own adoption of O/R mappers came with great trepidation and doubt. You'll essentially be asked to trade in your knowledge of a tried and true method for something that seems a little too magical. A leap of faith may be required.

The first thing to come to terms with is that O/R mappers generate your SQL for you. I know, it sounds like it's going to be slow, insecure and inflexible, especially

since you probably figured that it'll have to use inline SQL. But if you can push those fears out of your mind for a second, you have to admit that it could save you a lot of time and result in a lot less bugs. Remember, we want to focus on building behavior, not worry about plumbing (and if it makes you feel any better, a good O/R mapper will provide simple ways for you to circumvent the automated code generation and execute your own SQL or stored procedures).

Infamous Inline SQL vs Stored Procedure Debate

Over the years, there's been some debate between inline SQL and stored procedures. This debate has been very poorly worded, because when people hear inline SQL, they think of badly written code like:

```
581.        public int GetUserIdByCredentials(string userName, string password)
582.        {
583.           string sql = @"SELECT UserId FROM Users
584.                          WHERE UserName = '" + userName + "' AND Password = '" +
    password + "'";
585.           using (SqlCommand command = new SqlCommand(sql))
586.           {
587.              //todo
588.              return 0;
589.           }
590.        }
```

If you stop and think about it though, and compares apples to apples, I think you'll agree that neither is particularly better than the other. Let me help you out.

Stored Procedures are more Secure

Inline SQL should be written using parameterized queries just like you do with stored procedures. For example, the correct way to write the above code in order to eliminate the possibility of an SQL injection attack is:

```
591.        public int GetUserIdByCredentials(string userName, string password)
592.        {
593.           string sql = @"SELECT UserId FROM Users
594.                          WHERE UserName = @UserName AND Password = @Password";
595.           using (SqlCommand command = new SqlCommand(sql))
596.           {
597.              command.Parameters.Add("@UserName", SqlDbType.VarChar, 64).Value = u
    serName;
598.              command.Parameters.Add("@Password", SqlDbType.VarChar, 64).Value = p
    assword;
599.              //todo
600.              return 0;
601.           }
602.        }
```

Stored procedures provide an abstraction to the underlying schema

Whether you're using inline SQL or stored procedures, what little abstraction you can put in a SELECT statement is the same. If any substantial changes are made, your stored procedures are going to break and there's a good chance you'll need to change the calling code to deal with the issue. O/R Mappers on the other side, generally provide much better abstraction.

If I make a change, I don't have to recompile the code
Somewhere, somehow, people got it in their head that code compilations should be avoided at all cost (maybe this comes from the days where projects could take hours to compile). If you change a stored procedure, you still have to re-run your unit and integration tests and deploy a change to production. It genuinely scares and puzzles me that developers consider a change to a stored procedure or XML trivial compared to a similar change in code.

Stored Procedures reduce network traffic
Who cares? In most cases your database is sitting on a GigE connection with your servers and you aren't paying for that bandwidth. You're literally talking fractions of milliseconds. On top of that, a well configured O/R mapper can save round-trips via identify map implementations, caching and lazy loading.

Stored procedures are faster
This is the excuse I held onto the longest. Write a reasonable/common SQL statement inline and then write the same thing in a stored procedure and time them. Go ahead. In most cases there's little or no difference. In some cases, stored procedures will be slower because a cached execution plan will not be efficient given a certain parameter. Jeff Atwood called using stored procedures for the sake of better performance a fairly extreme case of premature optimization. He's right. The proper approach is to take the simplest possible approach (let a tool generate your SQL for you), and optimize specific queries when/if bottlenecks are identified.

It took a while, but after a couple years, I realized that the debate between inline and stored procedures was as trivial and meaningless as the one about C# and VB.NET. Of course, since the differences are practically non-existing; why not just use stored procedures? If you aren't willing to adopt an O/R mapper, that's certainly what I would suggest – there's no sense in dynamically creating your own inline SQL. However, O/R mappers, which rely on inline SQL, provide three very important benefits (there are more, but with respect to maintainability, I think these are the most important:

1. You end up writing a lot less code – which obviously results in a more maintainable system,
2. You gain a true level of abstraction from the underlying data source – both because you're querying the O/R mapper for your data directly (and it converts that into the appropriate SQL), and because you're providing mapping information between your table schemas and domain objects,
3. If your impedance mismatch is low, they save you from having to write a lot of repetitive code; however, if your impedance mismatch is high, you'll be able to design your database the way it should be, and your domain layer the way it should be, without having to create an uncomfortable compromise – the O/R mapper will handle the mismatch for you.

