Part I: Introduction to Practical Enterprise Development

1

What is Enterprise Design?

"Mr. Arking, your experience, certifications, and references are terrific, but unfortunately we can't hire you. Your resume just doesn't have enough enterprise experience . . . ."

That was the first time I had ever heard the word "enterprise." Wrapped subtly within a casserole of deprecating niceties and empty affirmations, assailing my staunch sense of geek-honor as I wrestled with the gravity of the moment. That was the first time I was ever turned away from a job in software development. Like most fallen nerds, I went through the normal stages of post-interview withdrawal. Similar to the departing of a close friend or relative, I felt denial, then outrage, then moved slowly into a state of grief. As the eccentric side of me skimmed through the torrent of emotions that accompany job overreaction, I began to evaluate what I did wrong. I have no enterprise experience? What does that mean? Didn't my interviewers read my resume? Weren't they impressed with my vast experience with different APIs? Didn't they appreciate my deep knowledge of different platforms and languages? I mean, look at all of the companies I worked for . . . all of the different applications I had built! Surely *some* of that demonstrated hands-on practice with enterprise!

After my failed interview I was determined to figure out what I had missed. I prided myself on being the consummate interviewee. I had all the right answers, knew all the best programming tricks. I had stacks of code samples and lots of great references to back up my work. I had to know where it all went wrong. My investigations inevitably led me back the job description which, among a great many requirements, listed the following work experience:

> *Ideal candidate will have extensive exposure to enterprise architecture, with a background in designing large systems for multifaceted, heterogeneous platform support.*

At first glance, this requirement didn't seem like much. Yet there must be something more to it than I was grasping. Multifaceted, heterogeneous platform support? What was that all about? I began to investigate what the employer meant by "enterprise patterns," expecting to find some subtlety of coding that I had likely touched on in one way or another. However, when I looked up enterprise architectures online, I was stunned to find a whole new level of design, one that had completely eluded me for over 8 years of computer programming. I took a good look at my resume

and began to understand that I was a horrible fit for the position. I had built some great applications, but that's all they were: applications. Some of the things I had developed were very impressive, requiring a lot of knowledge in coding and multi-tiered design. Still, I had never actually developed an *enterprise system*. I had never designed an infrastructure or set of processes for other developers in other areas to support. I had never participated in an architectural process, never wrote tests before I wrote my code. I hadn't ever established patterns and practices to support a broad context of business needs. In fact, I had little or no experience in all of the key concepts behind all things enterprise.

I had no enterprise experience. Despite the fancy software that I had spent my career building, I hadn't even so much as dabbled in the enterprise fray. Like a great many before me, I had fallen prey to the enterprise assumption that haunts even the fanciest computer programmer. I presumed that if I've programmed enough powerful software, and learned everything there is to learn about platform libraries, APIs and SDKs, then I have earned the right to call myself an enterprise developer. Not only is this assumption wrong, it is a widely accepted assumption in Microsoft programming community. Unfortunately, I had to learn that the hard way.

And so it began, my long journey into the world of enterprise architecture and development. Wiping away all preconceived notions of what software development was supposed to be, I immersed myself into the very culture of enterprise architecture. I read books on different development processes. I purchased trade magazines and followed articles dedicated to large system development. I subscribed to countless blogs and forums published by some of the biggest players in the enterprise community. I went enterprise native, if you will. The result was a complete retooling of my skills, and a revision of my approach to software design.

In this first chapter, we will cover the following areas:

- Discuss enterprise architecture and what it means
- Discuss enterprise development and how it complements enterprise architecture
- Talk about tools, patterns, and features that enforce the key goals of an enterprise design for .NET

Enterprise design can be confusing. It requires knowledge of many different languages and disciplines. It requires both low-level programming skills and higher level, comprehensive design experience. Most of all, it requires patience and tolerance for new ideas. Enterprise design patterns and methodologies come in many different forms, each of which has slowly worked its way into the Microsoft development platform. A platform almost exclusively dedicated to rapid application development (RAD), Microsoft applications have long been riddled with poor design, exchanging extensibility and flexibility for quick time-to-market. As enterprise methodologies become more widely embraced by a new generation of Microsoft developers, many find themselves faced with the daunting task of learning these new patterns, and incorporating them into their existing skill sets and applications.

