The Code Wiki

Learn. Share. Grow.

Karl Seguin

WWW.THECODEWIKI.COM

License

Updates & Evolution

The road for The Code Wiki isn't set in stone. I'm aiming for roughly eight to ten chapters, with a new chapter released each month. I realize that a chapter a month isn't much, but it's the reality of an amateur writer writing for fun rather than profit.

Ideally other authors will use The Code Wiki as the framework for their own articles. If you're an aspiring author who'd like to write about The Code Wiki, check out DotNetSlackers as a possible place to publish your articles.

If you want to stay up to date with new chapter releases then I suggest you check out my Code Wiki blog on CodeBetter.com and subscribe to its RSS feed.

Table of Contents

About the Author

Karl Seguin is a senior developer at Fuel Industries, a Microsoft ASP.NET 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 is located at: http://www.codebetter.com/

Arm Yourself to Succeed

1

```
void strcat(char* dest, char* src)
{
  while (*dest) dest++;
  while (*dest++ = *src++) ;
}
```

Some readers might recognize the above code as a possible implementation of C's string concatenation library function. Surprisingly, even though this function is over 30 years old, it doesn't suffer from the same performance issues as .NET's string concatenation.

Thanks to countless blog posts, most .NET developers know that if you need to concatenate a string more than a couple times, you should use a `StringBuilder` – it's much faster! That's because, say the blogs, concatenating a string in C# or VB.NET with the + or & operators is a costly operation. Strings are immutable (can't be changed), so concatenating with a `String` object always results in memory reallocation. `StringBuilder`'s, on the other hand, use an internal buffer so that reallocation isn't always necessary.

Like I said, the C version above doesn't suffer from *that* performance issue – `src` is simply appended to `dest` without any reallocation. Instead, the code suffers from a different performance issue; can you guess what it is? Since the length of `dest` is unknown, it must be traversed from start to end before anything can be appended. This quickly becomes problematic when dealing with long strings or multiple concatenations (think for-loop). In addition to the performance hit, there's also a security issue – there's no check to ensure that `dest` can even accommodate `src`. If it can't, an exploitable buffer overflow will occur.

Buffer overflows, which have been the root cause of numerous security vulnerabilities (such as the SQL Slammer worm), aren't something .NET developers really need to worry about. In order to make developers more productive, higher level languages, like C#, VB.NET and Java abstract much of the low level details away. As far as we're concerned, we really are dealing with a string – not a pointer to some buffer on the heap. This abstraction though, isn't perfect. To protect against buffer overflows, the `String` object is immutable, meaning string concatenation isn't as fast as it could be.

In some cases some developers are able to get away with being completely ignorant. After all, a powerful server under moderate load will laugh at a handful, or even hundreds, of string concatenations. In a lot of cases it's good enough to understand that given a particular situation (numerous concatenations) an alternative (`StringBuilder`) might be a better choice. But in your career as a programmer, you will run into a situation when it's necessary to have a fundamental understanding of how a programming language interacts with system resources. You could very well end up being a hero because you understand how the `StringBuilder` class works and in what situation it might be **slower** than string concatenations.

Introducing *The Code Wiki* Book

This book isn't about squeezing performance out of every line of code you write. (I do believe that developers with a good understanding of C (or similar languages) have a significant advantage over those who don't.) This book is about introducing and exploring alternatives to how you design, write and test your code on a day to day basis. The goals are simple:

1. To increase our arsenal of available tools,
2. To understand, in detail, each tool in order to quickly identify and customize the right one for a given situation, and
3. To become considerably better programmers by encouraging our natural curiosity and creativity.

You can't truly know whether to use string concatenation or a `StringBuilder` without knowing their inner workings (you can read some recommendations that say `StringBuilders` are better and *probably* be ok). Similarly, you can't build effective systems without knowing and understanding what alternatives are available for the current piece of the puzzle you are working on.

At a high level, the topics we'll be discussing include domain driven design, object oriented programming, unit testing and design patterns. This book is about building simple and maintainable applications. As far as I'm concerned *Simple* and *Maintainable* are synonyms – a topic we'll discuss shortly.

If you came from programming windows form in VB6 or websites in classic ASP, .NET had a severe learning curve. We all bought various reference books, which explained what a `Textbox` was (quite a new concept for web programmers), and proceeded to *learn* .NET. We learnt about `DataAdapters` and `DataSets`, bound controls and how to deal with exceptions using try/catch. Once we reached the end of our massive books, we put them down and called ourselves .NET programmers. But knowing about the classes that make up the `System.*` namespace and understanding the ASP.NET page lifecycle is only the first step in a much longer learning process. .NET made it possible to build multi-tiered applications, leverage object oriented programming, take advantage of design patterns and introduce unit testing to our day-to-day programming. Sadly, few developers continued their education past the basics needed to get something running on the screen (I personally procrastinated for a year or so before taking the initiative). The result tends to be low quality code.

It's possible that I've had a unique experience, but from what I've seen many developers don't even know that their code isn't very good. Their .NET code looks frighteningly a lot like classic ASP or VB6 windows forms. Most developers are eager to learn though (which can't be as easily said for other programming communities) and only lack the opportunity and guidance to do so.

The Code Wiki is meant as the next step in the learning process.

Introducing *The Code Wiki* Application

The Code Wiki application was built specifically for the purpose of this book. I highly recommended that you familiarize yourself a bit with the application before getting too deep into this book (you can find it

at http://www.thecodewiki.com/code.aspx). It's a good idea to have The Code Wiki application open while you read so that you can follow along and explore the code at will when something catches your eye.

The Code Wiki is a web application used to display a solution's code much like Visual Studio (tree navigation on the side and a color-coded source code viewer). In addition, users can leave comments about the source code in the hopes of offering advice or alternative approaches (newsgroups and forums are better suited to ask questions). The Code Wiki is bootstrapped – meaning when you go to the Code Wiki website, the code you'll be navigating is the actual Code Wiki code.

The Code Wiki isn't a perfect application (or very unique – I've seen somewhat similar implementations before). Sometimes I chose a solution because it demonstrated something I thought was worthwhile, rather than because I thought it was the best choice. At times, balancing features, design and learnability was a challenge. There's some really good functionality that could be added to The Code Wiki which would make it a much better application. To fulfill its purpose as a learning tool, The Code Wiki is simple and straightforward.

Total Cost of Ownership

Sometimes I'm amazed at how many lines of code went into making The Code Wiki. If you are new to domain models and unit testing, you might very well think that I'm some crazy purist preaching from an ivory tower. If I had to build The Code Wiki for a client, there are some things, depending on the exact requirements, that I'd likely change. The entire thing could probably be built with a couple ASPX pages, their CodeBehind and a handful of classes. This is certainly highlighted by the fact that I intentionally picked a simple application to explore advanced programming topics.

At Fuel Industries our team is responsible for all types of applications. In some cases we build simple web applications or Active X controls which don't take more than a couple of days of work. These applications are expected to have a short lifespan and thus require little maintenance. For such projects our goal is to get a functional system done as quickly as possible and forget about it. However, some of our projects are much more complex, involving months of work and testing and years of support and maintenance. It shouldn't surprise anyone that our mentality for these projects is quite different. From experience, we know that long-lived projects will cost a lot more to maintain than to initially develop. Therefore, it only makes sense to spend time and money upfront building a robust and flexible system so that when fixes, additions or changes are needed our costs are minimized. I've read studies that suggest post-release support and maintenance accounts for 70% to 90% of the total cost of ownership (and it's a growing trend as years go by).