In the end, this really comes down to building the simplest solution upfront. After a few iterations, you can spend time profiling your code, and only if you detect an actual problem do you have to address that specific case. It might not sound so much simpler because you have to learn a fairly complex framework upfront, but that's the reality of our profession.

Remember, our goal is to widen our knowledge base by looking at different ways to build systems in order to provide our clients with greater value. While we may be specifically talking about NHibernate, the goal is really to introduce to concept of O/R mappers, and try to correct the blind faith .NET developers have put into stored procedures and ADO.NET.

NHibernate

Of the frameworks and tools we've looked at so far, NHibernate is the most complex. This complexity is certainly something you should take into account when deciding on a persistence solution, but once you do find a project that allows for some R&D time, the payoff will be well worth it in future projects. The nicest thing about NHibernate, and a major design goal of the framework, is that it's completely transparent – your domain objects aren't forced to inherit a specific base class and you don't have to use a bunch of decorator attributes. This makes unit testing your domain layer possible – if you're using a different persistent mechanism, say typed datasets, the tight coupling between domain and data makes it hard/impossible to properly unit test.

At a very high level, you configure NHibernate by telling it how your database (tables and columns) map to your domain objects, use the NHibernate API and NHibernate Query Language to talk to your database, and let it do the low level ADO.NET and SQL work.

In previous parts we focused on a system for a car dealership – specifically focusing on cars and upgrades. In this part we'll change perspective slightly and look at car sales (sales, models and sales people).The domain model is simple – a SalesPerson has zero or more Sales which reference a specific Model.

I've also included a VS.NET solution that contains sample code and annotations – you can find a link at the end of this article. All you need to do to get it running is create a new database, execute the provide SQL script (a handful of create tables), and configure the connection string. The sample, along with the rest of this article, is meant to help you get started with NHibernate – a topic too often overlooked.

You might also be interested in the excellent NHibernate Reference Manual as well as Manning's NHibernate in Action book.

Configuration

The secret to NHibernate's amazing flexibility lies in its configurability. Initially it

can be rather daunting to set it up, but after a coupe project it becomes rather natural. The first step is to configure the NHibernate itself. The simplest such configuration, which must be added to your app.config or web.config, looks like:

```
603.       <?xml version="1.0" encoding="utf-8" ?>
604.       <configuration>
605.         <configSections>
606.           <section name="hibernate-
    configuration" type="NHibernate.Cfg.ConfigurationSectionHandler, NHibernate" />
607.         </configSections>
608.         <hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
609.           <session-factory>
610.             <property name="hibernate.dialect">NHibernate.Dialect.MsSql2005Diale
    ct</property>
611.             <property name="hibernate.connection.provider">NHibernate.Connection
    .DriverConnectionProvider</property>
612.             <property name="hibernate.connection.connection string">Server=SERVE
    R;Initial Catalog=DATABASE;User Id=USER;Password=PASSWORD;</property>
613.             <mapping assembly="CodeBetter.Foundations" />
614.           </session-factory>
615.         </hibernate-configuration>
616.       </configuration>
```

Of the four values, dialect is the most interesting. This tells NHibernate what specific language our database speaks. If, later on, we ask NHibernate to return a paged result of Cars and our dialect is set to SQL Server 2005, NHibernate will issue an SQL SELECT utilizing the ROW_NUMBER() ranking function. However, if the dialect is set to MySQL, NHibernate will issue a SELECT with a LIMIT. In most cases, you'll set this once and forget about it, but it does provide some insight into the capabilities provide by a layer that generates all of your data access code.