But what do we mean when we talk about enterprise architecture? What exactly is enterprise development? The term enterprise is used widely in today's technical vernacular. Most software programmers throw the term around far too casually, applying it to almost any type of application design or framework. Yet enterprise software is anything but application- or system-specific. To clarify the matter, it's best to begin with a common understanding of what enterprise means and how it changes the way you develop.

What Is Enterprise Architecture?

Enterprise architecture is typically used to describe an agency-wide or organization-wide framework for portraying and incorporating the business processes, information flows, systems, applications, data, and infrastructure to effectively and efficiently support the organization's needs. At the heart of this definition lies a very broad context aimed at including many different portions of an organization's participating branches, chief among them the business and information technology departments. We could wax intellectual all day long on the merits of these descriptions, but seeing as how this is a book for developers, let's cut to the chase. What does enterprise architecture mean from a developer's point of view? It means defining a process, framework, and set of patterns to design, develop, build, and maintain all of the software that an agency or company needs to operate. The operative phrase here is *all of the software*. It is a unified development platform for creating all elements of software at all levels of design. It includes reusable tools for building client applications, websites, databases, office applications, business automation tools, scripts, and just about anything else that a company may use to get things done. Enterprise architecture also endeavors to break down each of an application's layers into modular pieces for reusability. These reusable elements can then be used to feed or drive other applications with similar needs. Here's where the picture starts to get a bit fuzzy. Most developers take on projects with a finite set of business goals, goals that satisfy a specific need or company requirement. Within that scope, there is little consideration for modularity or reusability outside of the system that is being built. On the contrary, project goals rarely allot the time and resources needed to accommodate what is in essence the *possibility* of component reuse. Instead, typical projects focus development on the end goal only, marginalizing or downright ignoring the larger enterprise picture. Understanding enterprise development means first realizing that this kind of myopic, and often cavalier, development is ultimately counterproductive.

Enterprise architecture is also about defining a solid foundation of code and practices that eventually (and inevitably) facilitate interoperability in a heterogeneous software environment. This foundation provides both a toolset for creating software application, as well as a set of boundaries and rules within which those writings said applications need to work. The combination of both process and toolset is one of the key concepts to creating enterprise software. It expands on the otherwise traditional concepts of computer programming that concentrated on *what* one coded and mostly ignored *how* one coded. The incorporation of software development methodologies and lifecycle management becomes as important a part of building an application as the code itself.

Of course, chances are that if you're reading this book, you've already come to know some sort of development methodology. From the iterative and flexible like Agile and Extreme Programming, to the evolving and maturing like Six Sigma and the Capability Maturity Model, software development methodologies have worked their way into mainstream software development. Still, methodologies alone do not define an enterprise architecture. Plenty of shops that build applications apply these methodologies rigidly, and many of them do not have enterprise software. An organization that embraces enterprise architecture endeavors to combine a broad-context framework with a development approach, ultimately yielding code that conforms to a level of quality and design that suits the organization's core needs. Ideally, this approach can do wonders for a company, ensuring quality and uniformity throughout all tiers of design. Yet anyone who's participated or contributed to more than one enterprise shop would agree that the ideal is difficult to attain. Business needs and company politics often work counter to the spirit of enterprise planning. They force stringent timelines and tight project budgets, and don't easily allow for the sort of flexibility that a good enterprise architecture requires. The result is a hackneyed combination of some processes and standards that add little more than cumbersome meetings and a few tidy lifecycle diagrams that barely placate the folks in charge. Thus, a

successful implementation of an enterprise system requires a comprehensive "buy-in" from both members of the business side and from the IT side.

What Is Enterprise Development?

Enterprise development commonly refers to the patterns and practices adopted by programmers endeavoring to implement enterprise architecture. It is the employment of certain approaches and methodologies that aim to achieve many of the root goals inherent to a successful enterprise system. What these goals are specifically changes from organization to organization; however, at the root, they address five key areas of system development:

1. Reliability
2. Flexibility
3. Separation of concerns
4. Reusability
5. Maintainability

These base tenets are embraced by all developers of enterprise systems, and they help to define the core of what most modern developers consider to be well-designed software. Enterprise development embraces these ideals, weaving them subtly into the tools and processes that drive software logic.

