Foundations of Programming

Building Better Software

By Karl Seguin

WWW.CODEBETTER.COM

This page was intentionally left blank

License

Downloadable Learning Application

Reading about code is a good way to learn, but if you're anything like me, nothing beats a real application. That's why I created the Canvas Learning Application - a simple (yet complete) ASP.NET MVC website which leverages many of the ideas and tools covered in this book. The application is a Visual Studio 2008 solution with helpful inline documentation meant to help you bridge the gap between theory and practice. The Canvas Application and this book are pretty independent, so you can approach this journey however you prefer.

http://codebetter.com/blogs/karlseguin/archive/2009/05/25/revisiting-codebetter-canvas.aspx

Acknowledgement

There are countless of people who deserve thanks. This book is but a small contribution to the incalculable time donated and knowledge shared by the software community at large. Without the quality books, forums, newsgroup posts, blogs, libraries and open source projects, I would still be trying to figure out why my ASP script was timing-out while looping through a recordset (stupid MoveNext).

It's no surprise that the software community has leveraged the openness of the internet more than any other profession in order to advance our cause. What is surprising is how the phenomenon appears to have gone unnoticed. Good!

Of course, there is one special person without whom this couldn't have happened.

To Wendy,

People call me lucky for being with someone as beautiful and intelligent as you. They don't know the half of it. You are not only beautiful and intelligent, but you let me spend far too much time on my computer, either working, learning, writing or playing. You're also more than happy to read over my stuff or listen to me blab on about nonsense. I don't appreciate you nearly as much as I should.

Table of Contents

About the Author

Karl Seguin is a developer at Epocal Corporation, a former Microsoft MVP, a member of the influential CodeBetter.com community and an editor for DotNetSlackers. He has written numerous articles and is an active member of various Microsoft public newsgroups. He lives in Ottawa, Ontario Canada.

His personal webpage is: http://www.openmymind.net/

His blog, along with that of a number of distinguished professionals, is located at: http://www.codebetter.com/

ALT.NET

1

> IF THERE IS DISSATISFACTION WITH THE STATUS QUO, GOOD. IF THERE IS FERMENT, SO MUCH THE BETTER. IF THERE IS RESTLESSNESS, I AM PLEASED. THEN LET THERE BE IDEAS, AND HARD THOUGHT, AND HARD WORK. IF MAN FEELS SMALL, LET MAN MAKE HIMSELF BIGGER. – HUBERT H HUMPHREY

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 dominant 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, despite what many people think, ALT.NET isn't about *ALT*ernatives 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.

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 firsthand 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 chapter, 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 chapter, 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 chapter.

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.

In This Chapter

Even though this chapter didn't have any 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 in. Hopefully we've already managed to clear up some of the buzz words you've been hearing so much about lately. The next couple chapters 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.

Domain Driven Design

2

> WHAT IS DESIGN? IT'S WHERE YOU STAND WITH A FOOT IN TWO WORLDS - THE WORLD OF TECHNOLOGY AND THE WORLD OF PEOPLE AND HUMAN PURPOSES - AND YOU TRY TO BRING THE TWO TOGETHER. – MITCHELL KAPOR

It's predictable to start off by talking about domain driven design and object oriented programming. 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.

Domain/Data Driven Design

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 afterthought. 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 chapters. 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 having to always catch-up.)

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.

> *In the past, I'd frequently get pissed off with clients. They were annoying, didn't know what they wanted and always made the wrong choice.*
>
> *Once I actually thought about it though, I realized that I wasn't nearly as smart as I thought I was. The client knew far more about his or her business than I did. Not only that, but it was their money and my job was to make them get the most out of it.*
>
> *As the relationship turned more collaborative and positive, not only did the end result greatly improve, but programming became fun again.*

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 else's. 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[1].

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 chapter 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 chapter we'll cover the basics of persistence, and in following chapters, 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 `StrategicOutcomes` 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

[1] Amazon Link: http://www.amazon.com/User-Stories-Applied-Development-Addison-Wesley/dp/0321205685

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:

```
public class Car{}
public class Model{}
public class Package{}
public class Upgrade{}
```

Next we'll add a bit of code based on some pretty safe assumptions:

```
public class Car
{
   private Model _model;
   private List<Upgrade> _upgrades;

   public void Add(Upgrade upgrade){ //todo }
}
public class Model
{
   private int _id;
   private int _year;
   private string _name;

   public ReadOnlyCollection<Upgrade> GetAvailableUpgrades()
   {
      return null;  //todo
   }
}
public class Upgrade
{
   private int _id;
   private string _name;

   public ReadOnlyCollection<Upgrade> RequiredUpgrades
   {
      get { return null; //todo }
   }
}
```


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.

```
public class Car
{
   private Model _model;
   //todo where to initialize this?
   private List<Upgrade> _upgrades;

   public void Add(Upgrade upgrade)
   {
      _upgrades.Add(upgrade);
   }
   public ReadOnlyCollection<Upgrade> MissingUpgradeDependencies()
   {
      List<Upgrade> missingUpgrades = new List<Upgrade>();
      foreach (Upgrade upgrade in _upgrades)
      {
         foreach (Upgrade dependentUpgrade in upgrade.RequiredUpgrades)
         {
            if (!_upgrades.Contains(dependentUpgrade)
                && !missingUpgrades.Contains(dependentUpgrade))
            {
               missingUpgrades.Add(dependentUpgrade);
            }
         }
      }
      return missingUpgrades.AsReadOnly();
   }
}
```

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 intermix 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.

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 necessarily 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 chapter, but if you're interested to find out more, I suggest you look at MonoRails (which is a Rails framework for .NET) or the recently released MVC framework by Microsoft. 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 this chapter 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:

```
public class Car
 {
    private Model _model;
    private List<Upgrade> _upgrades;

    private Car()
    {
       _upgrades = new List<Upgrade>();
    }
    public static Car CreateCar(Model model)
    {
```

```
            Car car = new Car();
            car._model = model;
            return car;
        }
    }
```

There are two advantages to this approach. Firstly, 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 chance 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.

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.

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:

```
internal class SqlServerDataAccess
{
   internal List<Upgrade> RetrieveAllUpgrades()
   {
       return null;  //todo implement
   }
}
public void ASampleMethod()
{
      SqlServerDataAccess da = new SqlServerDataAccess();
      List<Upgrade> upgrades = da.RetrieveAllUpgrades();
}
```

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 `ASampleMethod` without having a fully functional `RetrieveAllUpgrades` method). We can relieve this tight coupling by programming against an interface instead:

```
internal interface IDataAccess
{
   List<Upgrade> RetrieveAllUpgrades();
}

internal class DataAccess
{
   internal static IDataAccess CreateInstance()
   {
      return new SqlServerDataAccess();
   }
}

internal class SqlServerDataAccess : IDataAccess
{
   public List<Upgrade> RetrieveAllUpgrades()
   {
      return null; //todo implement
   }
}
public void ASampleMethod()
{
   IDataAccess da = DataAccess.CreateInstance();
   List<Upgrade> upgrades = da.RetrieveAllUpgrades();
}
```

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 a 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 object's 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.

In This Chapter

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 real life things. 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 chapters and experiment on your own, you'll start adopting some of the concepts and tweaking them to fit yours and your clients needs.

Persistence

3

> CODD'S AIM WAS TO FREE PROGRAMMERS FROM HAVING TO KNOW THE PHYSICAL STRUCTURE OF DATA. OUR AIM IS TO FREE THEM IN ADDITION FROM HAVING TO KNOW ITS LOGICAL STRUCTURE. – LAZY SOFTWARE

In the previous chapter 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 chapter 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 will look at more advanced alternatives (two different O/R mapping approaches) which, like `DataSets`, do much of the heavy lifting for us. This chapter 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):


```
public class Upgrade
{
   private int _id;
   private string _name;
   private string _description;
   private decimal _price;
   private List<Upgrade> _requiredUpgrades;

   public int Id
   {
      get { return _id; }
      internal set { _id = value; }
   }
   public string Name
   {
      get { return _name; }
      set { _name = value; }
   }
   public string Description
   {
      get { return _description; }
      set { _description = value; }
   }
   public decimal Price
   {
      get { return _price; }
      set { _price = value; }
   }

   public List<Upgrade> RequiredUpgrades
   {
      get { return _requiredUpgrades; }
   }
}
```

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.

```
CREATE TABLE Upgrades
(
      Id INT IDENTITY(1,1) NOT NULL PRIMARY KEY,
      [Name] VARCHAR(64) NOT NULL,
      Description VARCHAR(512) NOT NULL,
      Price MONEY NOT NULL,
 )
```

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)


```
internal class SqlServerDataAccess
{
   private readonly static string _connectionString = "FROM_CONFIG"

   internal List<Upgrade> RetrieveAllUpgrades()
   {
      //use a sproc if you prefer
      string sql = "SELECT Id, Name, Description, Price FROM Upgrades";
      using (SqlCommand command = new SqlCommand(sql))
      using (SqlDataReader dataReader = ExecuteReader(command))
      {
         List<Upgrade> upgrades = new List<Upgrade>();
         while (dataReader.Read())
         {
            upgrades.Add(DataMapper.CreateUpgrade(dataReader));
         }
         return upgrades;
      }
   }


   private SqlDataReader ExecuteReader(SqlCommand command)
   {
      SqlConnection connection = new SqlConnection(_connectionString);
      command.Connection = connection;
      connection.Open();
      return command.ExecuteReader(CommandBehavior.CloseConnection)
   }
}
```

`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.

```
internal static class DataMapper
{
   internal static Upgrade CreateUpgrade(IDataReader dataReader)
   {
      Upgrade upgrade = new Upgrade();
      upgrade.Id = Convert.ToInt32(dataReader["Id"]);
      upgrade.Name = Convert.ToString(dataReader["Name"]);
      upgrade.Description = Convert.ToString(dataReader["Description"]);
      upgrade.Price = Convert.ToDecimal(dataReader["Price"]);
      return upgrade;
   }
}
```


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:

```
internal static SqlParameter[] ConvertUpgradeToParameters(Upgrade upgrade)
{
   SqlParameter[] parameters = new SqlParameter[4];
   parameters[0] = new SqlParameter("Id", SqlDbType.Int);
   parameters[0].Value = upgrade.Id;

   parameters[1] = new SqlParameter("Name", SqlDbType.VarChar, 64);
   parameters[1].Value = upgrade.Name;

   parameters[2] = new SqlParameter("Description", SqlDbType.VarChar, 512);
   parameters[2].Value = upgrade.Description;

   parameters[3] = new SqlParameter("Price", SqlDbType.Money);
   parameters[3].Value = upgrade.Price;
   return parameters;
}
```

We have a problem

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 a 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).