In our configuration, we also told NHibernate that our mapping files were located in the CodeBeter.Foundations assembly. Mapping files are embedded XML files which tell NHibernate how each class is persisted. With this information, NHibernate is capable of returning a Car object when you ask for one, as well as saving it. The general convention is to have a mapping file per domain object, and for them to be placed inside a Mappings folder. The mapping file for our Model object, name Model.hbm.xml, looks like:

```
617.       <hibernate-mapping xmlns="urn:nhibernate-mapping-
    2.2" assembly="CodeBetter.Foundations" namespace="CodeBetter.Foundations">
618.         <class name="Model" table="Models" lazy="true" proxy="Model">
619.           <id name="Id" column="Id" type="int" access="field.lowercase-
    underscore">
620.             <generator class="native" />
621.           </id>
622.           <property name="Name" column="Name" type="string" not-
    null="true" length="64" />
623.           <property name="Description" column="Description" type="string" not-
    null="true" />
624.           <property name="Price" column="Price" type="double" not-
    null="true" />
625.         </class>
626.       </hibernate-mapping>
```

(it's important to make sure the "Build Action" for all mapping files is set to "Embedded Resources")

This file tells NHibernate that the Model class maps to rows in the Models table, and that the 4 properties Id, Name, Description and Price map to the Id, Name, Description and Price columns. The extra information around the Id property specifies that the value is generated by the database (as opposed to NHibernate itself (for clustered solutions), or our own algorithm) and that there's no setter, so it should be accessed by the field with the specified naming convention (we supplied Id as the name, and lowercase-underscore as the naming strategy, so it'll use a field named _id).

With the mapping file set up, we can start interacting with the database:

```
627.        //Let's add a new car model
628.        Model model = new Model();
629.        model.Name = "Hummbee";
630.        model.Description = "Great handling, built-
    in GPS to always find your way back home, Hummbee2Hummbe(tm) communication";
631.        model.Price = 50000.00;
632.        ISession session = _sessionFactory.OpenSession();
633.        session.Save(model);
634.
635.
636.        //Let's discount the x149
637.        Model model = session.CreateQuery("from Model model where model.Name = ?")
    .SetString(0, "X149").UniqueResult<Model>();
638.        model.Price -= 5000;
639.        ISession session =  sessionFactory.OpenSession();
640.        session.Update(model);
```

The above example shows how easy it is to persist new objects to the database, retrieve them and update them – all without any ADO.NET or SQL.

You may be wondering where the _sessionFactory object comes from, and exactly what an ISession is. The _sessionFactory (of type ISessionFactory) is a global thread-safe object that you'd likely create on application start. You'll typically need one per database your application is using (which means you'll typically only need one), and its job, like most factories, is to create a preconfigured object: an ISession. The ISession has no ADO.NET equivalent, but it does map loosely to a database connection. However, creating an ISession doesn't necessarily open up a connection. Instead, ISessions smartly manage connections and command objects for you. Unlike connections which should be opened late and closed early, you needn't worry about having ISessions stick around for a while (although they aren't thread-safe). If you're building an ASP.NET application, you could safely open an ISession on BeginRequest and close it on EndRequest (or better yet, lazy-load it in case the specific request doesn't require an ISession).

ITransaction is another piece of the puzzle which is created by calling BeginTransaction on an ISession. It's common for .NET developers to ignore the need for transactions within their applications. This is unfortunate because it can lead to unstable and even unrecoverable states in the data. An ITransaction is used to keep track of the unit of work – tracking what's changed, been added or

deleted, figuring out what and how to commit to the database, and providing the capability to rollback should an individual step fail.

Relationships

In our system, it's important that we track sales – specifically with respect to sales people, so that we can provide some basic reports. We're told that a sale can only ever belong to a single sales person, and thus set up a one to many relationship – that is, a sales person can have multiple sales, and a sales can only belong to a single sales person. In our database, this relationship is represented as a SalesPersonId column in the Sales table (a foreign key). In our domain, the SalesPerson class has a Sales collection and the Sales class has a SalesPerson property (references).

Both ends of the relationship needs to be setup in the appropriate mapping file. On the Sales end, which maps a single property, we use a glorified property element called many-to-one:

```
641.       ...
642.       <many-to-
    one name="SalesPerson" class="SalesPerson" column="SalesPersonId" not-
    null="true"/>
643.       ...
```

We're specifying the name of the property, the type/class, and the foreign key column name. We're also specifying an extra constraint, that is, when we add a new Sales object, the SalesPerson property can't be null.

