PART I

Introducing Patterns and Principles

COPYRIGHTED MATERIAL

1

The Pattern for Successful Applications

WHAT'S IN THIS CHAPTER?

- An introduction to the Gang of Four Design Patterns
- An overview of some common design principles and the SOLID design principles
- A description of Fowlers Enterprise Patterns

John Lennon once wrote, "There are no problems, only solutions." Now, Mr. Lennon never, to my mind, did much in the way of ASP.NET programming; however, what he said is extremely relevant in the realm of software development and probably humanity, but that's a whole other book. Our job as software developers involves solving problems — problems that other developers have had to solve countless times before albeit in various guises. Throughout the lifetime of object-oriented programming, a number of patterns, principles, and best practices have been discovered, named, and catalogued. With knowledge of these patterns and a common solution vocabulary, we can begin to break down complex problems, encapsulate what varies, and develop applications in a uniformed way with tried and trusted solutions.

This book is all about introducing you to design patterns, principles, and best practices that you can apply to your ASP.NET applications. By their very nature, patterns and principles are language agnostic, so the knowledge gained in this book can be applied to win forms, WPF and Silverlight applications, as well as other first-class object-oriented languages.

This chapter will cover what design patterns are, where they come from, and why it's important to study them. Fundamental to design patterns are solid object-oriented design principles, which will be covered in this chapter in the form of Robert Martin's S.O.L.I.D. principles. I will also introduce you to some more advanced patterns as laid out in Martin Fowler's *Patterns of Enterprise Application Architecture* book.

DESIGN PATTERNS EXPLAINED

Design patterns are high-level abstract solution templates. Think of them as blueprints for solutions rather than the solutions themselves. You won't find a framework that you can simply apply to your application; instead, you will typically arrive at design patterns through refactoring your code and generalizing your problem.

Design patterns aren't just applicable to software development; design patterns can be found in all areas of life from engineering to architecture. In fact, it was the architect Christopher Alexander who introduced the idea of patterns in 1970 to build a common vocabulary for design discussion. He wrote:

> *The elements of this language are entities called patterns. Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use this solution a million times over without ever doing it the same way twice.*

Alexander's comments are just as applicable to software design as they are to buildings and town planning.

Origins

The origins of the design patterns that are prevalent in software architecture today were born from the experiences and knowledge of programmers over many years of using object-oriented programming languages. A set of the most common patterns were catalogued in a book entitled *Design Patterns: Elements of Reusable Object-Oriented Software*, more affectionately known as the *Design Patterns Bible*. This book was written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, better known as the Gang of Four.

They collected 23 design patterns and organized them into 3 groups:

- **Creational Patterns:** These deal with object construction and referencing.
- **Structural Patterns:** These deal with the relationships between objects and how they interact with each other to form larger complex objects.
- **Behavioral Patterns:** These deal with the communication between objects, especially in terms of responsibility and algorithms.

Each pattern is presented in a template so readers can learn how to decipher and apply the pattern. We will be covering the practical knowledge necessary to use a design pattern template in Chapter 2 along with a brief overview of each pattern that we will be looking at in the rest of this book.

Necessity

Patterns are essential to software design and development. They enable the expression of intent through a shared vocabulary when problem solving at the design stage as well as within the source code. Patterns promote the use of good object-oriented software design, as they are built around solid object-oriented design principles.

Patterns are an effective way to describe solutions to complex problems. With solid knowledge of design patterns, you can communicate quickly and easily with other members of a team without having to be concerned with the low-level implementation details.

Patterns are language agnostic; therefore, they are transferable over other object-oriented languages. The knowledge you gain through learning patterns will serve you in any first-class object-oriented language you decide to program in.

Usefulness

The useful and ultimate value of design patterns lies in the fact that they are tried and tested solutions, which gives confidence in their effectiveness. If you are an experienced developer and have been programming in .NET or another object-oriented language for a number of years, you might find that you are already using some of the design patterns mentioned in the Gang of Four book. However, by being able to identify the patterns you are using, you can communicate far more effectively with other developers who, with an understanding of the patterns, will understand the structure of your solution.