Reliability

Most would agree that designing systems that are reliable is a must. Yet coding for reliability is a departure from business as usual. This is especially true in the rapid application development community. Many enterprise enthusiasts exchange the term *reliability* for *testability*, since most modern enterprise coding patterns aim to facilitate unit testing. Writing code that can be well tested means changing the way that a system's functionality is modularized. It means flattening out otherwise bloated classes and removing dependencies, or rather, removing references to other code that prevent a class or module from being tested. Many of these design patterns are integral to a process known as Test Driven Development, or TDD. We will cover TDD in depth in Chapters 3 and 4.

Flexibility

Requirements can change. As a result, so must the software that supports them. If the code that you write prevents an application or system from being extensible or pliable, we would say it lacks flexibility. Many people mistake the need for flexibility for other popular engineering subjects, such as interoperability. However, enterprise flexibility addresses the ability of code to be broken down and shared by different systems applications. A program might be functional on different platforms, or contain logic for lots of different failure scenarios, but that wouldn't mean it was flexible.
A flexible system allows for the changing of core features without violating unrelated services or attributes.

Separation of Concerns

Separation of concerns is simply the process of breaking a system or application down into distinct functional layers with limited overlapping of functionality. Like flexibility, separation of concerns addresses the ability to modularize code and make it more pliable, with the added benefit of logical division. Much of this division can be achieved through well-known object-oriented tenets, such as modularization and encapsulation. As we explore new patterns of development, separation of concerns becomes an all-important piece of the enterprise puzzle.

Reusability

Sharing features and services is tantamount to good enterprise design. As code is broken down and separated into logical pieces, these pieces should be designed to provide a distinct feature or satisfy a particular requirement of other systems that invoke it. The scope of a class's reusability depends on the context in which it is used; however, most agree that other modular pieces of code within a similar context should always be callable. In other words, classes at any one logical level should be reusable by other classes in the same logical level. Classes that provide data should be consumable by all other classes that demand data within scope. Classes that implement a user interface (UI) behavior should deliver the same behavior to all UI-implementing classes on the same UI tier. The notion of reusability is especially important when designing true enterprise architectures.

Maintainability

Maintainability refers to the capacity of a system to be altered or modified. Although most software engineers think they know what maintainability means, it actually has a distinct meaning in the world of software design. According to the international standard of software development defined in ISO 9126, the term maintainability actually means the ease with which a software product can be modified in order to support:

- Stability
- Analyzability
- Changeability
- Testability

Maintainable code should be the natural result of following these four tenets, provided that the designer has not introduced unnecessary levels of complexity. The inherent balance between complexity and maintainability will be explored further in Chapter 2.

For a great many software engineers this can be a dramatic shift in the way they program. It requires a rigid manner of programming, employing new concepts and demanding more upfront design than the typical developer usually executes. At close glance, one might think that simple, non-enterprise computer code that delivers a particular feature is identical in value to enterprise code that delivers precisely the same feature. Yet this shortsighted evaluation fails to address the greater needs of the system, namely core enterprise concepts. While the code may deliver similar results, the enterprise code takes strides to accommodate better design. So while the enterprise sample might look a bit more complex (only at first, mind you), the resulting class or module ultimately provides more reliability or is more maintainable.