In my experience, clients, managers and developers always agree with the concept of spending a dollar now to save ten later. In fact they wholeheartedly support it and want their own project to be driven by similar thinking. Somehow, it rarely happens. I can't intelligently speak for clients or managers (though I can rant about it at length), but from what I've been exposed to, many developers just don't know how to make it happen. I haven't seen any data to support this, but I'd venture to guess that code quality over the last three decades has steadily declined – the only thing keeping everything together are better tools, more powerful hardware and higher levels of abstraction (which can be dangerous).

This is especially troubling when you consider that programs are growing in complexity, more and more systems are being developed and the job market is starting to heat up – meaning most employers can't be as picky as they should be.

Every project will have its own unique priorities. Some must be completed tomorrow and some have special performance requirements. As a developer, it's generally easy to identify the one or two driving forces behind a project. A description of the job, knowledge of the client, a sense for the budget or a timeframe is generally all you need to get an idea. Remember, the goal of this book isn't tell you how to build systems (you already know how to do that). The goal is to provide alternatives that have their pros and cons and enable you to wisely pick the right one.

Most of what you'll learn from The Code Wiki is aimed at making a system more maintainable – possibly increasing the initial cost but decreasing the total cost of ownership. Both .NET and Java are heavily used in enterprise development, where maintainability is always a key factor. Chances are you're a developer working on a medium to large system with a two-or-more year shelf-life. MAINTAINABILITY!!

What Makes A System Maintainable?

Maintainability isn't the only factor you must consider as a developer, but hopefully you agree that it is something you must actively pursue. When I say maintainability, I don't only mean fixing bugs and adding features after you've released your program. To me, maintainability has a much broader meaning which covers the complete development lifecycle, including planning and development. After all, the things you do to make your system easier to maintain after release are likely to make your life easier during development.

Keep it Simple

Not too long ago, I worked on a project which no one would use the word *simple* to describe (the project ended up being a failure). It proved to be one of the most invaluable learning experiences of my career. Because of it, simplicity became the key concept guiding any piece of code I wrote. It became the mantra by which I programmed.

The system itself was a mission critical enterprise application for an organization in the financial sector (I know, I know everyone likes to say they work on mission critical systems, and unless people's lives actually depend on it, it's just an exaggeration). It had to solve the problems of a complex domain, had various clients and stakeholders, and tight deadlines. To boot, the project was over-funded. The system was replacing an old legacy system which had done its job admirably for 15 years – it was hoped that the new system would also have a long lifespan. Instead, the system was scrapped after 4 years of development (2 of which were also spent in production) and a new replacement built.

The approach the two lead "architects" took was to build a very flexible and powerful framework on which the system would sit. Unfortunately, it didn't work the way the client wanted it to (the separate pieces of the system couldn't speak to each other), it suffered horrible performance issues and finding developers to maintain it (after the two leads left) proved to be a challenge. The core of the system was an abstract meta repository which could be used to hold and manipulate data as needed. Rather than having a `Users` table in your database, think of having an `Entities` table which contains a row with the value "User" and a foreign key to a separate `EntityData` table containing a row for each value of your user (that's a gross exaggeration). In theory the system was infinitely flexible, in practice it ran out of memory every 15 minutes. I once went 14 layers deep, through 2 reflection calls, trying to figure out how an object in CodeBehind was getting populated.

The architects knew that they had limited time to build a system which would need to be improved upon and fixed for a long period of time. They thought they could build a system upfront for features to come in the following years. Their hubris killed the project.

It ends up that, in most cases, the right approach to such a situation is the exact opposite: build the simplest system possible that addresses the current needs. It's easy to change a simple solution and easy to find developers to maintain it. If there's a bug it takes hours, not weeks to fix it. Code is easier to debug, profile and adapt. A serious bug at the core of a complex system, such as the memory issues we had, can end up taking months to fix. It's so logical it's almost silly to have to say it.

One consequence which developers get hung up on is the frequent need to change and rebuild the code. For some reason, recompiling code has a worse negative connotation association to it than it should. Instead they'd rather build XML-based rules engines and dumb meta-repositories. To be sure, both those solutions have their uses, but even though changing a configuration file, a table schema or a stored procedure doesn't require a recompile, they are just as likely to introduce major bugs. Developers shouldn't be afraid to modify their code and recompile it. Unit testing and refactoring tools, which this book will cover, can help a developer be more confident with the changes he or she is making.

The advantages of a simple solution should be obvious to anyone. How do we go about writing simple code though? The Extreme Programming methodology has a programming practice I hold dear and true: *You Aren't Gonna Need It* (YAGNI), which states:

> Always implement things when you actually need them, never when you just foresee that you need them.

The logic being that you probably won't actually need the feature (even if you were sure you would) or it won't be like you thought it would. The result tends to be a lot of saved time and much cleaner code. Of course, YAGNI is just a general rule of thumb. You'll know best what does and doesn't have to be implemented, but following YAGNI can help keep a developer on target.

Once I decide that I really do need to write the code, I find explicit solutions to be the simplest. For example, look at the difference between retrieving a `userId` from a `DataRow` and an instance of a `User` class:

```
//DataTable
dim userId as integer = cint(dr("UserId"))

//User class
dim userId as integer = user.UserId
```

It's true that in both cases it's pretty clear what's going on (this is a simplified example after all). In a large system, there's little doubt which approach I rather deal with. Martin Fowler eloquently explained the need for explicit design in his short essay To Be Explicit, he says:

> This drive toward changeability is why it's so important for a design to clearly show what the program does—and how it does it. After all, it's hard to change something when you can't see what it does. An interesting corollary of this is that people often use specific designs because they are easy to change, but when they make the program difficult to understand, the effect is the reverse of what was intended.

He concludes by stating that sometimes a clever design is worthwhile, but that it's a programmer's job to carefully weigh the cost of such a design. You should diligently work to ensure your code is as simple, explicit and as straightforward as possible.

Use Proven Techniques

Keeping code as simple as possible might be good advice, but there are specific things we can look at to achieve greater maintainability. Specifically, two of the fundamental aspects of The Code Wiki, which we'll discuss at length, were picked because they substantially improve the ease with which code can be maintained. Both of these "architectural" pieces have been around for decades and have significantly improved software development as a whole. They are, domain driven design and N-Tier design – two terms you've likely heard before as well as patterns you've likely employed.

Domain Driven Design

By domain driven design we mean that the domain will be the driving force behind our entire design. The *domain* is whatever business you're system will be part of (it's also called the problem-domain). For example, your domain might be a financial institute, a hospital or a video game (we'll give more concrete examples briefly). Our chief tool to build domains is object oriented programming (OOP) which is particularly well suited for the task (I'd even say it was developed specifically for it, but that would be an assumption). Knowing that the building blocks of object oriented programming are Classes, let's consider what the domain for a real estate system might look like. At a very high level, we'd likely have a `Property` class, an `Agent` class and a `Listing` class. Each class would then be composed of properties which represent the data associated with an entity (such as the `Value` of a `Property`), and methods which represents abilities of an entity (such as an `Agent` can `Create` new `Listings`).