Design patterns are all about the reuse of solutions. All problems are not equal, of course, but if you can break down a problem and find the similarities with problems that have been solved before, you can then apply those solutions. After decades of object-oriented programming, most of the problems you'll encounter will have been solved countless times before, and there will be a pattern available to assist in your solution implementation. Even if you believe your problem to be unique, by breaking it down to its root elements, you should be able to generalize it enough to find an appropriate solution.

The name of the design pattern is useful because it reflects its behavior and purpose and provides a common vocabulary in solution brainstorming. It is far easier to talk in terms of a pattern name than in detail about how an implementation of it would work.

What They Are Not

Design patterns are no silver bullet. You have to fully understand your problem, generalize it, and then apply a pattern applicable to it. However, not all problems require a design pattern. It's true that design patterns can help make complex problems simple, but they can also make simple problems complex.

After reading a patterns book, many developers fall into the trap of trying to apply patterns to everything they do, thus achieving quite the opposite of what patterns are all about — making things simple. The better way to apply patterns, as stated before, is by identifying the fundamental problem you are trying to solve and looking for a solution that fits it. This book will help with the identification of when and how to use patterns and goes on to cover the implementation from an ASP.NET point of view.

You don't always have to use design patterns. If you have arrived at a solution to a problem that is simple but not simplistic and is clear and maintainable, don't beat yourself up if it doesn't fit into one of the 23 Gang of Four design patterns. Otherwise, you will overcomplicate your design.

This talk of patterns might seem rather vague at the moment, but as you progress through the book, you will learn about the types of problems each pattern was designed to solve and work through implementations of these patterns in ASP.NET. With this knowledge, you can then apply the patterns to your applications.

DESIGN PRINCIPLES

Design principles form the foundations that design patterns are built upon. They are more fundamental than design patterns. When you follow proven design principles, your code base becomes infinitely more flexible and adaptable to change, as well as more maintainable. I will briefly introduce you to some of the more widely known design principles and a series of principles known as the S.O.L.I.D. principles. Later in the book we will look at these principles more deeply and implement them and best practices in ASP.NET.

Common Design Principles

There are a number of common design principles that, like design patterns, have become best practice over the years and helped to form a foundation onto which enterprise-level and maintainable software can be built. The following sections preview some of the more widely known principles.

Keep It Simple Stupid (KISS)

An all-too-common issue in software programming is the need to overcomplicate a solution. The goal of the KISS principle is concerned with the need to keep code simple but not simplistic, thus avoiding any unnecessary complexities.

Don't Repeat Yourself (DRY)

The DRY principle aims to avoiding repetition of any part of a system by abstracting out things that are common and placing those things in a single location. This principle is not only concerned with code but any logic that is duplicated in a system; ultimately there should only be one representation for every piece of knowledge in a system.

Tell, Don't Ask

The Tell, Don't Ask principle is closely aligned with encapsulation and the assigning of responsibilities to their correct classes. The principle states that you should to tell objects what actions you want them to perform rather than asking questions about the state of the object and then making a decision yourself on what action you want to perform. This helps to align the responsibilities and avoid tight coupling between classes.

You Ain't Gonna Need It (YAGNI)

The YAGNI principle refers to the need to only include functionality that is necessary for the application and put off any temptation to add other features that you may think you need. A design methodology that adheres to YAGNI is test-driven development (TDD). TDD is all about writing tests that prove the functionality of a system and then writing only the code to get the test to pass. TDD is discussed a little later in this chapter.

Separation of Concerns (SoC)

SoC is the process of dissecting a piece of software into distinct features that encapsulate unique behavior and data that can be used by other classes. Generally, a concern represents a feature or behavior of

a class. The act of separating a program into discrete responsibilities significantly increases code reuse, maintenance, and testability.

The remainder of this book refers back to these principles so you can see how they are implemented and help form clean and maintainable object-oriented systems. The next group of design principles you will look at were collected together under the grouping of the S.O.L.I.D. design principles.