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

Next we modify `RetrieveAllUpgrade` to load-in required upgrades:

```
internal List<Upgrade> RetrieveAllUpgrades()
{
   string sql = @"SELECT Id, Name, Description, Price FROM Upgrades;
                  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);
         int requiredUpgradeId = dataReader.GetInt32(1);
         Upgrade upgrade;
         Upgrade required;
         if (!localCache.TryGetValue(upgradeId, out upgrade)
            || !localCache.TryGetValue(requiredUpgradeId, out required))
         {
            //probably should throw an exception
            //since our db is in a weird state
            continue;
         }
         upgrade.RequiredUpgrades.Add(requiredUpgrade);
      }
      return upgrades;
   }
}
```

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 to another.

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:

```
SqlServerDataAccess da = new SqlServerDataAccess();
Upgrade upgrade1a = da.RetrieveAllUpgrades()[0];
Upgrade upgrade1b = da.RetrieveAllUpgrades()[0];

upgrade1b.Price = 2000;
upgrade1b.Save();
```

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 Identify 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 googling 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 objects. 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).

In This Chapter

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 chapters – but when we do re-address it we'll examine full-blown solutions that pack quite a punch.

Dependency Injection

4

> I WOULD SAY THAT MODERN SOFTWARE ENGINEERING IS THE ONGOING REFINEMENT OF THE EVER-INCREASING DEGREES OF DECOUPLING. YET, WHILE THE HISTORY OF SOFTWARE SHOWS THAT COUPLING IS BAD, IT ALSO SUGGESTS THAT COUPLING IS UNAVOIDABLE. AN ABSOLUTELY DECOUPLED APPLICATION IS USELESS BECAUSE IT ADDS NO VALUE. DEVELOPERS CAN ONLY ADD VALUE BY COUPLING THINGS TOGETHER. THE VERY ACT OF WRITING CODE IS COUPLING ONE THING TO ANOTHER. THE REAL QUESTION IS HOW TO WISELY CHOOSE WHAT TO BE COUPLED TO. - JUVAL LÖWY

It's common to hear developers promote layering as a means to provide extensibility. The most common example, and one I used in Chapter 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 chapter 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 function in 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 chapter.

(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, provided 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 another, 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:

```
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");
      }
      if (_id == 0)
      {
         _id = DataAccess.CreateInstance().Save(this);
      }
      else
      {
         DataAccess.CreateInstance().Update(this);
      }
   }

   private bool IsValid()
   {
      //todo: make sure the object is in a valid state
       return true;
   }
}
```

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.

Don't avoid Coupling like the Plague

In case you forgot from Chapter 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 an `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 chapter, and once you play with unit testing yourself, you'll get a feel for what should and shouldn't be avoided.

> *Like Juval is quoted at the start of this chapter, I'm tempted to throw my vote in for decoupling as the single greatest necessity for modern applications.*
>
> *In fact, in addition to the oft-cited benefits of unit testing, I've found that the most immediate advantage is to help developers learn the patterns of good and bad coupling.*
>
> *Who hasn't looked back at code they wrote two or three years ago and been a little embarrassed? I can categorically say that the single greatest reason my code stunk was because of horrible coupling. I'm ready to bet your old code has the exact same stench!*

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

Dependency Injection

In Chapter 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()
   {
      if (!IsValid())
      {
         //todo: come up with a better exception
         throw new InvalidOperationException("The car must be in a valid
                                              state");
      }
      if (_id == 0)
      {
```

```
            _id = _dataProvider.Save(this);
        }
        else
        {
            _dataProvider.Update(this);
        }
    }
}
```

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:

```
public void AlmostATest()
{
    Car car = new Car(new MockDataAccess());
    car.Save();
    if (car.Id != 1)
    {
        //something went wrong
    }
}
```

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 chapter will focus on StructureMap, an open source 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.SqlDataAccess, 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);
      }
   }
}
```

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();
   if (car.Id != 1)
   {
      //something went wrong
   }
   ObjectFactory.ResetDefaults();
}
```

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 has increased our testability. In the past, I've introduced StructureMap into existing large codebases in a matter of minutes – the impact is minor.

A Final Improvement

With the introduction of the `IDataAccess` class as well as the use of our DI framework, we've managed to remove much of the bad coupling present in our simple example. We could probably take it a couple steps further, even to a point where it might do more harm than good. There is however one last dependency that I'd like to hide away - our business objects are probably better off not knowing about our specific DI implementation. Rather than calling StructureMap's `ObjectFactory` directly, we'll add one more level of indirection:

```
public static class DataFactory
{
   public static IDataAccess CreateInstance
   {
      get
      {
         return ObjectFactory.GetInstance<IDataAccess>();
      }
   }
}
```

Again, thanks to a simple change, we're able to make massive changes (choosing a different DI framework) well into the development of our application with ease.

In This Chapter

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 chapter we'll look at unit testing which will 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 located on DotNetSlackers.

Unit Testing

5

> We choose not to do Unit Testing since it gives us higher profit not to!
> - (Random Consultancy Company)

Throughout this book we've talked about the importance of testability and we've 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 testing 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, refactoring, you name it). Being that it's a fairly large system, we're sometimes asked to make a change that ought to 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.

I simply can't stress how important unit tests are. Sure they're helpful in finding bugs and validating that my code does what it should, but far more important is their seemingly magical ability to uncover fatal flaws, or priceless gems in the design of a system. I get excited whenever I run across a method or behavior which is mind-blowingly difficult to test. It means I've likely found a flaw in a fundamental part of the system which likely would have gone unnoticed until some unforeseen change was asked for. Similarly, whenever I put together a test in a couple seconds for something which I was pretty sure was going to be difficult, I know someone on the team wrote code that'll be reusable in other projects.

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 used by bad 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). 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 your client might not feel the upfront cost is justified. In these situations I suggest you identify 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 chapter, we need only add 2 frameworks and 1 tool to our unit testing framework: nUnit, RhinoMocks and TestDriven.NET.

TestDriven.NET is an add-on for Visual Studio that basically adds a "Run Test" option to our context (right-click) menu but we won't spend any time talking about it. The personal license of TestDriven.NET is only valid for open source and trial users. However 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. (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.

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.

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 (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:

```
using NUnit.Framework;

[TestFixture]
public class CarTests
{
   [SetUp]
   public void SetUp() {  //todo }

   [TearDown]
   public void TearDown(){  //todo }


   [Test]
```

```
   public void SaveThrowsExceptionWhenInvalid(){  //todo  }

   [Test]
   public void SaveCallsDataAccessAndSetsId(){  //todo }

  //more tests
}
```

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:

```
[Test]
public void MathUtilityReturnsZeroWhenNoParameters()
{
 Assert.AreEqual(0, MathUtility.Add());
}
[Test]
public void MathUtilityReturnsValueWhenPassedOneValue()
{
 Assert.AreEqual(10, MathUtility.Add(10));
}
[Test]
public void MathUtilityReturnsValueWhenPassedMultipleValues()
{
 Assert.AreEqual(29, MathUtility.Add(10,2,17));
}
[Test]
public void MathUtilityWrapsOnOverflow()
{
 Assert.AreEqual(-2, MathUtility.Add(int.MaxValue, int.MaxValue));
}
```

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. If we were to write tests for our `Car`'s `Save` method, we wouldn't write an all encompassing test, 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 say, 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.

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 chapter 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 RhinoMocks 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 RhinoMocks 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:

```
[TestFixture]
public class CarTest
{
   [Test]
   public void SaveCarCallsUpdateWhenAlreadyExistingCar()
   {

      MockRepository mocks  = new MockRepository();
      IDataAccess dataAccess = mocks.CreateMock<IDataAccess>();
      ObjectFactory.InjectStub(typeof(IDataAccess), dataAccess);

      Car car = new Car();
      dataAccess.Update(car);
      mocks.ReplayAll();

      car.Id = 32;
      car.Save();

      mocks.VerifyAll();
      ObjectFactory.ResetDefaults();
   }
}
```

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 RhinoMocks (nothing really happens since `dataAcces` is simply a mock object with no real implementation). 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, RhinoMocks 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.

We can test all of this by forcing our test to fail (notice the extra `dataAccess.Update` call):