Consider a simple example. Two developers are required to build a web page that displays dynamic feeds of financial data. These feeds can range from stock quotes to bond prices to billboard articles from popular financial journals. The non-enterprise developer might write an ASP.NET page with a set of data grids, each bound to different database calls and online web services. The bindings are created in the ASP.NET code-behind page directly, placing a good amount of data logic side by side with some of the user interface behaviors. The enterprise developer would take a slightly different tack. Using the Model-View-Presenter pattern widely embraced within the enterprise community, they define a class to garner and hold the data on its own. They then create another class to handle the user interface logic and events, defining both classes with an abstract interface that defines each of the classes' core methods. Finally, the developer writes a series of unit tests and mock classes to test the code and ensure that all portions function as intended. The resulting software delivers precisely the same page and the same data, with much more orchestration and modularity. So the two efforts were a wash, right? Let's take this model a step further. The sample page hits its mark. Management is happy and they request that the page be published on the company website. The website requires a level of account authorization in order to query data from other data sources, but neither developer is aware of this. The non-enterprise developer deploys the web page directly and takes a quick look at it using a browser running on his/her desktop. They are logged in as an administrator, so the web page loads just fine. Unfortunately when others try to view the page they get a horrible system error that crashes the entire web session. On the other hand the enterprise developer took time to write a unit test that impersonates an anonymously authenticated user in the data access class. They run the test as a part of the build process and the problem becomes immediately apparent. The enterprise code is more reliable than the non-enterprise code. What's more, the composition of the web page code to support the unit tests allows for modularity and separation of concerns. So when management provides a new data service to use, the data layer can be altered with minimal impact on the user interface. The code is now more flexible, too. The added flexibility, along with the reliability and logical separation, makes the enterprise developer's web page far more maintainable than the non-enterprise developer's page.

Figure 1-1

An enterprise architecture can be difficult to build in its entirety. Typically speaking, an individual software engineer has limited say over how a business expends its resources.

For most developers this is a game-changing approach to writing code. The vast majority of developers in the Microsoft world concentrate on developing software for a single system or application. This is completely understandable when you consider that an overwhelming number of career Microsoft developers began as business analysts or specialists who, thanks to the proliferation of rapid application development tools in the Microsoft software space, took on coding as a means of automation. Yet despite the upfront convenience of rapid application development, the cost in the long run can be, and usually is, very high. Enterprise development aims to reduce these long-term costs with prudent designs, approaches, and testability.

Where Is All of the Microsoft Enterprise?

Enterprise development is not new. In fact, many of the core values that drive enterprise architecture have been around for quite a few years. The problem is, if you're a Microsoft programmer, you probably haven't encountered any of them. Most enterprise patterns come from the great wide open world of common developer contribution known by most of us as *open source*. As a result, most enterprise systems have been built with platform-independent technologies such as Java or even C++. That's not to say that Microsoft technologies have never been used to build big systems for large organizations. Remember, enterprise architecture does not necessarily mean "made for big companies." We simply point out that the evolution of the patterns and methodologies driving the core values have depended on contributions from the developer community. Until recently Microsoft technologies were anything but open source. Since the open source community was the fecund environment from which modern enterprise concepts emerged, Microsoft software has played a relatively minor role in the evolution of enterprise architecture.

In the Microsoft world, a great deal of emphasis is placed on the ability to create applications quickly. Tools that provide quick automation and almost instant results can be found all over the Windows world. These tools generally fall within a category of software development known as rapid application development, or RAD. RAD-style programming isn't necessarily meant to deliver production-worthy code. Rather, its primary goal is to enable developers and analysts to emulate the core behaviors of a system quickly, intentionally ignoring patterns and process for a quick bang factor. Unfortunately, a good deal of RAD development is downright counter to the core values of enterprise development.

RAD culture has worked its way strongly into the large developer base of Microsoft programmers. At the heart of this pervasive movement lies the simple, easy-to-use language Visual Basic. First given the moniker "Project Thunder" and released as VB 1.0 in mid-1991, VB has since become the most popular programming language among Windows developers and one of the most widely used programming languages in the world. Its verbose, non-C-style syntax is easy for analysts and the less technically inclined to follow. The language itself has undergone a number of dramatic changes and enhancements over the years. However, until the language itself was retooled for the managed world and named VB.NET, its primary focus was to provide RAD tools for building powerful, production applications. At the height of its popularity, VB was most widely employed in one of two forms: Visual Basic 6.0, or Visual Basic for Applications (VBA). VB6 was its own development suite. It included tools for quickly creating object-structured applications using visual designers and wizard-like feature builders (Figure 1-1). VBA was, and still remains, the language of macros and automation within Microsoft Office applications. Between these two suites, new generations of coders were empowered. Forgoing a great deal of process and design, applications were churned out at alarming rates, with little planning and even less testing. Simple Word documents and Excel spreadsheets were fashioned with VB forms that provided a more flexible user experience than would otherwise have been achievable. Websites that once required complex Internet Server API (ISAPI) filters and direct Internet Information Server (IIS) extensions could now use VB6 ActiveX components, giving a website access to the entire Component Object Model (COM) library with very little development overhead. As usage increased, so did the power of VB. In time, Visual Basic eclipsed C++ as the Microsoft language of choice, yielding just about as much power as a complicated C module but without all of the messy planning beforehand.