The S.O.L.I.D. Design Principles

The S.O.L.I.D. design principles are a collection of best practices for object-oriented design. All of the Gang of Four design patterns adhere to these principles in one form or another. The term S.O.L.I.D. comes from the initial letter of each of the five principles that were collected in the book *Agile Principles, Patterns, and Practices in C#* by Robert C. Martin, or Uncle Bob to his friends. The following sections look at each one in turn.

Single Responsibility Principle (SRP)

The principle of SRP is closely aligned with SoC. It states that every object should only have one reason to change and a single focus of responsibility. By adhering to this principle, you avoid the problem of monolithic class design that is the software equivalent of a Swiss army knife. By having concise objects, you again increase the readability and maintenance of a system.

Open-Closed Principle (OCP)

The OCP states that classes should be open for extension and closed for modification, in that you should be able to add new features and extend a class without changing its internal behavior. The principle strives to avoid breaking the existing class and other classes that depend on it, which would create a ripple effect of bugs and errors throughout your application.

Liskov Substitution Principle (LSP)

The LSP dictates that you should be able to use any derived class in place of a parent class and have it behave in the same manner without modification. This principle is in line with OCP in that it ensures that a derived class does not affect the behavior of a parent class, or, put another way, derived classes must be substitutable for their base classes.

Interface Segregation Principle (ISP)

The ISP is all about splitting the methods of a contract into groups of responsibility and assigning interfaces to these groups to prevent a client from needing to implement one large interface and a host of methods that they do not use. The purpose behind this is so that classes wanting to use the same interfaces only need to implement a specific set of methods as opposed to a monolithic interface of methods.

Dependency Inversion Principle (DIP)

The DIP is all about isolating your classes from concrete implementations and having them depend on abstract classes or interfaces. It promotes the mantra of coding to an interface rather than an implementation, which increases flexibility within a system by ensuring you are not tightly coupled to one implementation.

Dependency Injection (DI) and Inversion of Control (IoC)

Closely linked to the DIP are the DI principle and the IOC principle. DI is the act of supplying a low level or dependent class via a constructor, method, or property. Used in conjunction with DI, these dependent classes can be inverted to interfaces or abstract classes that will lead to loosely coupled systems that are highly testable and easy to change.

In IoC, a system's flow of control is inverted compared to procedural programming. An example of this is an IoC container, whose purpose is to inject services into client code without having the client code specifying the concrete implementation. The control in this instance that is being inverted is the act of the client obtaining the service.

Throughout this book, you will examine each of the S.O.L.I.D. principles in more detail. Next, however, you will investigate some enterprise-level patterns designed to deal with specific scenarios that are built upon common design principles and design patterns.

FOWLER'S ENTERPRISE DESIGN PATTERNS

Martin Fowler's *Patterns of Enterprise Application Architecture* book is a best practice and patterns reference for building enterprise-level applications. As with the GoF patterns book, experienced developers will no doubt already be following many of the catalogued patterns. The value in Fowler's work, however, is the categorization of these patterns along with a common language for describing them. The book is split into two sections. The first half deals with n-tier applications and the organizing of data access, middleware, and presentation layers. The second half is a patterns reference rather like the GoF patterns book but more implementation specific.

Throughout this book, you will be looking at the ASP.NET implementations of Fowler's patterns. The following sections examine what the rest of the book will tackle.

Layering

Chapter 3 covers the options at your disposal to layer an enterprise ASP.NET application. You will look at the problems with the traditional code behind the model of web forms, and how to separate the concerns of presentation, business logic, and data access with a traditional layered approach.

Domain Logic Patterns

Chapter 4 examines three popular methods for organizing your business logic: Transaction Script, Active Record, and Domain Model.

Transaction Script

Transaction Script is the organization of business logic in a linear, procedural fashion. It maps fine-grained business use cases to fine-grained methods.

Active Record

Active Record organizes business logic in a way that closely matches the underlying data structure, namely an object that represents a row in a table.