```
[Test]
public void SaveCarCallsUpdateWhenAlreadyExistingCar()
{
   MockRepository mocks  = new MockRepository();
   IDataAccess dataAccess = mocks.CreateMock<IDataAccess>();
   ObjectFactory.InjectStub(typeof(IDataAccess), dataAccess);

   Car car = new Car();
   dataAccess.Update(car);
   dataAccess.Update(car);
   mocks.ReplayAll();

   car.Id = 32;
   car.Save();

   mocks.VerifyAll();
   ObjectFactory.ResetDefaults();
}
```

Our test will fail with a message from RhinoMocks saying two calls to `Update` were expected, but only one actually occurred.

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:

```
[Test]
public void SaveCarCallsSaveWhenNew()
{
 MockRepository mocks  = new MockRepository();
 IDataAccess dataAccess = mocks.CreateMock<IDataAccess>();
 ObjectFactory.InjectStub(typeof(IDataAccess), dataAccess);

 Car car = new Car();
 Expect.Call(dataAccess.Save(car)).Return(389);
 mocks.ReplayAll();

 car.Save();

 mocks.VerifyAll();
 Assert.AreEqual(389, car.Id);
 ObjectFactory.ResetDefaults();
}
```

Using the `Expect.Call` method allows us to specify the return value we want. Also notice the `Assert.Equal` 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.

If we changed our `Save` function to throw an exception if the returned id was invalid, our test would look like:

```
[TestFixture]
public class CarTest
{
   private MockRepository _mocks;
   private IDataAccess _dataAccess;

   [SetUp]
   public void SetUp()
   {
      _mocks = new MockRepository();
      _dataAccess = _mocks.CreateMock<IDataAccess>();
      ObjectFactory.InjectStub(typeof(IDataAccess), _dataAccess);
   }
   [TearDown]
   public void TearDown()
   {
      _mocks.VerifyAll();
   }

   [Test, ExpectedException("CodeBetter.Foundations.PersistenceException")]
   public void SaveCarCallsSaveWhenNew()
   {
      Car car = new Car();
      Expect.Call(_dataAccess.Save(car)).Return(0);
      _mocks.ReplayAll();
      car.Save();
   }
}
```

In addition to showing how you can test for an exception (via the `ExpectedException` attribute), we've also extracted the repetitive code that creates, sets up and verifies the mock object into the SetUp and TearDown methods.

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 that 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).

Combined with a utility like NCover, 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 a 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 book. 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 like any other class (depending on exactly what it is your doing of course).

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.

In This Chapter

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 tests (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!

Object Relational Mappers

6

> THE OTHER OPTION INVOLVED WRITING WAY TOO MUCH SQL. - CHRIS KOCH

In chapter 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 objects and more functionality would bloat our DAL into an enormously unmaintainable violation of DRY (don't repeat yourself). In this chapter 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.

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:

```
string sql = @"SELECT UserId FROM Users
                WHERE UserName = '" + userName + "'
                   AND Password = '" + password + "'";
 using (SqlCommand command = new SqlCommand(sql))
 {
    return 0; //todo
 }
```

Of course, phrased this way, inline SQL really does suck. However, if you stop and think about it and actually compare apples to apples, the truth is that neither is particularly better than the other. Let's examine some common points of contention.

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:

```
string sql = @"SELECT UserId FROM Users
                WHERE UserName = @UserName AND Password = @Password";
 using (SqlCommand command = new SqlCommand(sql))
 {
    command.Parameters.Add("@UserName", SqlDbType.VarChar).Value = userName;
    command.Parameters.Add("@Password ", SqlDbType.VarChar).Value = password;
    return 0; //todo
 }
```

From there on, there's not much difference - views can be used or database roles / users can be set up with appropriate permissions.

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. Essentially, it's the same code, simply residing in a different location, therefore it cannot provide greater abstraction. O/R Mappers on the other side, generally provide much better abstraction by being configurable, and implementing their own query language.

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 days 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 nanoseconds. On top of that, a well configured O/R mapper can save round-trips via identify map implementations, caching and lazy loading.

> *As I've said before, I think it's generally better to err on the side of simplicity whenever possible. Writing a bunch of mindless stored procedures to perform every database operation you think you may need is definitely not what I'd call simple ... I'm certainly not ruling out the use of stored procedures, but to start with procs? That seems like a fairly extreme case of premature optimization to me. - Jeff Atwood, codinghorror.com*

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 as the one about C# and VB.NET. If it was just a matter of one or the other, then pick whichever you prefer and move on to your next challenge. If there was nothing more to say on the topic, I'd pick stored procedures. However, when you add an O/R mapper into the mix, you suddenly gain significant advantages. You stop participating in stupid flame wars, and simply say "I want that!".

Specifically, there are three major benefits to be had with O/R mappers:

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. Your code becomes simpler- if your impedance mismatch is low, you'll write far less repetitive code. If your impedance mismatch is high you won't have to compromise your database design and your domain design - you can build them both in an optimized way, and let the O/R mapper manage the mismatch.

In the end, this really comes down to building the simplest solution upfront. Optimizations ought to be left until after you've profiled your code and identified actual bottlenecks. Like most things, it might not sound that simple because of the fairly complex learning to do upfront, but that's the reality of our profession.

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. This not only provides separation between your table structure and domain objects, but also decouples your code from a specific database implementation.

Remember, our goal is to widen our knowledgebase 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.

In previous chapters we focused on a system for a car dealership – specifically focusing on cars and upgrades. In this chapter 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`.

Also included is a VS.NET solution that contains sample code and annotations – you can find a link at the end of this chapter. 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 chapter, are meant to help you get started with NHibernate – a topic too often overlooked.

Finally, you'll find the NHibernate reference manual to be of exceptional quality, both as a helpful tool to get started, and as a reference to lookup specific topics. There's also a book being published by Manning, NHibernate in Action, that'll be available in June. In the meantime, you can purchase a pre-release version of the book in electronic format.

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 NHibernate itself. The simplest configuration, which must be added to your app.config or web.config, looks like:

```
<configuration>
  <configSections>
    <section name="hibernate-configuration"
             type="NHibernate.Cfg.ConfigurationSectionHandler, NHibernate" />
  </configSections>
  <hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
    <session-factory>
      <property name="hibernate.dialect">
         NHibernate.Dialect.MsSql2005Dialect
      </property>
      <property name="hibernate.connection.provider">
         NHibernate.Connection.DriverConnectionProvider
      </property>
      <property name="hibernate.connection.connection_string">
         Server=SERVER;Initial Catalog=DB;User Id=USER;Password=PASSWORD;
      </property>
      <mapping assembly="CodeBetter.Foundations" />
    </session-factory>
  </hibernate-configuration>
</configuration>
```

Of the four values, `dialect` is the most interesting. This tells NHibernate what specific language our database speaks. If, in our code, 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 `CodeBetter.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, named `Model.hbm.xml`, looks like:

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



```
</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 for example), 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:

```
private static ISessionFactory _sessionFactory;
public void Sample()
{
 //Let's add a new car model
 Model model = new Model();
 model.Name = "Hummbee";
 model.Description = "Great handling, built-in GPS to always find your
                      way back home, Hummbee2Hummbe(tm) communication";
 model.Price = 50000.00;
 ISession session = _sessionFactory.OpenSession();
 session.Save(model);

 //Let's discount the x149 model
 IQuery query = session.CreateQuery("from Model model where model.Name = ?");
 Model model = query.SetString(0, "X149").UniqueResult<Model>();
 model.Price -= 5000;
 ISession session = _sessionFactory.OpenSession();
 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` (which implements `ISessionFactory`) is a global thread-safe object that you'd likely create on application start. You'll typically need one per database that 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 salespeople, so that we can provide some basic reports. We're told that a sale can only ever belong to a single salesperson, and thus set up a one-to-many relationship – that is, a salesperson can have multiple sales, and a sales can only belong to a single salesperson. 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 (reference).

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:

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

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 salesperson 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 indination 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<T>` 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 `HashSet` which is fast for larger collections and the `HybridSet` which initially uses a `ListSet` and automatically switches itself to a `HashSet` 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`:

```
private IList<Sale> _sales;
public IList<Sale> Sales
{
   get { return _sales;}
}
```

and add our `<bag>` element to the `SalesPerson` mapping file:

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

With the release of .NET 3.5, a HashSet collection has finally been added to the framework. Hopefully, future versions will add other types of sets, such as an OrderedSet. Sets are very useful and efficient collections, so consider adding them to your arsenal of tools! You can learn more by reading Jason Smith's article describe sets.

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 `SalesPerson` object, 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 `IQuery` you can return individual entities, collections, substitute parameters and more. Here are some examples:

```
string lastName = "allen";
ISession session = _sessionFactory.OpenSession();

//retrieve a salesperson by last name
IQuery query = s.CreateQuery("from SalesPerson p where p.LastName =
                              'allen'");
SalesPerson p = query.UniqueResult<SalesPerson>();

//same as above but in 1 line, and with the last name as a variable
SalesPerson p = session.CreateQuery("from SalesPerson p where p.LastName =
?").SetString(0, lastName).UniqueResult<SalesPerson>();