Unfortunately, VB development led to some staggeringly bad applications. Once the emphasis was placed on delivery and away from design and process, a large number of companies found themselves stuck with unreliable, inflexible systems. They might have been delivered quickly, but the cost of maintenance over time became staggering. Note that we are not passing blanket judgment on the VB developer community. On the contrary, we too, found ourselves building systems in VB6 for quite a few years. We simply mean to demonstrate how this formidable trend in Microsoft development is one of the big reasons why enterprise design patterns still elude the common Microsoft programmer. The RAD culture that grew out of Visual Basic development polarized the pattern-minded from the results-oriented, ultimately blocking the mainstream Windows developers from participating in the enterprise effort.

The COM Factor

Of course, Visual Basic alone wasn't the only thing preventing enterprise patterns from making their way into Microsoft code. Some tend to think that Microsoft's previous software development platform had much to do with this as well. Before we had the neat and clean world of .NET, most C and C++ programmers were forced to use its predecessor, the Component Object Model, or COM. COM is a Windows-specific technology that allowed for components from different applications to communicate and interoperate with one another. Specifically, it is an interface standard that facilitates inter-process communication and object creation within any application, service, or system that supports the technology. COM patterns drive most objects that were used in Visual Basic programming, many of which are still in use on the Windows operating system. However, many developers felt that implementing COM introduced complexity that wasn't well suited to business applications. At the heart of every COM object lives a single common interface named `IUnknown`. This interface defines three key COM method definitions: `AddRef`, `Release`, and `QueryInterface`, each of which is used to manage objects in memory and communicate between objects.

As an object-oriented pattern, COM was well orchestrated and seemed to deliver on some of the goals that drive enterprise development. However, not everyone embraced COM programming. Implementing `IUnknown` generally meant that you had to write your program in a lower-level programming language such as C or C++. Since most Microsoft developers preferred to develop with RAD tools and languages, direct COM usage was hard to come by. Instead, COM-powered technologies such as Object Linking and Embedding (OLE) and ActiveX were provided to RAD programmers as a means to leverage COM objects without all of the hassle of the `IUnknown` plumbing. Unfortunately, COM-powered technologies were much heavier than straight COM. They introduced overhead and had a tendency to run more slowly than systems that implanted COM directly. Additionally, integration with other platforms became mostly impossible. COM objects became synonymous with Microsoft objects. As a pattern it didn't accommodate interoperability with non-Windows platforms. So, while COM was a fine pattern for designing enterprise architectures, it was only used as a means to define the Microsoft API itself. Precious few organizations employed the COM pattern directly within their own code.

The Shift to Java

Java, a (mostly) operating system-independent development platform, was released by Sun Microsystems in 1995 and gained popularity as a powerful alternative to otherwise limited Windows-based development. Long before .NET was released, Java developers enjoyed the use of well-designed APIs, automatic memory management, and just-in-time compilation. The Java community experienced a ground swell of low-level language programmers interested in porting their skills to more business-friendly development platforms. A large number of C and C++ developers were drawn to Java as a

comfortable flavor of a C-style language that included some tenets of RAD without compromising some of the more academic portions of software engineering. As a platform, Java has also always been very community-oriented. It was released as free software under the GNU General Public Agreement and made freely available via downloading to programmers on different platforms. Other Java compilers have been released under the same set of public agreements, and although Sun never formalized Java with the ISO/IEC JTC1 standards body or the ECMA International, it quickly became the de facto standard for enterprise object-oriented systems. In late 2006, Sun released most of the Java SDK as free and open source software under the GNU General Public License.