Notice that we've used nouns that people who would use a real estate system are likely to understand. You might not know the difference between a `Property` and a `Listing`, but to an agent they are core parts of his or her business (if our system will have a public-facing section, we can hide the concept of a `Listing` and simply deal with `Properties`). It's a small example, but it does highlight how it can help make our code more readable and easier to maintain – there's less likely to be confusion when a client (who's likely a domain expert, and knows nothing of programming) tells you he needs the system to let `Boards` create `Listings` on behalf of their `Agents`. Other benefits of OOP and domain driven design is greater re-use, high testability and an inherit flexibility.

Some developers take domain driven design rather literally and insist that the domain layer must be built first. I prefer to see domain driven design as an objective to build our code in the image of the domain – where and how you start isn't nearly as important. In the next chapter we'll cover this point in more detail when we talk about the interaction between our database and our domain layer.

N-Tier

There's a lot of overlap between N-Tier and domain driven design. That's because most of the domain logic we talked about above (classes, properties and methods) makes up one of the tiers in most N-Tier designs. N-Tier refers to the functional layering of your code, where N refers to the number of actual layers used. A traditional 3-Tier system is composed of a presentation layer (UI), domain layer and database layer. It must be noted that I use the terms *layer* and *tier* interchangeably (although some use *tier* specifically to describe *where* code runs, such as a 2-Tier mainframe/dumb-terminal system which has much more serious performance and design implications).

The Code Wiki is based on a standard 3-Tier system, although, like most 3-Tier system, it also makes use of two additional layers (which I like to call sub-tiers): the Data Access Layer (DAL) and the presentation logic layer. I find layering so crucial to proper code organization that much of this book is divided along the same layers (chapter 3 is all about the domain layer, chapter 4 about the database layer and so on).

A large number of ASP.NET and WinForm applications I've seen would best be characterized as using a 2-Tier architecture. Despite the importance I've already attributed to the domain layer, most of the applications I've seen chiefly rely on the presentation layer and the data layer. In other words, they are void of any domain layer. I see this as one of the most critical issues facing software development today (at least with respect to the kind of software most of us build). What's worse is that the problem is particularly serious amongst .NET developers. If you consider the lack of tools to build proper domain layers in classic-ASP or VB6, the situation is rather expected. As developers moved to C# and VB.NET, they learnt the new technology but not the new methodology to go along with it.

(I have limited experience with the Java community but, as I would have expected, they seem to have grasped these two concepts much better. Most Java developers come from other object oriented languages, such as C++ and Smalltalk, and a number of them were born directly into the language. The Java community is facing other, equally serious, issues.)

Layering is a great way to help organize your code; it also forces you to write interfaces which provide a different view into your system. Another benefit I'm really fond of is the robustness offered by layered code. With a proper implementation, changes made in a lower layer can often be hidden or abstracted from higher layers minimizing the amount of code that needs to be updated.

You'll often hear that the best part about N-Tier architecture is the ease with which layers can be reused. For example, a properly designed domain layer can service a web application, a windows application and web services. I didn't use to think much of that argument. Most of the time you know which database you'll be using and whether it'll be a web-based application or a windows application. With the ever-increasing popularity of web-services, my opinion has changed a bit. Also, if you're a 3rd party vendor, enabling your clients to easily integrate with the database of their choice is bound to be a key feature.

Understanding Cohesion and Coupling

You might have heard that a good design is one which has high cohesion and low coupling. It's even been said that the main purpose of software design is to achieve high cohesion and low coupling. One reason I find the concept of cohesion and coupling important is that it applies at all levels of design. Individual objects should strive to be very cohesive with little coupling as should the layers within your system. Coupling is the level of interdependency between like-things, such as classes or layers. An object with high coupling has many dependencies on other objects. Coupling makes code harder to change and maintain, regardless of whether it applies to objects or layers. If class A is dependent on class B, then changes to B might break A. On the other side of the coin, cohesion measures the functional uniformity offered by a class or layer. A highly cohesive object is easy to understand because it fulfills a single, well defined task (at a high level). When talking about cohesion, it's important to stay at a relatively high level. An `Agent` class might have methods to `Update` and `Delete` the `Agent`. These might be two separate tasks, but the `Agent` class itself has a single task: to model an `Agent` of the system.

It's impossible to build a system with no coupling. The `Agent` class must have knowledge and interact with the `Board` class. Coupling can be minimized by ensuring interaction only happens with a well defined public API, class A needn't know about class B's internals, nor does the presentation layer need to know anything about the internals of the business layer.

If two objects have high coupling, it could be an indication that they share the same task and should be merged one way or another. Conversely, if a class has poor cohesion, that is its purpose isn't clear, there might be opportunities to break the class up into multiple classes or to subclass part of it.

Didn't I Say Something About Simple?

It can be hard to imagine how creating 5 layers of code along with a high fidelity representation of the problem-domain can be consistent with our goal of *keeping it simple*. There's a chance you've looked at part of The Code Wiki application and shaken your head in disgust with the amount of code it took to make it happen. I've already admitted that some design decisions were taken for the purpose of teaching/learning, but the truth is that's a cop-out.

Keeping it simple isn't about writing the fewest lines of code. Sometimes the simplest solution *is* the one that requires the fewest lines of code, but that's more often true of specific algorithms, not overreaching architectural groundwork. Learning about design driven domain, unit testing and cohesion might not be the easiest route to take, but the cost of learning something should only be an issue if you need an immediate solution (if that's the case, what are you doing reading this?) or what you learn has little value once the project finishes. If you insist on taking the cost of learning into account, then the simplest solution is to forget everything you know about programming and do something else, like joining the sales department.

It's also crucial to understand that The Code Wiki introduces a lot of different approaches and few of them are absolutely tied to each other. Maybe you really like the benefits of layering, but think typed-datasets would be well suited for your domain and data access layer. Maybe you can't benefit from

custom server controls, but can take advantage of `HttpHandlers`. It's up to you to decide which tools are best suited for the job.

Be Vigilant About the Little Things

I can't stress how important it is to constantly stay on top of all the little things. The first thing that comes to mind is code readability – it can make or break the maintainability of a system, regardless of everything else. A naming convention must be established and enforced (whatever the size of the team, even if it's 1). I've been using Microsoft's Design Guidelines since the release of .NET and think they are phenomenal. Use meaningful function names and make sure your functions don't do more than their names say they will.

Comment your code appropriately, but not too much. I personally find The Code Wiki has too many comments and that readability is adversely affected by it (especially since there's a bug with the indentation of collapsed comments). If your code is so complicated that you need to comment every other line, you have serious problems on your hands. On the other hand, if you're building a public API for someone who'll be blind to your implementations, XML documentation is key.

Never swallow exceptions. Be mindful of cleaning up after yourself (not talking about `IDisposable` although that's good advice too). Use `//todo` comments. Don't be afraid to create a new class instead of polluting an existing one. Don't be afraid to get your hands dirty. Don't just say you need to go back and clean it up, do it!

C# vs. VB.NET

You might have noticed that The Code Wiki is written in C# - no VB.NET version exists. However, this book uses a mix of C# and VB.NET – except for when I'm referencing a code example from the application (which is admittedly often).