//people with few sales
IList<SalesPerson> slackers = session.CreateQuery("from SalesPerson person
where size(person.Sales) < 5").List<SalesPerson>();
```

This is just a subset of what can be accomplished with HQL (the downloadable sample has slightly more complicated examples).

Lazy Loading

When we load a salesperson, say by doing: `SalesPerson person = session.Get<SalesPerson>(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 `SalesPerson`, we need to specify the `Model`, but don't want to load every property – 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<T>(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:

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



```
session.SaveOrUpdate(salesPerson);
```

without ever having to actually hit the database to load the `Model`.

Download

You can download a sample project with more examples of NHibernate usage at: http://codebetter.com/files/folders/codebetter_downloads/entry172562.aspx. The code is heavily documented to explain various aspects of using NHibernate. (If the above link does not work for you, you can try this alternative download location: http://openmymind.net/CodeBetter.Foundations.zip).

In This Chapter

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), 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 but, stepping beyond what's comfortable, it's impossible to ignore the benefits of O/R mappers.

You're more than half way through! I hope you're enjoying yourself and learning a lot. This might be a good time to take a break from reading and get a little more hands-on with the free Canvas Learning Application.

Back to Basics: Memory

7

> NOT 'GETTING' ALGEBRA IS NOT ACCEPTABLE FOR A MATHEMATICIAN, AS NOT 'GETTING' POINTERS IS NOT ACCEPTABLE FOR PROGRAMMERS. TOO FUNDAMENTAL.
> - WARD CUNNINGHAM

Try as they might, modern programming language can't fully abstract fundamental aspects of computer systems. This is made evident by the various exceptions thrown by high level languages. For example, it's safe to assume that you've likely faced the following .NET exceptions: `NullReferneceException`, `OutOfMemoryException`, `StackOverflowException` and `ThreadAbortException`. As important as it is for developers to embrace various high level patterns and techniques, it's equally important to understand the ecosystem in which your program runs. Looking past the layers provided by the C# (or VB.NET) compiler, the CLR and the operating system, we find memory. All programs make extensive use of system memory and interact with it in marvelous ways, it's difficult to be a good programmer without understanding this fundamental interaction.

Much of the confusion about memory stems from the fact that C# and VB.NET are managed languages and that the CLR provides automatic garbage collection. This has caused many developers to erroneously assume that they need not worry about memory.

Memory Allocation

In .NET, as with most languages, every variable you define is either stored on the stack or in the heap. These are two separate spaces allocated in system memory which serve a distinct, yet complimentary purpose. What goes where is predetermined: value types go on the stack, while all reference types go on the heap. In other words, all the system types, such as `char`, `int`, `long`, `byte`, `enum` and any structures (either defined in .NET or defined by you) go on the stack. The only exception to this rule are value types belonging to reference types - for example the `Id` property of a `User` class goes on the heap along with the instance of the `User` class itself.

The Stack

Although we're used to magical garbage collection, values on the stack are automatically managed even in a garbage collectionless world (such as C). That's because whenever you enter a new scope (such as a method or an if statement) values are pushed onto the stack and when you exit the stack the values are popped off. This is why a stack is synonymous with a LIFO - last-in first-out. You can think of it this way: whenever you create a new scope, say a method, a marker is placed on the stack and values are added to it as needed. When you leave that scope, all values are popped off up to and including the method marker. This works with any level of nesting.

If you've ever wondered why a variable defined in a for loop or if statement wasn't available outside that scope, it's because the stack has unwound itself and the value is lost.

Until we look at the interaction between the heap and the stack, the only real way to get in trouble with the stack is with the `StackOverflowException`. This means that you've used up all the space available on the stack. 99.9% of the time, this indicates an endless recursive call (a function which calls itself ad infinitum). In theory it could be caused by a very, very poorly designed system, though I've never seen a non-recursive call use up all the space on the stack.

The Heap

Memory allocation on the heap isn't as straightforward as on the stack. Most heap-based memory allocation occurs whenever we create a `new` object. The compiler figures out how much memory we'll need (which isn't that difficult, even for objects with nested references), carves up an appropriate chunk of memory and returns a pointer to the allocated memory (more on this in moments). The simplest example is a string, if each character in a string takes up 2 bytes, and we create a new string with the value of "Hello World", then the CLR will need to allocate 22 bytes (11x2) plus whatever overhead is needed.

Speaking of strings, you've no doubt heard that string are immutable - that is, once you've declared a string and assigned it a value, if you modify that string (by changing its value, or concatenating another string onto it), then a new string is created. This can actually have negative performance implications, and so the general recommendation is to use a `StringBuilder` for any significant string manipulation. The truth though is that any object stored on the heap is immutable with respect to size allocation, and any changes to the underlying size will require new allocation. The `StringBuilder`, along with some collections, partially get around this by using internal buffers. Once the buffer fills up though, the same reallocation occurs and some type of growth algorithm is used to determined the new size (the simplest being oldSize * 2). Whenever possible it's a good idea to specify the initial capacity of such objects in order to avoid this type of reallocation (the constructor for both the `StringBuilder` and the `ArrayList` (amongst many other collections) allow you to specify an initial capacity).

Garbage collecting the heap is a non-trivial task. Unlike the stack where the last scope can simply be popped off, objects in the heap aren't local to a given scope. Instead, most are deeply nested references of other referenced objects. In languages such as C, whenever a programmer causes memory to be allocated on the heap, he or she must also make sure to remove it from the heap when he's finished with it. In managed languages, the runtime takes care of cleaning up resources (.NET uses a Generational Garbage Collector which is briefly described on Wikipedia).

There are a lot of nasty issues that can sting developers while working with the heap. Memory leaks aren't only possible but very common, memory fragmentation can cause all types of havoc, and various performance issues can arise due to strange allocation behavior or interaction with unmanaged code (which.NET does a lot under the covers).

Pointers

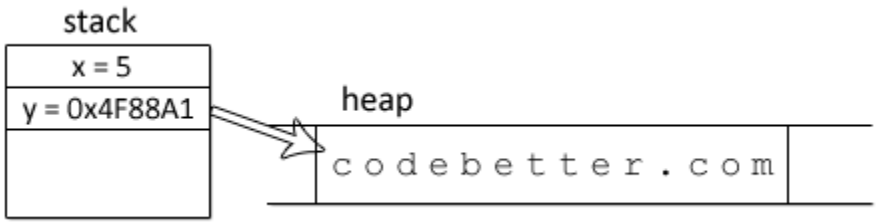
For many developers, learning pointers in school was a painful experience. They represent the very real indirection which exists between code and hardware. Many more developers have never had the experience of learning them - having jumped into programming directly from a language which didn't expose them directly. The truth though is that anyone claiming that C# or Java are pointerless languages is simply wrong. Since pointers are the mechanism by which all languages manage values on the heap, it seems rather silly not to understand how they are used.

Pointers represent the nexus of a system's memory model - that is, pointers are the mechanism by which the stack and the heap work together to provide the memory subsystem required by your program. As we discussed earlier, whenever you instantiate a `new` object, .NET allocates a chunk of memory on the heap and returns a pointer to the start of this memory block. This is all a pointer is: **the starting address for the block of memory containing an object**. This address is really nothing more than an unique number, generally represented in hexadecimal format. Therefore, a pointer is nothing more than a unique number that tells .NET where the actual object is in memory. When you assign a reference type to a variable, your variable is actually a pointer to the object. This indirection is transparent in Java or .NET, but not in C or C++ where you can manipulate the memory address directly via pointer arithmetic. In C or C++ you could take a pointer and add 1 to it, hence arbitrarily changing where it points to (and likely crashing your program because of it).

Where it gets interesting is where the pointer is actually stored. They actually follow the same rules outlined above: as integers they are stored on the stack - unless of course they are part of a reference object and then they are on the heap with the rest of their object. It might not be clear yet, but this means that ultimately, all heap objects are rooted on the stack (possibly through numerous levels of references). Let's first look at a simple example:

```
static void Main(string[] args)
{
  int x = 5;
  string y = "codebetter.com";
}
```

From the above code, we'll end up with 2 values on the stack, the integer 5 and the pointer to our string, as well as the actual string on the heap. Here's a graphical representation:



When we exit our main function (forget the fact that the program will stop), our stack pops off all local values, meaning both the x and y values are lost. This is significant because the memory allocated on the heap still contains our string, but we've lost all references to it (there's no pointer pointing back to it). In C or C++ this results in a memory leak - without a reference to our heap address we can't free up the memory. In C# or Java, our trusty garbage collector will detect the unreferenced object and free it up.

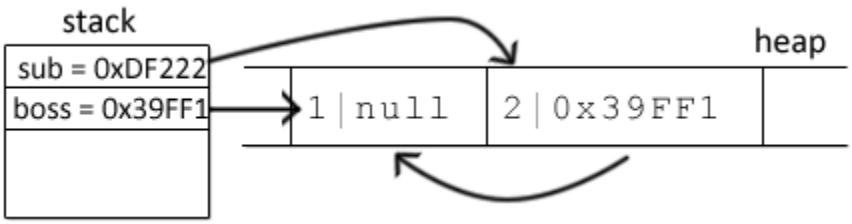
We'll look at a more complex examples, but aside from having more arrows, it's basically the same.