As a result, the Java community was a fertile ground for the next generation of enterprise architecture. Large, multifaceted development patterns began to grow, weaving their way into medium-sized organizations at first and into vastly larger ones as time went on. The Java 2 Enterprise Edition (J2EE) provided distinct multi-tiered patterns, such as Servlets and Java Server Pages (JSP) for Web and *n*-tiered systems. Message Oriented Middleware (MOM) patterns began to form, evolving into Java Message Service (JMS) and facilitating asynchronous communication between disparate servers. Front-end orchestration patterns such as Struts paved the way for better decoupling of front-end interfaces from the code that handles data and application state. To this day, an overwhelmingly large percentage of the open source community is firmly rooted in Java and J2EE, contributing to public software projects and providing a loud voice in the enterprise community.

The .NET Revolution

In 2002, Microsoft officially released the .NET Framework version 1.0. .NET was a revolutionary departure from any development platform Microsoft had ever released. It came with a large set of pre-coded libraries that exposed or wrapped most of the core functionality within the Microsoft software development kit (SDK). It included a code management system known as the Common Language Runtime (CLR) that managed memory, loaded classes, and delivered just-in-time compilation to applications written in a .NET-enabled language. .NET languages run the gamut from script languages to older mainframe languages such as Cobol.net. However, the most popular are VB .NET, Microsoft's next generation of the popular Visual Basic language, and C#, a C-based language created by Microsoft specifically for building .NET applications. Unlike the COM and ActiveX of old, .NET was designed from the ground up to be a comprehensive development and runtime environment. Its mature combination of APIs, development tools, and runtime services makes it a far better candidate for building enterprise applications. .NET mirrors other managed software platforms in terms of tools and services provided, and it is still considered by most to be Microsoft's answer to the Java development platform. However, unlike Java, .NET didn't have the impetus of academic and enterprise developers driving it. On the contrary, at the time of .NET's release most Microsoft developers were RAD programmers or automation engineers. Thus, there wasn't a whole lot of enterprise development happening in the world of Windows.

In time, that began to change. As the .NET Framework became more mature (and as Microsoft was beefing up its server architecture a bit) more and more people began to recognize some of its enterprise advantages. .NET languages, particularly C#, were well suited for complex patterns. Language constructs such as delegates and events, properties, and indexers, made C# a suitable candidate for writing consumable, decoupled APIs. Microsoft has also taken a more accepting position to the world of open source. Upon initial release of the framework, Microsoft published the Common Language Infrastructure (CLI) and submitted it, along with specifications for the C# language and C++/CLI

language, to both ECMA and ISO, making them accessible as open standards. Microsoft at first held tight to its patents on its core technologies such as its user interface API (Windows Forms) and their data access API (ADO.NET). However, in October of 2007, Microsoft announced that it would release much of the source code for the .NET base class libraries under the shared source Microsoft Reference License. This code was released in the next version of their popular interactive development environment, Visual Studio.NET 2008 edition. These steps helped to make .NET more attractive to a broader set of software engineers.

Over the last few years a number of .NET open source communities have emerged, driving new projects and embracing enterprise patterns as a means of more complicated system development. The Mono project (Figure 1-2), an initiative to create a suite of .NET tools and services targeting multiple operating systems, was established in July of 2001 and uses the public CLI specifications within its software. SourceForge.NET (see Figure 1-3), an open source community code repository, has a growing number of .NET projects.

Figure 1-2

Many of the popular Java open source projects, such as LogForJ and Hibernate, have .NET sister projects that started in SourceForge.NET.