Every .NET developer should be comfortable in both C# and VB.NET. If you actually understand what you are doing, learning *the other* language is a trivial thing – just use it next time you have a little app to build. Developers who only know one or the other are hurting themselves in the long run. If you only know C# and you refuse to learn VB.NET, or only know VB.NET and refuse to learn C#, this book isn't for you.

I'm glad to see the debate between C# and VB.NET has mostly died down. I've said it before, and I'll say it again, a good programmer will write good code in either language. A bad programmer will write bad code in either language.

Further Reading

The following are links to resources relevant to the topics we've covered in this chapter. I strongly encourage you to thoroughly read each one.

In the first part we hinted at Joel Spolsky's Law of Leaky Abstraction – the string class is an abstraction of the character pointer in C. The concept of leaky abstraction is neat, the implications are real. It's a well written piece about the danger of blindly using abstractions:
http://www.joelonsoftware.com/articles/LeakyAbstractions.html

Since strings are such a big part of your every day programming, it's a good idea to go beyond the abstraction and build a fundamental understanding of what's really going on. Again from Joel Spolsky with the basics of strings:
http://www.joelonsoftware.com/articles/fog0000000319.html

I hinted that in some situations using a `StringBuilder` is actually slower than plain-old string concatenation. I figured it wasn't fair to leave it at that, so here's a link that explains when and why:
http://geekswithblogs.net/johnsperfblog/archive/2005/05/27/40777.aspx

Martin Fowler has written a lot of great articles and books, but his short essay To Be Explicit has always struck me as particularly well done (maybe because I agree with it so much).
http://www.martinfowler.com/ieeeSoftware/explicit.pdf

I don't necessary recommend that you run out and buy it, but The C Programming Language Book is, and will remain, my favorite programming book of all time. Even though I no longer write C, I still read it every couple years as a refresher on what's really going on when I use high level things like constructors and delegates. As an amateur author, I also think that the book is stupidly well written and puts to shame almost all other language/reference books (Paul Vick's VB.NET book is actually very high on my list as well).
http://cm.bell-labs.com/cm/cs/cbook/

The View from 30 000 Feet Above

2

We begin our journey by building an understanding of how the system is implemented at a high level. Future chapters will zoom into specific layers, classes and even statements which make up The Code Wiki; but without a broad understanding of the architecture and the thoughts behind it, this book has little value.

Domain. Domain. Domain.

I previously mentioned that the lack of domain layers within .NET applications is a critical issue facing software producers and purchasers. To compensate, these applications overburden their presentation and data layers. The result is that business logic is stuffed into the presentation layer, making it hard to reuse, or into stored procedures, making it hard to write and maintain. Typically, both approaches are used, giving us the worst of both worlds along with pointless overhead.

The problem can be pinpointed to our roots as ASP or VB developers and the lack of tools available to do it any better. We've learnt to live with code like what you see below (an actual example I pulled out from an archive):

```
<%
rsNews = conn.execute("SELECT Title, Date FROM News")
while not rsNews.eof
%>
    <tr>
      <td><%=rsNews(1)%></td>
      <td><%=formatdatetime(rsNews(2), 1)%></td>
    </tr>
<%
   rsNews.movenext()
wend
%>
```

We knew it wasn't right, and hated the tightness between HTML and code, but that was how the technology had to be used. There was a single layer, and it combined our presentation, business and data access code (VB6 developers had it marginally better).

When ASP.NET came out we were gleeful with the separation provided by CodeBehind and bound controls. With little effort, our code suddenly became much easier to maintain. In addition to a cleaner UI, a growing number of developers started using stored procedures exclusively and even started using DAL (Data Access Layer) classes in order to further increase code reusability.

In reality, this new and improved layering was highly cosmetic (which isn't to say it wasn't sorely needed). Business logic was still stuffed in the presentation layer (countless CodeBehind files) or the data layer (stored procedures). Few developers, and fewer books, decided to look at what other techniques were possible with .NET.

CodeBehind as the Domain Layer

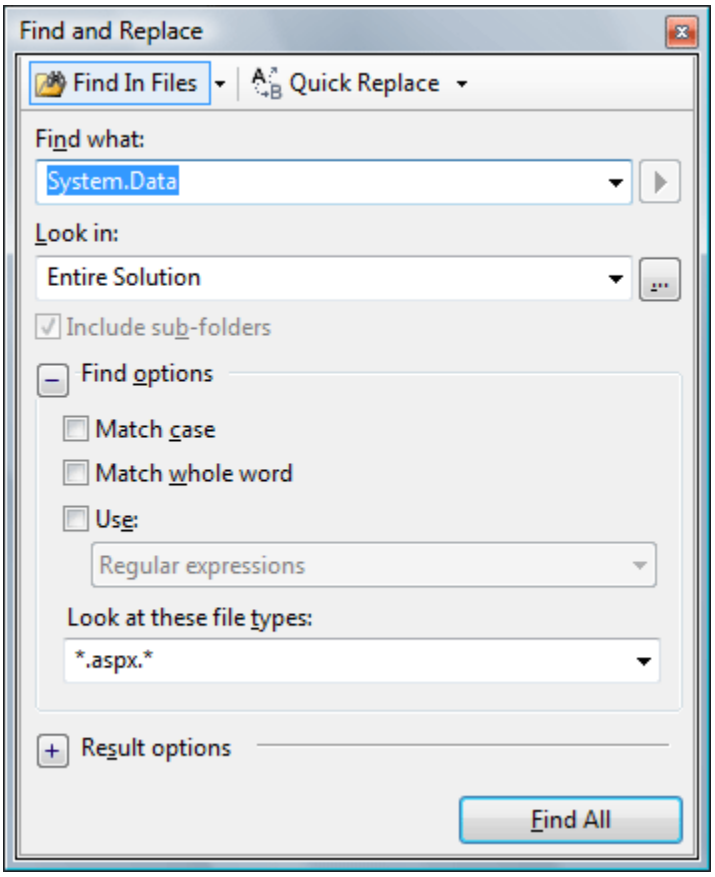
CodeBehind is a decent tool for separating the raw presentation layer with the code responsible for manipulating it. ASP.NET's databinding capabilities along with its rich event driven model make this separation possible. Whenever I hook into the `Repeater`'s `OnItemDataBound` event I'm reminded of how clean our HTML can be without sacrificing any functionality. However, there are many reasons why CodeBehind makes a lousy place to store your domain logic.

For the most part, CodeBehind files are tied to a specific ASP.NET page, making it unnatural for other pages or controls to reuse their logic. Another issue is that CodeBehind files are ASP.NET specific, making it impossible for another presentation layer, say a web service, to reuse them. In order to be reusable, your domain layer cannot be aware of any classes within the `System.Web.UI` or `System.Windows` namespaces (amongst others). Finally, it's common to see complex CodeBehind files which have a lot of presentation logic, especially with the increased use of Ajax. Burdening these pages with business and data access code significantly decreases their cohesiveness and readability. It's also worth noting that CodeBehind files aren't generally structured or even written in the same language that your business experts understand. Each page generally consists of performing a small number of actions, like `Update an Agent`. Your business experts simply see `Update an Agent` as a function of a `Board`, they don't care about the needs for a more granular structure and they certainly don't care that a `Textbox` is used to capture an `Agent`'s `Name`.