```
public class Employee
{
  private int _employeeId;
  private Employee _manager;

  public int EmployeeId
  {
    get { return _employeeId; }
    set { _employeeId = value; }
  }
  public Employee Manager
  {
    get { return _manager; }
    set { _manager = value; }
  }

  public Employee(int employeeId)
  {
    _employeeId = employeeId;
  }
}
public class Test
{
  private Employee _subordinate;

  void DoSomething()
  {
    Employee boss = new Employee(1);
    _subordinate = new Employee(2);
    _subordinate.Manager = _boss;
  }
}
```



Some prefer to call the type of pointers found in C#, VB.NET and Java References or Safe Pointers, because they either point to a valid object or null. There is no such guarantee in languages such as C or C++ since developers are free to manipulate pointers directly.

Interestingly, when we leave our method, the boss variable will pop off the stack, but the subordinate, which is defined in a parent scope, won't. This means the garbage collector won't have anything to clean-up because both heap values will still be referenced (one directly from the stack, and the other indirectly from the stack through a referenced object).

As you can see, pointers most definitely play a significant part in both C# and VB.NET. Since pointer arithmetic isn't available in either language, pointers are greatly simplified and hopefully easily understood.

Memory Model in Practice

We'll now look at the actual impact this has on our applications. Keep in mind though that understanding the memory model in play won't only help you avoid pitfalls, but it will also help you write better applications.

Boxing

Boxing occurs when a value type (stored on the stack) is coerced onto the heap. Unboxing happens when these value types are placed back onto the stack. The simplest way to coerce a value type, such as an integer, onto the heap is by casting it:

```
int x = 5;
object y = x;
```

A more common scenario where boxing occurs is when you supply a value type to a method that accepts an object. This was common with collections in .NET 1.x before the introduction of generics. The non-generic collection classes mostly work with the object type, so the following code results in boxing and unboxing:

```
ArrayList userIds = new ArrayList(2);
userIds.Add(1);
userIds.Add(2);;
int firstId = (int)userIds[0];
```

The real benefit of generics is the increase in type-safety, but they also address the performance penalty associated with boxing. In most cases you wouldn't notice this penalty, but in some situations, such as large collections, you very well could. Regardless of whether or not it's something you ought to actually concern yourself with, boxing is a prime example of how the underlying memory system can have an impact on your application.

ByRef

Without a good understanding of pointers, it's virtually impossible to understand passing a value by reference and by value. Developers generally understand the implication of passing a value type, such as an integer, by reference, but few understand why you'd want to pass a reference by reference. ByRef and ByVal affect reference and value types the same - provided you understand that they always work against the underlying value (which in the case of a reference type means they work against the pointer and not the value). Using ByRef is the only common situation where .NET won't automatically resolve the pointer indirection (passing by reference or as an output parameter isn't allowed in Java).

First we'll look at how ByVal/ByRef affects value types. Given the following code:

```
public static void Main()
{
  int counter1 = 0;
  SeedCounter(counter1);
  Console.WriteLine(counter1);

  int counter2 = 0;
  SeedCounter(ref counter2);
  Console.WriteLine(counter2);
}

private static void SeedCounter(int counter)
{
  counter = 1;
}
private static void SeedCounter(ref int counter)
{
  counter = 1;
}
```

We can expect an output of 0 proceeded by 1. The first call **does not** pass `counter1` by reference, meaning a copy of counter1 is passed into `SeedCounter` and changes made within are local to the function. In other words, we're taking the value on the stack and duplicating it onto another stack location.

In the second case we're actually passing the value by reference which means no copy is created and changes aren't localized to the `SeedCounter` function.

The behavior with reference types is the exact same, although it might not appear so at first. We'll look at two examples. The first one uses a `PayManagement` class to change the properties of an `Employee`. In the code below we see that we have two employees and in both cases we're giving them a $2000 raise. The only difference is that one passes the employee by reference while the other is passed by value. Can you guess the output?


```
public class Employee
{
  private int _salary;
  public int Salary
  {
    get {return _salary;}
    set {_salary = value;}
  }
  public Employee(int startingSalary)
  {
    _salary = startingSalary;
  }
}

public class PayManagement
{
  public static void GiveRaise(Employee employee, int raise)
  {
    employee.Salary += raise;
  }
  public static void GiveRaise(ref Employee employee, int raise)
  {
    employee.Salary += raise;
  }
}

public static void Main()
{
  Employee employee1 = new Employee(10000);
  PayManagement.GiveRaise(employee1, 2000);
  Console.WriteLine(employee1.Salary);

  Employee employee2 = new Employee(10000);
  PayManagement.GiveRaise(ref employee2, 2000);
  Console.WriteLine(employee2.Salary);
}
```

In both cases, the output is 12000. At first glance, this seems different than what we just saw with value types. What's happening is that passing a reference type by value does indeed pass a copy of the value, but not the heap value. Instead, we're passing a copy of our pointer. And since a pointer and a copy of the pointer point to the same memory on the heap, a change made by one is reflected in the other.

When you pass a reference type by reference, you're passing the actual pointer as opposed to a copy of the pointer. This begs the question, when would we ever pass a reference type by reference? The only reason to pass by reference is when you want to modify the pointer itself - as in where it points to. This can actually result in nasty side effects - which is why it's a good thing functions wanting to do so must specifically specify that they want the parameter passed by reference. Let's look at our second example.


```
public class Employee
{
  private int _salary;
  public int Salary
  {
    get {return _salary;}
    set {_salary = value;}
  }
  public Employee(int startingSalary)
  {
    _salary = startingSalary;
  }
}

public class PayManagement
{
  public static void Terminate(Employee employee)
  {
    employee = null;
  }
  public static void Terminate(ref Employee employee)
  {
    employee = null;
  }
}

public static void Main()
{
  Employee employee1 = new Employee(10000);
  PayManagement.Terminate(employee1);
  Console.WriteLine(employee1.Salary);

  Employee employee2 = new Employee(10000);
  PayManagement.Terminate(ref employee2);
  Console.WriteLine(employee2.Salary);
}
```

Try to figure out what will happen and why. I'll give you a hint: an exception will be thrown. If you guessed that the call to `employee1.Salary` outputted 10000 while the 2nd one threw a `NullReferenceException` then you're right. In the first case we're simply setting a copy of the original pointer to null - it has no impact whatsoever on what `employee1` is pointing to. In the second case, we aren't passing a copy but the same stack value used by `employee2`. Thus setting the employee to null is the same as writing `employee2 = null;`.

It's quite uncommon to want to change the address pointed to by a variable from within a separate method - which is why the only time you're likely to see a reference type passed by reference is when you want to return multiple values from a function call (in which case you're better off using an `out` parameter, or using a purer OO approach). The above example truly highlights the dangers of playing in an environment whose rules aren't fully understood.

Managed Memory Leaks

We already saw an example of what a memory leak looks like in C. Basically, if C# didn't have a garbage collector, the following code would leak:

```
private void DoSomething()
{
   string name = "dune";
}
```

Our stack value (a pointer) will be popped off, and with it will go the only way we have to reference the memory created to hold our string. Leaving us with no method of freeing it up. This isn't a problem in .NET because it does have a garbage collector which tracks unreferenced memory and frees it. However, a type of memory leak is still possible if you hold on to references indefinitely. This is common in large applications with deeply nested references. They can be hard to identify because the leak might be very small and your application might not run for long enough

Ultimately when your program terminates the operating system will reclaim all memory, leaked or otherwise. However, if you start seeing `OutOfMemoryException` and aren't dealing with abnormally large data, there's a good chance you have a memory leak. .NET ships with tools to help you out, but you'll likely want to take advantage of a commercial memory profiler such as dotTrace or ANTS Profiler. When hunting for memory leaks you'll be looking for your leaked object (which is pretty easy to find by taking 2 snapshots of your memory and comparing them), tracing through all the objects which still hold a reference to it and correcting the issue.

There's one specific situation worth mentioning as a common cause of memory leaks: events. If, in a class, you register for an event, a reference is created to your class. Unless you de-register from the event your objects lifecycle will ultimately be determined by the event source. In other words, if `ClassA` (the listener) registers for an event in `ClassB` (the event source) a reference is created from `ClassB` to `ClassA`. Two solutions exists: de-registering from events when you're done (the IDisposable pattern is the ideal solution), or use the WeakEvent Pattern or a simplified version.

Fragmentation

Another common cause for `OutOfMemoryException` has to do with memory fragmentation. When memory is allocated on the heap it's always a continuous block. This means that the available memory must be scanned for a large enough chunk. As your program runs its course, the heap becomes increasingly fragmented (like your hard drive) and you might end up with plenty of space, but spread out in a manner which makes it unusable. Under normal circumstances, the garbage collector will compact the heap as it's freeing memory. As it compacts memory, addresses of objects change and .NET makes sure to update all your references accordingly. Sometimes though, .NET can't move an object: namely when the object is pinned to a specific memory address.

Pinning

Pinning occurs when an object is locked to a specific address on the heap. Pinned memory cannot be compacted by the garbage collector resulting in fragmentation. Why do values get pinned? The most common cause is because your code is interacting with unmanaged code. When the .NET garbage collector compacts the heap, it updates all references in managed code, but it has no way to jump into unmanaged code and do the same. Therefore, before interoping it must first pin objects in memory. Since many methods within the .NET framework rely on unmanaged code, pinning can happen without you knowing about it (the scenario I'm most familiar with are the .NET Socket classes which rely on unmanaged implementations and pin buffers).

A common way around this type of pinning is to declare large objects which don't cause as much fragmentation as many small ones (this is even more true considering large objects are placed in a special heap (called the Large Object Heap (LOH) which isn't compacted at all). For example, rather than creating hundreds of 4KB buffers, you can create 1 large buffer and assign chunks of it yourself. For an example as well as more information on pinning, I suggest you read Greg Young's advanced post on pinning and asynchronous sockets.

There's a second reason why an object might be pinned - when you explicitly make it happen. In C# (not in VB.NET) if you compile your assembly with the `unsafe` option, you can pin an object via the `fixed` statement. While extensive pinning can cause memory pressures on the system, judicial use of the `fixed` statement can greatly improve performance. Why? Because a pinned object can be manipulated directly with pointer arithmetic - this isn't possible if the object isn't pinned because the garbage collector might reallocate your object somewhere else in memory.

Take for example this efficient ASCII string to integer conversion which runs over 6 times faster than using `int.Parse`.