Domain Model

The Domain Model pattern is an abstraction of real domain objects. Both data and behavior are modeled. Complex relationships between objects can exist that match the real domain.

You will look at how to use each of these patterns in ASP.NET and when it is appropriate to choose one pattern over another.

Object Relational Mapping

In Chapter 7 your attention will turn to how you can persist the state of our business entities as well as how you can retrieve them from a data store. You will look at the enterprise patterns required for the infrastructure code to support persistence, including the patterns introduced in the following sections.

Unit of Work

The Unit of Work pattern is designed to maintain a list of business objects that have been changed by a business transaction, whether that be adding, removing, or updating. The Unit of Work then coordinates the persistence of the changes as one atomic action. If there are problems, the entire transaction rolls back.

Repository

The Repository pattern, by and large, is used with logical collections of objects, or *aggregates* as they are better known. It acts as an in-memory collection or repository for business entities, completely abstracting away the underlying data infrastructure.

Data Mapper

The Data Mapper pattern is used to hydrate an object from raw data and transfer information from a business object to a database. Neither the business object nor the database is aware of the other.

Identity Map

An Identity Map keeps tabs on every object loaded from a database, ensuring everything is loaded only once. When objects are subsequently requested, the Identity Map is checked before retrieving from the database.

Lazy Loading

Lazy or deferred loading is the act of deferring the process of obtaining a resource until it's needed. If you imagine a Customer object with an address book, you could hydrate the customer from the database but hold the population of the address book until the address book is needed. This enables the on-demand loading of the address book, thus avoiding the hit to the database if the address data is never needed.

Query Object

The Query Object pattern is an implementation of a Gang of Four interpreter design pattern. The query object acts as an object-oriented query that is abstracted from the underlying database, referring to

properties and classes rather than real tables and columns. Typically, you will also have a translator object to generate the native SQL to query the database.

Web Presentation Patterns

In Chapter 8, you will turn your attention to the presentation needs of enterprise-level ASP.NET applications. The chapter focuses on patterns designed to keep business logic separate from presentation logic. First you will look at the problems with the code behind model that was prominent in early web forms development; then you will investigate patterns that can be used to keep domain and presentation logic separate, as well as allowing the presentation layer to be effectively tested.

Each of these patterns is tasked with separating the concerns of presentation logic with that of business logic. The patterns covered for ASP.NET presentation needs are:

- Model-View-Presenter
- Model-View-Controller
- Front Controller
- Page Controller

Base, Behavioral, and Structural Patterns

Throughout the book, you will be seeing how to leverage other enterprise patterns found in Fowler's book in enterprise ASP.NET applications. These patterns will include Null Object, Separated Interface, Registry, and Gateway.

Null Object Pattern

Also known as the Special Case pattern, this acts as a return value rather than returning null to the calling code. The null object will share the same interface or inherit from the same base class as the expected result, which alleviates the need to check for null cases throughout the code base.

Separated Interface

The Separated Interface pattern is the act of keeping the interfaces in a separate assembly or namespace to the implementations. This ensures that the client is completely unaware of the concrete implementations and can promote programming to abstractions rather than implementations and the Dependency Inversion principle.

Gateway

The Gateway pattern allows clients to access complex resources via a simplified interface. The Gateway object basically wraps the resource API into a single method call that can be used anywhere in the application. It also hides any API complexities.

All of the enterprise patterns introduced here will be covered in more detail throughout the book with exercises to see how they are implemented in an ASP.NET scenario. The next section wraps up the chapter with a brief look at design methodologies and practices that use the patterns and principles you have been introduced to in this chapter.

OTHER DESIGN PRACTICES OF NOTE

In addition to the design patterns, principles, and enterprise patterns that have been covered so far, I would like to introduce you to a few design methodologies: test-driven development, behavior-driven development, and domain-driven development. This section won't cover these topics deeply because they are out of the scope of this book. However, the sample code featured in each of the chapters to demonstrate patterns and principles that you can download from `www.wrox.com` has been designed using these methodologies.