Appendix A explains ASP.NET's page implementation in more detail.

It's easy to spot code that abuses CodeBehind files – open a solution, go to `Edit`→ `Find and Replace` → `Find in Files`, enter `System.Data` in the `Find what` textbox and `*.aspx.*` in the `Look at these file types` textbox under `Find options`. This will search all CodeBehind files, whether they are written in C# or VB.NET, for any data access code. You can also change the file type filter to `*.ascx.*` to perform the same search on user controls. I frequently find such code in event handlers or around data binding operations. Like The Code Wiki, your solution should have zero matching results.

(This quick search doesn't catch the use of the new `XXXDataSource` controls which shipped with ASP.NET 2.0. Aside from the `ObjectDataSource` control, I don't think much of these controls for anything but the simplest project.)



Stored Procedures aren't the Business Layer

A few sections from here, we'll take a very detailed look at the data layer as a whole and it's interaction with the business layer. For now, it's important that you understand that, like CodeBehind, stored procedures are also a poor location in which to implement your business logic. The main reason is that SQL is a specialized language for data access and manipulation. TSQL is well suited to handle basic rules, but lacks the power for any meaningful implementations (there are a lot of limitations; the lack of OO is a significant one). SQL's high level of specialization is such a major constraint that all other issues aren't even worth discussing.

The Business Layer Revealed

We've managed to cover a lot of ground and make numerous references to the business layer without going into any details about what it is and why it's so important. We've already mentioned that the business layer's goal is to represent the problem domain with a high level of accuracy and we've given a very simple example based on a real estate system. Now it's time to start looking at it in greater detail.

Whenever I start a new project, the hardest part is always learning the business. Try as you might (and I have), it's almost impossible to build an application for someone without a good grasp of their business. It's an unavoidable nuisance - one which is so necessary to our success that we might as well try turning it into an advantage. A major benefit of a rich business layer is to help bridge the gap between the business and the system you are building. It might seem trivial, but establishing a *common vocabulary* between business experts and developers, and between members of the development team itself, ends up saving a lot of time. It's particularly useful when new developers join the team and are required to quickly get up to speed. There's much less of a disconnect between what the client thought she asked for and what you've actually delivered.

A well designed business layer also has the benefit of being unencumbered with either the presentation or the data layer (to a certain point). Not only does this result in a clean layer specifically meant to model the problem-domain, but it also lets the presentation layer and data layer focus on their own respective roles. Problem domains, such as our real estate example, are composed of entities like *agents* and *listings*. Normally these are noun-words that are an integral part of the client's business. Each entity will typically correspond to a class in your code with a matching name, like `Agent` and `Listing`. These entities have a state, such as *names* and *address* which are represented by field and property members (again, named rather appropriately). Entities also have behavior which map to methods and relationship to other entities. One of the most important parts is business rules. Business rules vary wildly in complexity. A simple business rule might be that all `Agents` must have a `Name`. A slightly more complex business rule might be that when a `Properties` with multiple `Agents` is `Sold`, all `Commissions` must be provided, otherwise a default

> *Capturing and coding business rules might not be as glamorous as creating AJAX-driven UIs, but they are the raison d'être of most programs, especially enterprise systems. Programming has gotten so easy that anyone can quickly stick a cool UI atop a database and have something running. Without proper planning and execution though, the result will always be superficial.*

`Commission` is used. You might have noticed that we just created an incredibly basic specification. Given what we know, here's what our real estate system's business layer might look like:

```
'Agent.vb
Public Class Agent
 Private _name As String

 Public Property Name() As String
  Get
   Return _name
  End Get
  Set(ByVal value As String)
   If String.IsNullOrEmpty(value) Then
    Throw New ArgumentNullException("Name")
   End If
   _name = value
  End Set
 End Property
End Class

//Property.cs
public class Property
{
 private List<Agent> _agents;
 public void Sell(int closingAmount, params int[] commissions)
 {
  if (_agents.Count > 1 && _agents.Count != commissions.Length)
  {
   throw new ArgumentException("Commissions must be provided");
  }
  //todo save
 }
}
```

Since the business layer knows nothing of the presentation engine (ASP.NET, WinForms, Web Service) or the database engine (SQL Server, XML, MySQL) it happens to be the easiest layer to test. Since it contains all of the business rules, it happens to be the most important to test as well. Skilled developers are always on the lookout for areas which provide the most bang for the buck whatever they are trying to do (performance, usability, maintainability). Aside from actually building a business layer, nothing increases system quality more than unit testing it. We'll discuss unit testing at length in later chapters, but it's worth mentioning that unit testing also improves code maintainability.

The business layer provides a common vocabulary between client and developer, a toolset specifically designed to model the problem-domain (most notably business rules) and a highly testable layer. It also provides other benefits commonly associated with layering, such as abstraction of the layers below and the potential for a well-defined API. In our field, few things provide as much benefits as rich domain layers.

Domain vs. Data Model

A very pertinent issue to what we've covered so far is something known as the object-relational impedance mismatch. The object-relational impedance mismatch is the result of using two differing technologies within the same system: object oriented tools as the foundation of our business layer and relational databases as the basis of our data layer. For simple systems, the mapping between the two models is relatively straightforward. Even so, the mismatch is always present. For example, I'd consider The Code Wiki to be a simple system, yet the two technologies don't always line up perfectly. The problem is much more serious in larger systems.

A common cause of conflicts is how the two technologies deal with relationships. In the relational world, relationships are simply foreign keys which can be used to explore a relationship in both directions. For example, the relationship between a `Property` and its `Agent`, is expressed as an `AgentId` in the `Property` table. The `AgentId` column can be used to get a `Property`'s `Agent`, or all of the `Agent`'s `Properties`. In the object world, the `Property` class has an `Agent` field which holds a reference to the appropriate `Agent` (if you think about it, in and of itself that's quite different). This doesn't let us get all of an `Agent`'s `Properties`, thus we need additional logic to get that information. Object oriented programming also lets us leverage inheritance, composition, encapsulation and interfaces for which no equivalent exists in the relational world. When we delve into the domain and data layers in following chapters will expose the specific conflicts I ran into when coding The Code Wiki.

The mismatch isn't an insurmountable problem as there are tools and techniques to deal with it. As a matter of fact, the mismatch is something of a blessing – allowing us to leverage the two technologies to their fullest extent. All we really need is a layer in between which understands each model and is capable of going back and forth between them. That's exactly what the goal of our Data Access Layer is – a go-between for our business and data layers.

> *The mismatch isn't caused by the business layer. If you stick with a 2-layered system, you'll still have to deal with molding your data to fit your UI and then transforming it back to your database. The only way to avoid it is to tell your client to only use Excel.*

Since The Code Wiki is simple, I decided to manually write the code to handle this mapping (I also think it keeps things easier from a learning perspective and is the best way to introduce the topic). However, you should be aware that there are tools, called O/R (object-relational) mappers, specifically designed to assist you. Additionally, the next version of .NET will bake a type of O/R mapping directly into the framework with LINQ.