The other side of the relationship, the collection of sales a sales person has, is slightly more complicated – namely because NHibernate's terminology isn't standard .NET lingo. To set up a collection we use a set, list, map, bag or array element. Your first inclination might be to use list, but NHibernate requires that you have a column that specifies the index. In other words, the NHibernate team sees a list as a collection where the index is important, and thus must be specified. What most .NET developers think of as a list, NHibernate calls a bag. Confusingly, whether you use a list or a bag element, your domain type must be an IList (or its generic IList equivalent). This is because .NET doesn't have an IBag object. In short, for your every day collection, you use the bag element and make your property type an IList.

The other interesting collection option is the set. A set is a collection that cannot contain duplicates – a common scenario for enterprise application (although it is rarely explicitly stated). Oddly, .NET doesn't have a set collection, so NHibernate uses the Iesi.Collection.ISet interface. There are four specific implementations, the ListSet which is really fast for very small collections (10 or less items), the SortedSet which can be sorted, the HashedSet which is fast for larger collections and the HybridSet which initially uses a ListSet and automatically switches itself to a HashedSet as your collection grows.

For our system we'll use a bag (even though we can't have duplicate sales, it's just a little more straightforward right now), so we declare our Sales collection as an IList:

```
644.       private IList<Sale> _sales;
645.       public IList<Sale> Sales
646.       {
647.          get { return _sales;}
648.       }
```

And add our element to the SalesPerson mapping file:

```
649.       ...
650.       <bag name="Sales" access="field.lowercase-
    underscore" table="Sales" inverse="true" cascade="all">
651.          <key column="SalesPersonId" />
652.          <one-to-many class="Sale" />
653.       </bag>
654.       ...
```

Again, if you look at each element/attribute, it isn't as complicated as it first might seem. We identify the name of our property, specify the access strategy (we don' t have a setter, so tell it to use the field with our naming convention), the table and column holding the foreign key, and the type/class of the items in the collection.

We've also set the cascade attribute to all which means that when we call Update on a sales person, any changes made to his or her sales collection (additions, removals, changes to existing sales) will automatically be persisted. Cascading can be a real time saver as your system grows in complexity.

Querying

NHibernate supports two different querying approaches: Hibernate Query Language (HQL) and Criteria Queries (you can also query in actual SQL, but lose portability when doing so). HQL is the easier of two as it looks a lot like SQL – you use from, where, aggregates, order by, group by, etc. However, rather than querying against your tables, you write queries against your domain – which means HQL supports OO principles like inheritance and polymorphism. Either query methods are abstractions on top of SQL, which means you get total portability – all you need to do to target a different database is change your dialect configuration.

HQL works off of the IQuery interface, which is created by calling CreateQuery on your session. With the IQuery you can return individual entities, collections, substitute parameters and more. Here are some example:

```
655.       string lastName = "allen";
656.       ISession session = _sessionFactory.OpenSession();
657.
658.       //retrieve a salesperson by last name
659.       IQuery query = session.CreateQuery("from SalesPerson p where p.LastName =
    'allen'");
660.       SalesPerson p = query.UniqueResult<SalesPerson>();
```

```
661.
662.       //same as above but in 1 line, and with the last name as a variable
663.       SalesPerson p = session.CreateQuery("from SalesPerson p where p.LastName =
     ?").SetString(0, lastName).UniqueResult<SalesPerson>();
664.
665.       //people with few sales
666.       IList<SalesPerson> slackers = session.CreateQuery("from SalesPerson person
     where size(person.Sales) < 5").List<SalesPerson>();
```

Lazy Loading

When we load a sales person, say by doing: *SalesPerson person = session.Get(1);* the Sales collection won't be loaded. That's because, by default, collections are lazily loaded. That is, we won't hit the database until the information is specifically requested (i.e., we access the Sales property). We can override the behavior by setting lazy="false" on the bag element.

The other, more interesting, lazy load strategy implemented by NHibernate is on entities themselves. You'll often want to add a reference to an object without having to load the actual object from the database. For example, when we add a sales to a sales person, we need to specify the model, but don't want to load all the model information – all we really want to do is get the Id so we can store it in the ModelId column of the Sales table. When you use session.Load(id) NHibernate will load a proxy of the actual object (unless you specify lazy="false" in the class element). As far as you're concerned, the proxy behaves exactly like the actual object, but none of the data will be retrieved from the database until the first time you ask for it. This makes it possible to write the following code:

```
667.       Sale sale = new Sale(session.Load<Model>(1), DateTime.Now, 46000.00);
668.       salesPerson.AddSales(sale);
669.       session.SaveOrUpdate(salesPerson);
```

without ever having to actually hit the database to load the model.