```
public unsafe static int Parse(string stringToConvert)
{
  int value = 0;
  int length = stringToConvert.Length;
  fixed(char* characters = stringToConvert)
  {
    for (int i = 0; i < length; ++i)
    {
      value = 10 * value + (characters[i] - 48);
    }
  }
  return value;
}
```

Unless you're doing something abnormal, there should never be a need to mark your assembly as unsafe and take advantage of the `fixed` statement. The above code will easily crash (pass null as the string and see what happens), isn't nearly as feature rich as `int.Parse`, and in the scale of things is extremely risky while providing no benefits.

Setting things to null

So, should you set your reference types to null when you're done with them? Of course not. Once a variable falls out of scope, it's popped of the stack and the reference is removed. If you can't wait for the scope to exit, you likely need to refactor your code.

Deterministic Finalization

Despite the presence of the garbage collector, developers must still take care of managing some of their references. That's because some objects hold on to vital or limited resources, such as file handles or database connections which should be released as soon as possible. This is problematic since we don't know when the garbage collector will actually run - by nature the garbage collector only runs when memory is in short supply. To compensate, classes which hold on to such resources should make use of the Disposable pattern. All .NET developers are likely familiar with this pattern, along with its actual implementation (the `IDisposable` interface), so we won't rehash what you already know. With respect to this chapter, it's simply important that you understand the role deterministic finalization takes. It doesn't free the memory used by the object. It releases resources. In the case of database connections for example, it releases the connection back to the pool in order to be reused.

If you forget to call `Dispose` on an object which implements `IDisposable`, the garbage collector will do it for you (eventually). You shouldn't rely on this behavior, however, as the problem of limited resources is very real (it's relatively trivial to try it out with a loop that opens connections to a database). You may be wondering why some objects expose both a `Close` and `Dispose` method, and which you should call. In all the cases I've seen the two are generally equivalent - so it's really a matter of taste. I would suggest that you take advantage of the `using` statement and forget about `Close`. Personally I find it frustrating (and inconsistent) that both are exposed.

Finally, if you're building a class that would benefit from deterministic finalization you'll find that implementing the `IDisposable` pattern is simple. A straightforward guide is available on MSDN.

In This Chapter

Stacks, heaps and pointers can seem overwhelming at first. Within the context of managed languages though, there isn't really much to it. The benefits of understanding these concepts are tangible in day to day programming, and invaluable when unexpected behavior occurs. You can either be the programmer who causes weird `NullReferenceExceptions` and `OutOfMemoryExceptions`, or the one that fixes them.

Back to Basics: Exceptions

8

FAIL FAST - JIM SHORE

Exceptions are such powerful constructs that developers can get a little overwhelmed and far too defensive when dealing with them. This is unfortunate because exceptions actually represent a key opportunity for developers to make their system considerably more robust. In this chapter we'll look at three distinct aspects of exceptions : handling, creating and throwing them. Since exceptions are unavoidable you can neither run nor hide, so you might as well leverage.

Handling Exceptions

Your strategy for handling exceptions should consist of two golden rules:

1. Only handle exceptions that you can actually do something about, and
2. You can't do anything about the vast majority of exceptions

Most new developers do the exact opposite of the first rule, and fight hopelessly against the second. When your application does something deemed exceptionally outside of its normal operation the best thing to do is fail it right then and there. If you don't you won't only lose vital information about your mystery bug, but you risk placing your application in an unknown state, which can result in far worse consequences.

Whenever you find yourself writing a try/catch statement, ask yourself if you can actually do something about a raised exception. If your database goes down, can you actually write code to recover or are you better off displaying a friendly error message to the user and getting a notification about the problem? It's hard to accept at first, but sometimes it's just better to crash, log the error and move on. Even for mission critical systems, if you're making typical use of a database, what can you do if it goes down? This train of thought isn't limited to database issues or even just environmental failures, but also your typical every-day runtime bug . If converting a configuration value to an integer throws a `FormatException` does it make sense continuing as if everything's ok? Probably not.

Of course, if you can handle an exception you absolutely ought to - but do make sure to catch only the type of exception you can handle. Catching exceptions and not actually handling them is called exception swallowing (I prefer to call it wishful thinking) and it's a bad code. A common example I see has to do with input validation. For example, let's look at how not to handle a `categoryId` being passed from the `QueryString` of an ASP.NET page.

```
int categoryId;
try
{
  categoryId = int.Parse(Request.QueryString["categoryId"]);
}
catch(Exception)
```

```
{
  categoryId = 1;
}
```

The problem with the above code is that regardless of the type of exception thrown, it'll be handled the same way. But does setting the `categoryId` to a default value of 1 actually handle an `OutOfMemoryException`? Instead, the above could should catch a specific exception:

```
int categoryId;
try
{
   categoryId = int.Parse(Request.QueryString["categoryId"])
}
catch(FormatException)
{
   categoryId = -1;
}
```

(an even better approach would be the use the `int.TryParse` function introduced in .NET 2.0 - especially considering that `int.Parse` can throw two other types of exceptions that we'd want to handle the same way, but that's besides the point).

Logging

Even though most exceptions are going to go unhandled, you should still log each and every one of them. Ideally you'll centralize your logging - an HttpModule's `OnError` event is your best choice for an ASP.NET application or web service. I've often seen developers catch exceptions where they occur only to log and rethrow (more on rethrowing in a bit). This causes a lot of unnecessary and repetitive code - better to let exceptions bubble up through your code and log all exceptions at the outer edge of your system. Exactly which logging implementation you use is up to you and will depend on the criticalness of your system. Maybe you'll want to be notified by email as soon as exceptions occur, or maybe you can simply log it to a file or database and either review it daily or have another process send you a daily summary. Many developers leverage rich logging frameworks such as log4net or Microsoft's Logging Application Block.

> *A word of warning based on a bad personal experience: some types of exceptions tend to cluster. If you choose to send out emails whenever an exception occurs you can easily flood your mail server . A smart logging solution should probably implement some type of buffering or aggregation.*

Cleaning Up

In the previous chapter we talked about deterministic finalization with respect to the lazy nature of the garbage collector. Exceptions prove to be an added complexity as their abrupt nature can cause `Dispose` not to be called. A failed database call is a classic example:

```
SqlConnection connection = new SqlConnection(FROM_CONFIGURATION)
SqlCommand command = new SqlCommand("SomeSQL", connection);
connection.Open();
command.ExecuteNonQuery();
command.Dispose();
connection.Dispose();
```

If `ExecuteNonQuery` throws an exception, neither our `command` nor our `connection` will get disposed of. The solution is to use `Try/Finally`:

```
SqlConnection connection;
SqlCommand command;
try
{
   connection = new SqlConnection(FROM_CONFIGURATION)
   command = new SqlCommand("SomeSQL", connection);
   connection.Open();
   command.ExecuteNonQuery();
}
finally
{
   if (command != null) { command.Dispose(); }
   if (connection != null) { connection.Dispose(); }
}
```

or the syntactically nicer `using` statement (which gets compiled to the same try/finally above):

```
using (SqlConnection connection = new SqlConnection(FROM_CONFIGURATION))
using (SqlCommand command = new SqlCommand("SomeSQL", connection))
{
   connection.Open();
   command.ExecuteNonQuery();
}
```

The point is that even if you can't handle an exception, and you should centralize all your logging, you do need to be mindful of where exceptions can crop up - especially when it comes to classes that implement `IDiposable`.

Throwing Exceptions

There isn't one magic rule to throwing exceptions like there is for catching them (again, that rule is don't catch exceptions unless you can actually handle them). Nonetheless throwing exceptions, whether or not they be your own (which we'll cover next), is still pretty simple. First we'll look at the actual mechanics of throwing exceptions, which relies on the `throw` statement. Then we'll examine when and why you actually want to throw exceptions.

Throwing Mechanics

You can either throw a new exception, or rethrow a caught exception. To throw a new exception, simply create a new exception and `throw` it.