Up until now, I've managed to avoid talking about `DataSets`. You could think of `DataSets` and `DataAdapters` as a form or O/R mapping, although in reality they are just an in-memory relational database. Typed-`DataSets` provide a thin decoration atop of this relational model. `DataSets` are so easy to use and feature-rich it's tempting to fully rely on them. While they certainly have their use, they simply aren't appropriate for enterprise systems for various reasons, including performance and maintainability.

The real problem you're likely to face with the object-relational impedance mismatch is cultural, not technological. Some developers are real OO-bigots and think of the database as little more than an inconvenient storage facility. Many developers, especially in the Microsoft development area, tend to heavily favor the relational world and don't really understand OOP. As you might expect, it's common to find DBAs who think views, indexes and stored procedures are the only way to solve a problem. The only real solution is for all parties involved (developers, DBAs and managers) to have a decent understanding of each technology.

The Code Wiki Overview

Even if you haven't visited The Code Wiki yet, you're probably starting to have a good understanding of how it's coded. However, there is one final step we must take before getting any more involved with the code: going over what The Code Wiki is and how it works.

Remember that the goal of The Code Wiki is to make available a solutions source code much like Visual Studio does, in addition to providing comments and basic versioning capabilities. To achieve this, The Code Wiki only needs a handful of core business objects (or entities), of which the `Resource` class is the most vital. A `Resource` represents an object that must be represented by The Code Wiki application. It can be a container, like a project or a folder, a source code file or a binary file. As such, a `Resource` always has a `Name`, a `Type` and a `Parent Resource` (which is null in the case of the root `Resource`). In the case of containers, there's also a list of `Children` resources.

A special type of `Resource`, called a `Document`, also exists. A `Document` extends a `Resource`, by having a `VersionNumber`, a `Comment` explaining the version, a list of alternative `Versions` and a list of user-contributed `Comments`. Of course, the most important part of a `Document` is its `Content`.

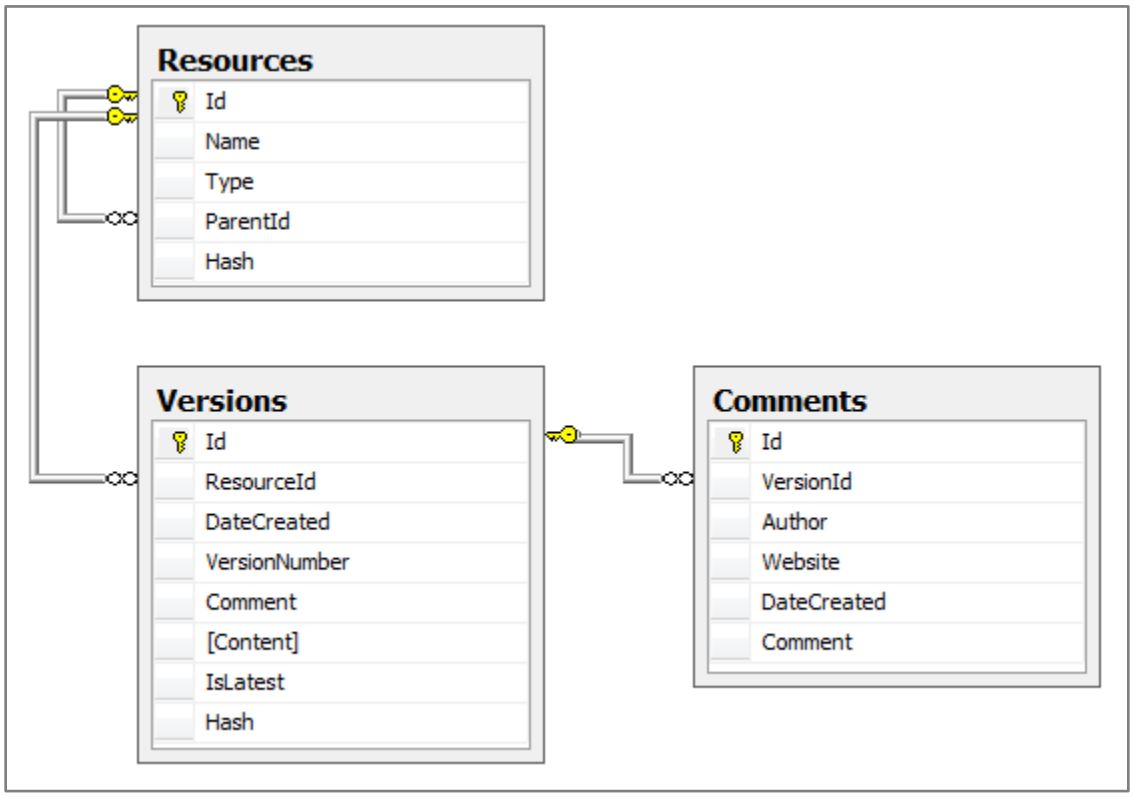
The last piece of the puzzle is the version `Hash`, which is used to detect new or modified `Resources`. We aren't quite ready to delve into the versioning system, so all you really need to understand is that the `Resource`'s hash is based on its file system path and the `Document`'s hash on its `Content`.

Combined, `Resources`, `Documents`, `Versions` and `Comments` make up the engine powering The Code Wiki.

Data Model

Despite the simple picture we've painted above, the data and domain model differ a little. It probably would have been possible to eliminate these differences, but only by sacrificing either sound relational design or sound object oriented design.

If we look at the data model, it's simple to spot the differences between what we described above and how we've actually implemented it in the relational world:

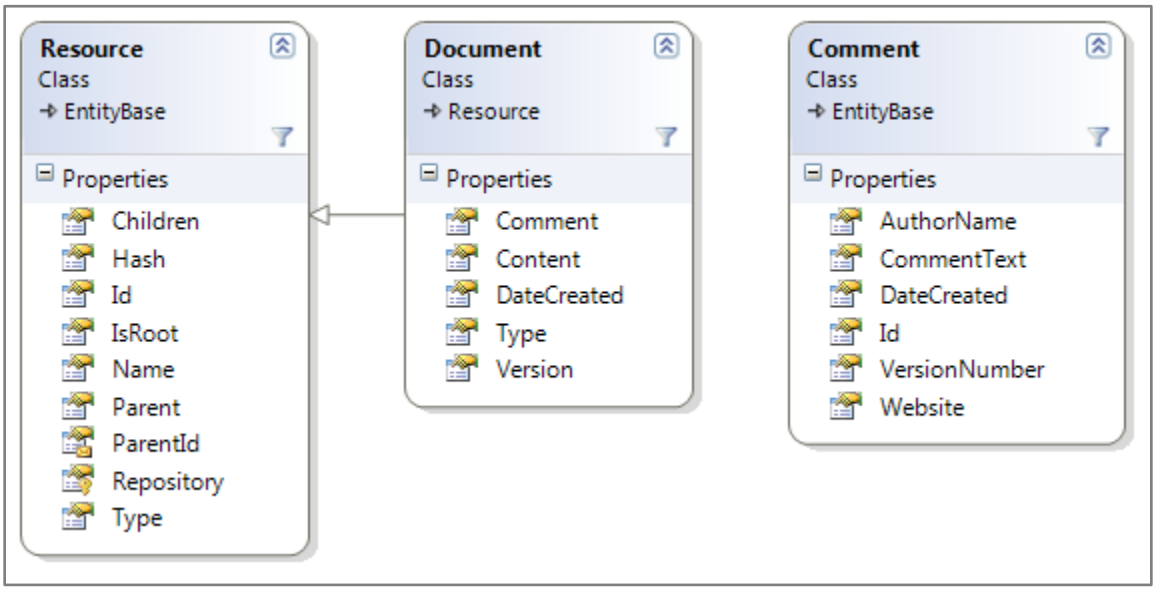


Where's our `Document` entity? It turns out that when I was designing the database, I realized that it wasn't at all necessary. A `Document` might be a concept that's very real to our client, but in a pure data world it's nothing more than a relationship between a `Resource`, `Version` and `Comments`. Initially, I tried to force a `Documents` table into the model, but I ended up with pointless horizontal partitioning.