Download

I've included a download which'll hopefully provide a base for you to start playing with NHibernate. The code is well documented - take special care to read the annotations withint he mapping files. To get it running:

1. Create a new database and run the CREATE TABLE commands located in CREATE_TABLES.sql,
2. Modify the hibernate.connection.connection_string property within the app.config so that it can connect to your newly created database

Once configured, take a look at the Run method within Sample.cs and walk through each call one at a time.

Download Project
(http://codebetter.com/files/folders/codebetter_downloads/entry172562.aspx)

Conclusion

We've only touched the tip of what you can do with NHibernate. We haven't looked at its Criteria Queries (which is a query API tied even closer to your domain than HQL), its caching capabilities, filtering of collections, performance optimizations, logging, or native SQL abilities. Beyond NHibernate the tool, hopefully you've learnt more about object relational mapping, and alternative solutions to the limited toolset baked into .NET. It is hard to let go of hand written SQL statement, but looking beyond the bias of what's comfortable, it's impossible to rationalize doing all that work upfront.

Part 7 - ActiveRecord

You may be wondering what happened to part 6. Well, it's still being worked on and should be available early next week.

I wasn't sure if there would be a part 7 and if so what it would be about - but I was heavily considering writing about an ActiveRecord implementation. Turns out that Kent Sharkey, beat me to it with his overview of Subsonic on DotNetSlackers.

I want to spend just a couple paragraphs trying to tie in Kent's article with the Foundations series.

In Part 3 we talked about persistence and manually wrote our data access layer and mapping code. As I said at the time, doing that manually is fine for simple cases with few domain objects and straightforward mapping, but can quickly get out of hand. To solve the problem and keep us efficient, we can leverage existing O/R mappers to do all the mundane work. Some O/R mappers, like NHibernate which we'll look at in Part 6, are unbelievably flexible and can be used to address very large systems (as systems grow in size, it's common for their impedance mismatch to grow as well, which makes a flexible tool all the more important). Of course, NHibernate isn't the simplest thing to configure and it has a relatively steep learning curve.

An alternative approach is to use the ActiveRecord (AR) design pattern. AR is still considered an O/R mapper, but it's specifically targeted at more straightforward applications (which likely represent 95% of the work we're all doing). AR is one of the two components that has made Ruby on Rails so popular and productive (the other is their first class MCV pattern implementation). There are two popular .NET implementations that I'm aware of: Subsonic and Castle ActiveRecord. Both the Castle and Subsonic line of products are open source, and come with a full range of products (MVC, ActiveRecord, Scaffolding, utility functions, etc) that integrate nicely with each other. Interestingly, Castle is implemented on top of NHibernate - which gives you a hit about just how flexible NHibernate really is.

With Castle ActiveRecord, you take your domain object and decorate it with attributes. You add the ActiveRecordAttribute to your class, and the PropertyAttribute to your properties. So, if you have a Car class with a Name property which are properly set up, Castle will automatically map to a table named Car and a column named Name.

It turns out that ActiveRecord implementations can be setup within minutes and handle 99% of all your data access needs. They can make you unbelievably productive by providing functionality and performance above that of DataSets, while maintaining your rich domain layer and allow you to easily write unit tests.

It's still worthwhile to learn a more complex framework like NHibernate for a couple reasons. First, once you're used to it, it too can be set up quickly and efficiently (avoiding upfront learning is a losing strategy). Secondly, NHibernate is a lot closer to important fundamental concepts. With AR implementations, a lot of the details are taken care of for you - which is great in most cases, but doesn't teach you nearly as much. If you spend some time with NHibernate, you'll learn about first and second level caching, identity map, concurrency and locking and various other concepts relating to persistence.

Enjoy Kent's article (http://dotnetslackers.com/articles/aspnet/IntroductionToSubSonic.aspx), and I hope take the time to download and play with either Subsonic or Castle.

Happy holidays to all our CB readers and your families.