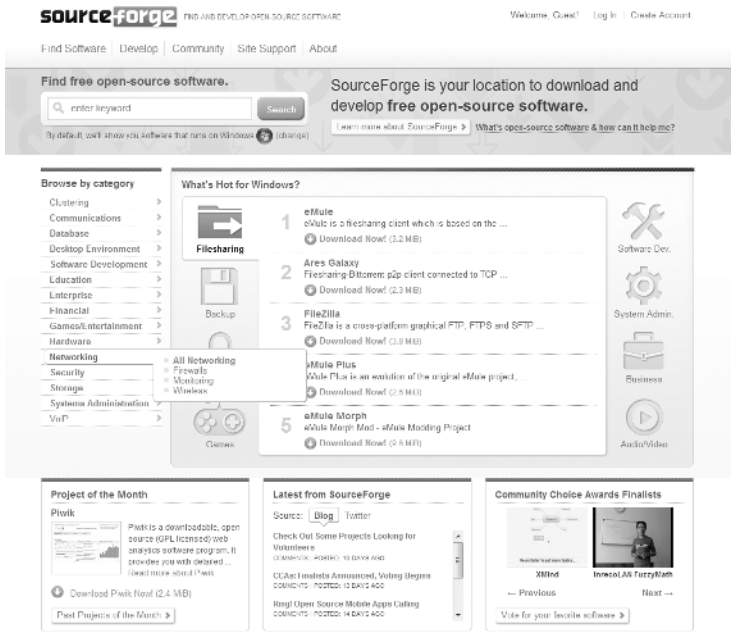

Figure 1-3

SourceForge.net is a terrific resource for finding and exploring open source projects from all areas of the software development industry. It also includes community forums, information for system administrators, and a marketplace for buying and selling projects.

Perhaps most notable among the open source communities is Codeplex.com (Figure 1-4). Codeplex is Microsoft's open source project hosting site. Launched in June of 2006 and powered by Microsoft's Team Foundation Server, Codeplex had accumulated over 3500 public .NET projects by early 2008. It remains one of the primary avenues of code patterns used by Microsoft today.

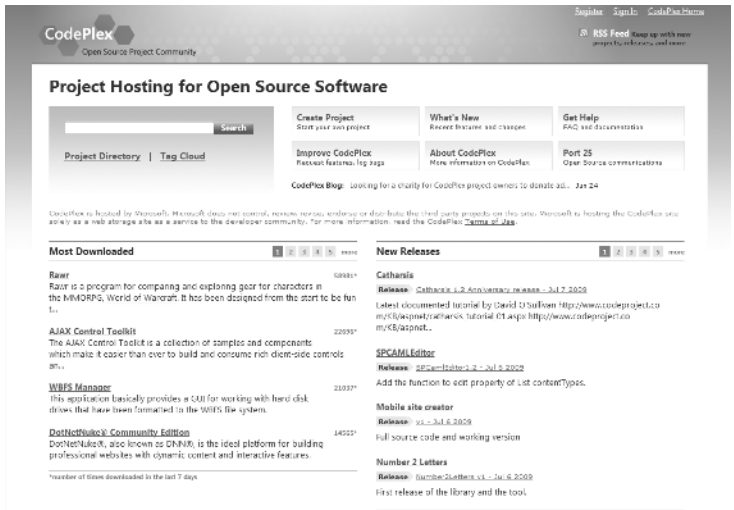

Figure 1-4

Today, there are a number of tools and projects available for the .NET developer. Some these tools have grown out of existing enterprise projects for non-Windows platforms, while others have been created specifically for .NET development. Throughout this book, we will be exploring those tools and patterns, familiarizing you with the features they offer, as well as with the processes and frameworks in which they operate. Some patterns might seem confusing at first, adding layers of complexity to otherwise simple code samples. As you begin to get used to enterprise patterns within your code, the intrinsic value provided by enterprise design will become evident. As we cover more and more topics and your enterprise horizons broaden, this learning curve will become increasingly shallow until the patterns and processes become a part of your everyday development approach.

Summary

In this chapter, we began our journey into the professional enterprise development world by focusing on some of the key concepts and core values of enterprise development:

- We explored the nature of enterprise development, discussing how enterprise applications are fundamentally different from standalone applications.
- We defined enterprise architecture as a comprehensive set of tools, features, patterns, and processes that enable an organization to achieve its goals.
- We distinguished software development from the broader context of enterprise architecture, defining enterprise development as software programming that aims to achieve and support enterprise goals.
- We reviewed the core values of enterprise development, underscoring the five base tenets of well-designed code: reliability, flexibility, separation of concerns, reusability, and maintainability.
- We explored some of the background of Microsoft development.
- We reviewed the history of rapid application development within the Microsoft community, comparing it to the more rigid and well-defined Java development platform from which a great number of enterprise patterns emerged.
- Finally, we discussed Microsoft's.NET Framework, exploring some of its key benefits and explaining how and why .NET is well suited for the next generation of enterprise development.

In the next chapter we will begin to explore some of the code patterns and methodologies inherent to building enterprise systems. Using the information from this chapter as a theoretical foundation, we will focus on using programming patterns that help to increase code quality and flexibility.