Test-driven Development (TDD)

Contrary to the name, TDD is more of a design methodology than a testing strategy; the name simply just doesn't do it justice. The main concept behind it is to allow your tests to shape the design of a system. When creating a software solution you start by writing a failing test to assert some business logic. Then you write the code to get that test to pass; last, you clean up any code via refactoring. This series of steps has been coined the *red-green-refactor*. The red and green refer to the colors that testing frameworks use to show tests passing and failing.

By going through the process of TDD, you end up with a loosely coupled system with a suite of tests that confirm all behavior. A byproduct of TDD is that your tests provide a sort of living documentation that describes what your system can and can't do. Because it is part of the system, the tests will never go out of date, unlike written documentation and code comments.

For more information on TDD, take a look at these books:

- *Test Driven Development: By Example* by Kent Beck
- *The Art of Unit Testing: With Examples in .NET* by Roy Osherove
- *Professional Enterprise .NET* by Jon Arking and Scott Millett (published by Wrox)

Domain-driven Design (DDD)

In a nutshell, DDD is a collection of patterns and principles that aid in your efforts to build applications that reflect an understanding of and meet the requirements of your business. Outside of that, it's a whole new way of thinking about your development methodology. DDD is about modeling the real domain by fully understanding it first and then placing all the terminology, rules, and logic into an abstract representation within your code, typically in the form of a domain model. DDD is not a framework, but it does have a set of building blocks or concepts that you can incorporate into your solution.

You'll use this methodology when you build the case study application in Chapters 10 and 11. Some of the deeper aspects of DDD are examined in Chapter 4.

For more information on DDD, take a look at these books:

- *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans
- *Applying Domain-Driven Design and Patterns: With Examples in C# and .NET* by Jimmy Nilsson
- *.NET Domain-Driven Design with C#: Problem - Design - Solution* by Tim McCarthy

Behavior-driven Design (BDD)

You can think of BDD as an evolution of TDD merged with DDD. BDD focuses on the behavior of a system rather than just testing it. The specifications created when using BDD use the same ubiquitous language as seen in the real domain, which can be beneficial for both technical and business users.

The documentation that is produced when writing specifications in BDD gives readers an idea of how a system will behave in various scenarios instead of simply verifying that methods are doing what they are supposed to. BDD is intended to meet the needs of both business and technical users by mixing in aspects of DDD with core TDD concepts. BDD can be performed using standard unit testing frameworks, but specific BDD frameworks have emerged, and BDD looks to be the next big thing.

Again, if you download from `www.wrox.com` the code for the case study you will build in Chapters 10 and 11, you will find BDD specifications written to demonstrate the behavior of the system. Unfortunately, at the time of writing, there were no books on the subject of BDD. Therefore, my advice is to search for as much information on the Internet as possible on this great methodology.

SUMMARY

In this chapter, you were introduced to a series of design patterns, principles, and enterprise patterns that can be leveraged in ASP.NET applications.

- The Gang of Four patterns are 23 patterns catalogued into a book known as the *Design Patterns Bible*. These design patterns are solution templates to common recurring problems. They can also be used as a shared vocabulary in teams when discussing complex problems.
- Robert Martin's S.O.L.I.D. design principles form the foundations to which many design patterns adhere. These principles are intended to promote object-oriented systems that are loosely coupled, highly maintainable, and adaptable to change.
- Fowler's enterprise patterns are designed to be leveraged in enterprise-level applications. They include patterns to organize business logic, patterns to organize presentation logic, patterns to organize data access, as well as a host of base patterns that you can use throughout a system.

The introduction to these patterns and principles has been fairly high level, but as you progress through the book, you will find a deeper explanation of all of the concepts discussed in this chapter, and ASP.NET implementations from real-world scenarios that you can hopefully relate to and apply in your systems to solve problems.

The next chapter takes a closer look at the Gang of Four patterns that will be covered in this book. You will be introduced to the practical knowledge necessary to use a design pattern template and how to read a pattern.