```
throw new Exception("something bad happened!");

//or

Exception ex = new Exception("something bad happened");
throw ex;
```

I added the second example because some developers think exceptions are some special/unique case - but the truth is that they are just like any other object (except they inherit from `System.Exception` which in turn inherits from `System.Object`). In fact, just because you create a new `exception` doesn't mean you have to throw it - although you probably always would.

On occasion you'll need to rethrow an exception because, while you can't handle the exception, you still need to execute some code when an exception occurs. The most common example is having to rollback a transaction on failure:

```
ITransaction transaction = null;
try
{
  transaction = session.BeginTransaction();
  // do some work
  transaction.Commit();
}
catch
{
  if (transaction != null) { transaction.Rollback(); }
  throw;
}
finally
{
  //cleanup
}
```

In the above example our vanilla `throw` statement makes our catch transparent. That is, a handler up the chain of execution won't have any indication that we caught the exception. In most cases, this is

what we want - rolling back our transaction really doesn't help anyone else handle the exception. However, there's a way to rethrow an exception which will make it look like the exception occurred within our code:

```
catch (HibernateException ex)
{
  if (transaction != null) { transaction.Rollback(); }
  throw ex;
}
```

By explicitly rethrowing the exception, the stack trace is modified so that the rethrowing line appears to be the source. This is almost always certainly a bad idea, as vital information is lost. So be careful how you rethrow exceptions - the difference is subtle but important.

If you find yourself in a situation where you think you want to rethrow an exception with your handler as the source, a better approach is to use a nested exception:

```
catch (HibernateException ex)
{
  if (transaction != null) { transaction.Rollback(); }
  throw new Exception("Email already in use", ex);
}
```

This way the original stack trace is still accessible via the `InnerException` property exposed by all exceptions.

When To Throw Exceptions

It's important to know how to throw exceptions. A far more interesting topic though is when and why you should throw them. Having someone else's unruly code bring down your application is one thing. Writing your own code that'll do the same thing just seems plain silly. However, a good developer isn't afraid to judicially use exceptions.

There are actually two levels of thought on how exceptions should be used. The first level, which is universally accepted, is that you shouldn't hesitate to raise an exception whenever a truly exceptional situation occurs. My favorite example is the parsing of configuration files. Many developers generously use default values for any invalid entries. This is ok in some cases, but in others it can put the system in an unreliable or unexpected state. Another example might be a Facebook application that gets an unexpected result from an API call. You could ignore the error, or you could raise an exception, log it (so that you can fix it, since the API might have changed) and present a helpful message to your users.

The other belief is that exceptions shouldn't just be reserved for exceptional situations, but for any situation in which the expected behavior cannot be executed. This approach is related to the design by

contract approach - a methodology that I'm adopting more and more every day. Essentially, if the `SaveUser` method isn't able to save the user, it should throw an exception.

In languages such as C#, VB.NET and Java, which don't support a design by contract mechanism, this approach can have mixed results. A `Hashtable` returns null when a key isn't found, but a `Dictionary` throws an exception - the unpredictable behavior sucks (if you're curious why they work differently check out Brad Abrams blog post). There's also a line between what constitutes control flow and what's considered exceptional. Exceptions shouldn't be used to control an if/else-like logic, but the bigger a part they play in a library, the more likely programmers will use them as such (the `int.Parse` method is a good example of this).

Generally speaking, I find it easy to decide what should and shouldn't throw an exception. I generally ask myself questions like:

- Is this exceptional,
- Is this expected,
- Can I continue doing something meaningful at this point and
- Is this something I should be made aware of so I can fix it, or at least give it a second look

Perhaps the most important thing to do when throwing exceptions, or dealing with exceptions in general, is to think about the user. The vast majority of users are naive compared to programmers and can easily panic when presented with error messages. Jeff Atwood recently blogged about the importance of crashing responsibly.

- IT IS NOT THE USER'S JOB TO TELL YOU ABOUT ERRORS IN YOUR SOFTWARE!
- DON'T EXPOSE USERS TO THE DEFAULT SCREEN OF DEATH.
- HAVE A DETAILED PUBLIC RECORD OF YOUR APPLICATION'S ERRORS.

It's probably safe to say that Windows' Blue Screen of Death is exactly the type of error message users shouldn't be exposed to (and don't think just because the bar has been set so low that it's ok to be as lazy).

Creating Custom Exceptions

One of the most overlooked aspect of domain driven design are custom exceptions. Exceptions play a serious part of any business domain, so any serious attempt at modeling a business domain in code must include custom exceptions. This is especially true if you believe that exceptions should be used whenever a method fails to do what it says it will. If a workflow state is invalid it makes sense to throw your own custom `WorkflowException` exception and even attach some specific information to it which might not only help you identify a potential bug, but can also be used to present meaningful information to the user.

Many of the exceptions I create are nothing more than marker exceptions - that is, they extend the base `System.Exception` class and don't provide further implementation. I liken this to marker interfaces (or marker attributes), such as the `INamingContainer` interface. These are particularly useful in allowing you to avoid swallowing exceptions. Take the following code as an example. If the `Save()` method doesn't throw a custom exception when validation fails, we really have little choice but to swallow all exceptions:

```
try
{
   user.Save();
}
catch
{
   Error.Text = user.GetErrors();
   Error.Visible = true;
}

//versus

try
{
   user.Save();
}
catch(ValidationException ex)
{
   Error.Text = ex.GetValidationMessage();
   Error.Visible = true;
}
```

The above example also shows how we can extend exceptions to provide further custom behavior specifically related to our exceptions. This can be as simple as an `ErrorCode`, to more complex information such as a `PermissionException` which exposes the user's permission and the missing required permission.

Of course, not all exceptions are tied to the domain. It's common to see more operational-oriented exceptions. If you rely on a web service which returns an error code, you may very wrap that into your own custom exception to halt execution (remember, fail fast) and leverage your logging infrastructure.

Actually creating a custom exception is a two step process. First (and technically this is all you really need) create a class, with a meaningful name, which inherits from `System.Exception`.

```
public class UpgradeException : Exception
{
}
```

You should go the extra step and mark your class with the `SerializeAttribute` and always provide at least 4 constructors:

- public YourException()
- public YourException(string message)
- public YourException(string message, Exception innerException)
- protected YourException(SerializationInfo info, StreamingContext context)

The first three allow your exception to be used in an expected manner. The fourth is used to support serialization incase .NET needs to serialize your exception - which means you should also implement the `GetObjectData` method. The purpose of support serialization is in the case where you have custom properties, which you'd like to have survive being serialized/deserialized. Here's the complete example:

```
[Serializable]
public class UpgradeException: Exception
{
  private int _upgradeId;

  public int UpgradeId { get { return _upgradeId; } }

  public UpgradeException(int upgradeId)
  {
    _upgradeId = upgradeId;
  }
  public UpgradeException(int upgradeId, string message, Exception inner)
    : base(message, inner)
  {
    _upgradeId = upgradeId;
  }
  public UpgradeException(int upgradeId, string message) : base(message)
  {
    _upgradeId = upgradeId;
  }
  protected UpgradeException(SerializationInfo info, StreamingContext c)
    : base(info, c)
  {
    if (info != null)
    {
     _upgradeId = info.GetInt32("upgradeId");
    }
  }
  public override void GetObjectData(SerializationInfo i, StreamingContext c)
  {
    if (i != null)
    {
      i.AddValue("upgradeId", _upgradeId);
    }
    base.GetObjectData(i, c)
  }
}
```


In This Chapter

It can take quite a fundamental shift in perspective to appreciate everything exceptions have to offer. Exceptions aren't something to be feared or protected against, but rather vital information about the health of your system. Don't swallow exceptions. Don't catch exceptions unless you can actually handle them. Equally important is to make use of built-in, or your own exceptions when unexpected things happen within your code. You may even expand this pattern for any method that fails to do what it says it will. Finally, exceptions are a part of the business you are modeling. As such, exceptions aren't only useful for operational purposes but should also be part of your overall domain model.

Back to Basics: Proxy This and Proxy That

9

> ONE OF THE BEAUTIES OF OO PROGRAMMING IS CODE RE-USE THROUGH INHERITANCE, BUT TO ACHIEVE IT PROGRAMMERS HAVE TO ACTUALLY LET OTHER PROGRAMMERS RE-USE THE CODE. - GARY SHORT

Few keywords are as simple yet amazingly powerful as `virtual` in C# (`overridable` in VB.NET). When you mark a method as virtual you allow an inheriting class to override the behavior. Without this functionality inheritance and polymorphism wouldn't be of much use. A simple example, slightly modified from Programming Ruby (ISBN: 978-0-9745140-5-5), which has a `KaraokeSong` overrides a `Song`'s `to_s` (ToString) function looks like:

```
class Song
   def to_s
      return sprintf("Song: %s, %s (%d)", @name, @artist, @duration)
   end
end

class KaraokeSong < Song
   def to_s
      return super + " - " @lyrics
   end
end
```

The above code shows how the `KaraokeSong` is able to build on top of the behavior of its base class. Specialization isn't just about data, it's also about behavior!

Even if your ruby is a little rusty, you might have picked up that the base `to_s` method isn't marked as virtual. That's because many languages, including Java, make methods virtual by default. This represents a fundamental differing of opinion between the Java language designers and the C#/VB.NET language designers. In C# methods are final by default and developers must explicitly allow overriding (via the `virtual` keyword). In Java, methods are virtual by default and developers must explicitly disallow overriding (via the `final` keyword).

The virtual by default vs final by default debate is interesting, but outside the scope of this chapter. If you're interested in learning more, I suggest you read this interview with Anders Hejlsberg (C#'s Lead Architect), as well as these two blog posts by Eric Gunnerson. As well, check out Michael Feathers point of view.

Typically virtual methods are discussed with respect to inheritance of domain models. That is, a `KaraokeSong` which inherits from a `Song`, or a `Dog` which inherits from a `Pet`. That's a very important concept, but it's already well documented and well understood. Therefore, we'll examine virtual methods for a more technical purpose: proxies.

Proxy Domain Pattern

A proxy is something acting as something else. In legal terms, a proxy is someone given authority to vote or act on behalf of someone else. Such a proxy has the same rights and behaves pretty much like the person being proxied. In the hardware world, a proxy server sits between you and a server you're accessing. The proxy server transparently behaves just like the actual server, but with additional functionality - be it caching, logging or filtering. In software, the proxy design pattern is a class that behaves like another class. For example, if we were building a task tracking system, we might decide to use a proxy to transparently apply authorization on top of a task object:

```
public class Task
{
   public static Task FindById(int id)
   {
      return TaskRepository.Create().FindById(id);
   }

   public virtual void Delete()
   {
      TaskRepository.Create().Delete(this);
   }
}
public class TaskProxy : Task
{
   public override void Delete()
   {
      if (User.Current.CanDeleteTask())
      {
         base.Delete();
      }
      else
      {
         throw new PermissionException(...);
      }
   }
}
```

Thanks to polymorphism, `FindById` can return either a `Task` or a `TaskProxy`. The calling client doesn't have to know which was returned - it doesn't even have to know that a `TaskProxy` exists. It just programs against the `Task`'s public API.

Since a proxy is just a subclass that implements additional behavior, you might be wondering if a `Dog` is a proxy to a `Pet`. Proxies tend to implement more technical system functions (logging, caching, authorization, remoting, etc) in a transparent way. In other words, you wouldn't declare a variable as `TaskProxy` - but you'd likely declare a `Dog` variable. Because of this, a proxy wouldn't add members (since you aren't programming against its API), whereas a `Dog` might add a `Bark` method.

Interception

The reason we're exploring a more technical side of inheritance is because two of the tools we've looked at so far, RhinoMocks and NHibernate, make extensive use of proxies - even though you might not have noticed. RhinoMocks uses proxies to support its core record/playback functionality. NHibernate relies on proxies for its optional lazy-loading capabilities. We'll only look at NHibernate, since it's easier to understand what's going on behind the covers, but the same high level pattern applies to RhinoMocks.

(A side note about NHibernate. It's considered a frictionless or transparent O/R mapper because it doesn't require you to modify your domain classes in order to work. However, if you want to enable lazy loading, all members must be virtual. This is still considered frictionless/transparent since you aren't adding NHibernate specific elements to your classes - such as inheriting from an NHibernate base class or sprinkling NHibernate attributes everywhere.)

Using NHibernate there are two distinct opportunities to leverage lazy loading. The first, and most obvious, is when loading child collections. For example, you may not want to load all of a `Model's Upgrades` until they are actually needed. Here's what your mapping file might look like:

```
<class name="Model" table="Models">
   <id name="Id" column="Id" type="int">
      <generator class="native" />
   </id>
   ...
   <bag name="Upgrades" table="Upgrades" lazy="true" >
      <key column="ModelId" />
      <one-to-many class="Upgrade" />
   </bag>
</class>
```

By setting the `lazy` attribute to `true` on our `bag` element, we are telling NHibernate to lazily load the `Upgrades` collection. NHibernate can easily do this since it returns its own collection types (which all implement standard interfaces, such as IList, so to you, it's transparent).

The second, and far more interesting, usage of lazy loading is for individual domain objects. The general idea is that sometimes you'll want whole objects to be lazily initialized. Why? Well, say that a sale has just been made. Sales are associated with both a sales person and a car model:

```
Sale sale = new Sale();
sale.SalesPerson = session.Get<SalesPerson>(1);
sale.Model = session.Get<Model>(2);
sale.Price = 25000;
session.Save(sale);
```

Unfortunately, we've had to go to the database twice to load the appropriate `SalesPerson` and `Model` - even though we aren't really using them. The truth is all we need is their ID (since that's what gets inserted into our database), which we already have.

By creating a proxy, NHibernate lets us fully lazy-load an object for just this type of circumstance. The first thing to do is change our mapping and enable lazy loading of both `Models` and `SalesPeoples`:

```
<class name="Model" table="Models" lazy="true" proxy="Model">...</class>

<class name="SalesPerson" table="SalesPeople"
      lazy="true" proxy="SalesPerson ">...</class>
```

The `proxy` attribute tells NHibernate what type should be proxied. This will either be the actual class you are mapping to, or an interface implemented by the class. Since we are using the actual class as our proxy interface, we need to make sure all members are virtual - if we miss any, NHibernate will throw a helpful exception with a list of non-virtual methods. Now we're good to go:

```
Sale sale = new Sale();
sale.SalesPerson = session.Load <SalesPerson>(1);
sale.Model = session.Load<Model>(2);
sale.Price = 25000;
session.Save(sale);
```

Notice that we're using `Load` instead of `Get`. The difference between the two is that if you're retrieving a class that supports lazy loading, `Load` will get the proxy, while `Get` will get the actual object. With this code in place we're no longer hitting the database just to load IDs. Instead, calling `Session.Load<Model>(2)` returns a proxy - dynamically generated by NHibernate. The proxy will have an id of 2, since we supplied it the value, and all other properties will be uninitialized. Any call to another member of our proxy, such as `sale.Model.Name` will be transparently intercepted and the object will be just-in-time loaded from the database.

Just a note, NHibernate's lazy-load behavior can be hard to spot when debugging code in Visual Studio. That's because VS.NET's watch/local/tooltip actually inspects the object, causing the load to happen right away. The best way to examine what's going on is to add a couple breakpoints around your code and check out the database activity either through NHibernate's log, or SQL profiler.

Hopefully you can imagine how proxies are used by RhinoMocks for recording, replaying and verifying interactions. When you create a partial you're really creating a proxy to your actual object. This proxy intercepts all calls, and depending on which state you are, does its own thing. Of course, for this to work, you must either mock an interface, or a virtual members of a class.

In This Chapter

In chapter 6 we briefly covered NHibernate's lazy loading capabilities. In this chapter we expanded on that discussion by looking more deeply at the actual implementation. The use of proxies is common enough that you'll not only frequently run into them, but will also likely have good reason to implement some yourself. I still find myself impressed at the rich functionality provided by RhinoMock and NHibernate thanks to the proxy design pattern. Of course, everything hinges on you allowing them to override or insert their behavior over your classes. Hopefully this chapter will also make you think about which of your methods should and which shouldn't be virtual. I strongly recommend that you follow the links provided on the first page of this chapter to understand the pros and cons of virtual and final methods.

Wrapping It Up

> WE ALL TEND TO TIE OUR SELF-ESTEEM STRONGLY TO THE QUALITY OF THE PRODUCT WE PRODUCE - NOT THE *QUANTITY* OF PRODUCT, BUT THE *QUALITY*. - TOM DEMARCO & TIMOTHY LISTER (PEOPLEWARE)

For many, programming is a challenging and enjoyable job that pays the bills. However, given that you've managed to read through all of this, there's a good chance that, like me, programming is something much more important to you. It's a craft, and what you create means more to you than probably any non-programmer can understand. I take great pleasure and pride is building something that stands up to my level of quality, and learning from the parts that I know need to be improved.

It isn't easy to build quality software. A new features here or a misunderstanding there, and our hard work starts, ever so slightly, to show weakness. That's why it's important to have a truly solid understanding of the foundations of good software design. We managed to cover a lot of real implementation details, but at a high level, here are my core principles of good software engineering:

- The most flexible solution is the simplest. I've worked on projects with flexibility built into the system upfront. It's always been a disaster. At the core of this belief lie in YAGNI, DRY, KISS and explicitness.
- Coupling is unavoidable, but must be minimized. The more dependent a class is on another class, or a layer on another layer, the harder your code is to change. I strongly believe that the path to mastering low-coupling is via unit tests. Badly coupled code will be impossible to test and plainly obvious.
- The notion that developers shouldn't test has been the anti-silver bullet of our time. You are responsible (and hopefully accountable) for the code that you write, and *thinking* that a method or class does what it's supposed to isn't good enough. The pursuit of perfection should be good enough reason to write tests, but, there are even better reasons to do so. Testing helps you identify bad coupling. Testing helps you find awkwardness in your API. Testing helps you document behavior and expectations. Testing lets you make small and radical changes with far greater confidence and far less risk.
- It's possible to build successful software without being Agile - but it's far less likely and a lot less fun. My joys would be short-lived without ongoing customer collaboration. I would quit this field without iterative development. I would be living in a dream if I required signed off specifications before starting to develop.
- Question the status quo and always be on the lookout for alternatives. Spend a good amount of time learning. Learn different languages and different frameworks. Learning Ruby and Rails has made me a far better programmer. I can pinpoint the beginning of my journey in becoming a better programmer to a few years ago when I was deeply entrenched in the source code for an open source project, trying to make heads of tails of it.

My last piece of advice is not to get discouraged. I'm a slow learner, and the more I learn, the more I realize how little I know. I still don't understand half of what my peers on CodeBetter.com blog about (seriously). If you're willing to take the time and try it out, you will see the progress. Pick a simple project for yourself and spend a weekend building it using new tools and principles (just pick one or two at a time). Most importantly, if you aren't having fun, don't do it. And if you have the opportunity to learn from a mentor or a project, take it - even if you learn more from the mistakes than the successes.

I sincerely hope you found something valuable in here. If you wish, you can thank me by doing something nice for someone special you care about, something kind for a stranger, or something meaningful for our environment.

Remember to download the free Canvas Learning Application for a more hands-on look at the ideas and toolsets represented in this book.