From the above diagram, a binary or container `Resource` (identified by the value in the `Type` column), wouldn't have corresponding rows in the `Version` or `Comments` table (because these aren't being versioned/commented on). However, for source code `Resources`, one or more `Version` could exist and each `Version` could have one or more `Comments`.

Domain Model

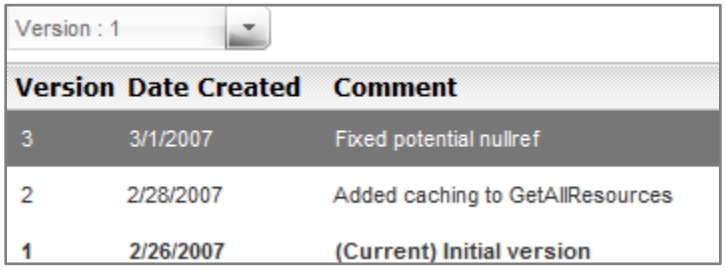
If we now compare the above data model to our domain model, we see subtle but important differences (the object-relational impedance mismatch is typically quite subtle):



The parent/child relationship of a `Resource` is now handled via references, rather than foreign keys. Notice that both a reference to `Parent` and the `ParentId` is provided, which enables different type of usage. The change in name from `Version` to `Document` seems awfully cosmetic, but I see them differently with respect to each layer – with respect to the data layer, they are simply versions of a given Resource; however, in our domain layer, they are first class citizens. I admit that I might have just created an artificial impedance-mismatch. The most striking difference is the implementation of a `Document` – it inherits from the `Resource` class, which is quite different from the foreign key approach used in the data model.

The `Comment` property of the `Document` class is the comment relating to that specific version of the `Document`. This brings up an obvious question: where's the link between a `Document` and its user-contributed `Comments`? The `Document` class has a method to lazily-load its `Comments`. (Methods aren't shown above, but their existence alone is a huge difference between the domain and data models.)

One entity which isn't represented above is the `VersionInformation` class. Its purpose is to represent partial information about a Document – namely any version details. This is used in the UI to display a dropdown of alternatives `Versions` for the current `Document` which can be viewed:



When we cover the domain layer in depth, we'll go over the pros and cons of creating a new class and new data mapping functions to accommodate this functionality – rather than trying to reusing what we've done for the `Document` class. This dilemma isn't unique to our implementation though. Even with an ever-so-easy `DataSet` you'd have to make the choice between pulling a lot of extra information or writing a new DAL method and stored procedure to get precisely what you need.

For me, the `VersionInformation` class really highlights the difference, from a business perspective, between a `Document` and a `Version`.

The Data Access Layer

The glue between our two models is the Data Access Layer (DAL) which exists as part of each layer – in the shape of C# classes, ADO.NET and stored procedures. With respect to this conversation, the mapping from the data model to our domain objects is done via the `AdoDataMapper` class which contains a number of static members which take in a `DataReader` and spit out an instance of a domain object. All of these methods are very straightforward:

```
internal static Document CreateDocument(SafeDataReader dr)
{
   Document document = new Document();
   document.Id = dr.GetInt32("Id");
   document.Name = dr.GetString("Name");
   document.DateCreated = dr.GetDateTime("DateCreated");
   ...
   return document;
}
```

The method takes in a `SafeDataReader` (just think of it as a `DbNull`-aware `DataReader`) and spits out an instance of a `Document` class. The real transformation occurs within our stored procedure; in the shape of a simple `INNER JOIN` (the code below actually gets the latest version of a `Document`):

```
SELECT R.[Name], V.[Content], @ResourceId AS Id, V.DateCreated, ...
FROM Resources R INNER JOIN Versions V ON R.Id = V.ResourceId
WHERE V.IsLatest = 1 AND V.ResourceId = @ResourceId
```

It would be possible to use the same kind of logic when doing an insert, update or delete. That is, we could give our `AdoDataMapper` a domain object, such as an instance of `Document`, and have it spit out an array of `DbParameter` types which is then assigned to an `SqlCommand` and executed. However, in my experience, it's common to `SELECT` complete entities, while only requiring parts of the data to fulfill an `UPDATE` or `DELETE`. For that reason, the mapping is done directly in our main DAL class.

Undoubtedly, some readers would have used an O/R mapper for their DAL. Hopefully I'm going to get the opportunity to dedicate a chapter to that possibility.

The Structure

The Code Wiki is composed of four highly cohesive projects. `CodeBetter.TheCodeWiki.Web` is the web project responsible for our presentation layer. It contains all our ASPX and CodeBehind files, our `HttpHandlers` (and `HttpModules` if we had any), our user controls and custom server controls.

> *I borrowed (with permission) the CodeBetter name to show proper namespace naming. I really hate namespaces that contain the name of an individual – so calling it KarlSeguin.TheCodeWiki was out.*

The presentation layer sits atop the `Codebetter.TheCodeWiki` project which is our business layer and part of our data access layer. This layer contains our data model and our business rules – it's the heart of our system. It isn't always necessary to physically separate our layers, but it can help you better organize your code.

The `CodeBetter` project is meant to be reused across all CodeBetter systems. It contains a simple validation engine and a base class, `EntityBase`, which provides basic functionality for entity classes. On most projects, I commonly rely on a system-agnostic Web project, say `CodeBetter.Web`. The need to build one for The Code Wiki just never really came up. (Truth be told, I initially skipped having a `CodeBetter` project, but eventually felt its addition would prove meaningful.)

Finally, the `CodeBetter.TheCodeWiki.UnitTests` project contains all of our unit tests. Most of our unit tests focus on the domain layer, but we also have some database integration tests.

There's actually a fifth project that isn't visible on the website: `CodeBetter.TheCodeWiki.Importer`. The Importer is a windows form application which is responsible for most of the data input into The Code Wiki. It scans my project folder, looks for new versions and commits the data to the database. Like the web project, it too is a consumer of the business layer. I left it out because I felt it would simply confuse readers, plus it's not particularly well written (I needed something quick which I would be the only user of, why architect a grand solution for that?) If you see a function in domain layer like `Document.Update` which doesn't seem to be used anywhere there's a good chance that it exists for the Importer.

Time to Zoom In

Our introduction of The Code Wiki and its goals, along with a high level overview is complete. From here on in we'll be dealing directly with The Code Wiki application, first by examining each of the three main layers, then covering topics such as unit testing, continuous integration and O/R mapping. It's worth repeating that The Code Wiki only represents one of the many ways to skin the proverbial cat. Some readers will be disappointed at my quick dismissal of `DataSets`, while others will be offended that I didn't use an O/R mapper. The vastness of tools available to us is both a blessing and a curse, especially when you consider that the combination of techniques is virtually unlimited. Perhaps you like the idea of a well-defined API via custom entities but rather use `DataSets` as the basis for your DAL. Maybe you're writing code that isn't very important or that has a very limited audience, such as The Code Wiki Importer, and just want to forgo clean design for the sake of getting something done quickly. Whatever you prefer, remember that the hard work you do upfront will be beneficial throughout the life of the project.

Further Reading

The following are links to resources relevant to the topics we've covered in this chapter.

Scott W. Ambler has a really nice introduction on the object-impedance mismatch and other relevant topics. It's a great link to pass on to a coworker or even a manager (while you're at it, don't forget to mention The Code Wiki's site too!):
http://www.agiledata.org/essays/impedanceMismatch.html

Jimmy Nilsson wrote a 5 part article for Informit which looks at the pros and cons of a number of possible data access approaches:
http://www.informit.com/articles/article.asp?p=31099
http://www.informit.com/articles/article.asp?p=31325
http://www.informit.com/articles/article.asp?p=31457
http://www.informit.com/articles/article.asp?p=31672
http://www.informit.com/articles/article.asp?p=99034

Jeremy Miller has a nice blog post about not letting your data model constrain your domain model and vice versa (the comments get a little off-track though):

http://nhibernate.codebetter.com/blogs/jeremy.miller/archive/2007/02/23/Don_2700_t-Let-the-Database-Dictate-Your-Object-Model.aspx

Appendix A –The ASP.NET Page

An ASP.NET page, and its corresponding CodeBehind file, is another one of those abstractions which skilled ASP.NET developers should strive to better understand. We'll look at the ASP.NET 1.x model because it's more straightforward than the 2.0 model. The 2.0 model is very similar, except it leverages the introduction of partial classes.

In ASP.NET 1.x, ASPX pages and User Controls, identified by the `@Page` and `@Control` directives respectively, are Just-In-Time (JIT) parsed into .NET classes and then compiled into assemblies. This partially explains why the first hit to your site is slower than subsequent ones. The relationship between a page and its CodeBehind is defined by the `Inherits` attribute of the ASPX page. The JIT'd class simply inherits from the CodeBehind class (which subsequently inherits from the `Page` class).

This explains why, in ASP.NET 1.X, your CodeBehind file must declare, as protected fields, all of the controls in your ASPX file. It's no different than any other super-class/sub-class relationship. Your ASPX file instantiates the controls, your CodeBehind file declares them so that you can programmatically access them.

Let's look at a simple ASPX page and CodeBehind file for a login form contained inside a table, called login.aspx and login.aspx.cs (which has been compiled into a dll and placed into the /bin folder). When you first request login.aspx, ASP.NET determines that it doesn't have a cached version of the login.aspx class and initializes the JIT compiler. The JIT compiler turns login.aspx into a class named `ASP.login_aspx`. If the JIT compiler doesn't find an `Inherits` attribute, `ASP.login_aspx` will inherit from the `Page` class. In our example, our page directive looks like: `<%@ Page Inherits="Login" %>`

The main entry point into the JIT'd class is the overridden `FrameworkInitialize` function. `FrameworkInitialize` sets up the proper file dependencies (so that if you change login.aspx, the file is automatically recompiled) and calls a private function named `BuildControlTree`. Looking at the code inside `BuildControlTree`, you start to get a sense for how an ASPX file can be turned into a class:

```
private void BuildControlTree(Control ctrl)
{
  IParserAccessor accessor = ctrl;
  accessor.AddParsedSubObject(
      new LiteralControl("\r\n<html><head><title>User
                          Login</title></head><body\r\n>")
  );
  BuildControl2();
  accessor.AddParsedSubObject(this.control2);
  accessor.AddParsedSubObject(new LiteralControl("\r\n</body>\r\n</html>"));
}
```


Notice how the HTML is simply turned into a `Literal` control and added to the control hierarchy. `BuildControl2` actually builds everything inside (and including) our `<form runat="server">`. Here's a simplified version of `BuildControl2`:

```
private Control BuildControlControl2()
{
  HtmlForm form = new HtmlForm();
  this.control2 = form;
  IParserAccessor accessor = form;


  accessor.AddParsedSubObject(
          new LiteralControl("\r\n<table border=\"0\"><tr><td>Username :
                                  </td>\r\n<td>"));
   BuildControlUsername();
   accessor.AddParsedSubObject(base.Username);
   // now do the same thing for the Password cell/label/textbox and the
   // Submit button
   return form;
}
```

Here the private `control2` field, which was used in the above `BuildControlTree` function, is instantiated as an `HtmlForm`. Once again the HTML (specifically the table) is turned into a `Literal` control and added to the form. The most interesting part in all of this is the call to `BuildControlUsername` which is responsible for instantiating our username textbox. For each of the `runat="server"` controls inside your form, a similar function will exist:

```
private void BuildControlUsername()
{
   TextBox box = new TextBox();
   base.Username = box;
   box.ID = "Username";
   box.MaxLength = 0x40;
}
```

Notice that the instance of our new textbox is assigned to our base class's `Username` field – that is the `protected TextBox Username` definition in our CodeBehind file. To me at least, this makes the relationship between an ASPX page and its CodeBehind file very clear.

Since ASPX pages are just classes, it is possible to have multiple classes inherit from the same CodeBehind file. This could be useful if you have two very similar forms, such as a basic and advanced search and want to reuse the same CodeBehind logic. Simply make both ASPX or User Controls `Inherit` the same CodeBehind file and defensively write your CodeBehind logic. Here's a sample CodeBehind file that could be used by two separate search User Controls:


```
Protected Search As Button
Protected Name As TextBox
Protected City As DropDownList
Protected MaximumPrice As TextBox

Private Sub Search_Click(ByVal sender As Object, ByVal e As EventArgs)
   Dim cityValue As String
   Dim maximumPriceValue As Integer

   If (City Is Nothing) Then
      cityValue = "DefaultCity"
   Else
      cityValue = City.SelectedValue
   End If
   If (MaximumPrice Is Nothing) Then
      maximumPriceValue = 100000
   Else
      'todo: better input validation
      maximumPriceValue = CInt(MaximumPrice.Text)
   End If

    Search(Name.Text, cityValue, maximumPriceValue)
End Sub

Private Sub Search(...)
   'todo: do the search and bind the results to a grid
End Sub
```

If a User Control doesn't create a control with an `Id` of `City` or `MaximumPrice`, the CodeBehind fields will never get assigned and the logic will fall back to default values.

If you want to see the JIT'd code for yourself, and you're using Visual Studio 2003, you can find it in: `C:\windows\Microsoft.NET\Framework\v1.1.4322\Temporary ASP.NET Files`.

If you are running Visual Studio 2005 and using the built-in webserver, you'll need to look at: `%Temp%/Temporary ASP.NET Files`.

It's somewhat confusing in there, but go ahead and dump some of those files in Reflector until you find the right one.

> Milan Negova has a good blog entry about the impact of `debug=true` in your web.config's `<compilation>` element. It gives some relevant insight about how the JIT works and what its impact could be:
> http://aspnetresources.com/articles/debug_code_in_production.aspx

